Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Master's thesis

Deep Reinforcement Learning in Complex Structured Environments

Bc. Adam Volný

Supervisor: Mgr. Viliam Lisý, MSc., Ph.D.

25th May 2018

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Volný** Jméno: **Adam** Osobní číslo: **466730**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Umělá inteligence**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Hluboké posilované učení ve složitých strukturovaných prostředích

Název diplomové práce anglicky:

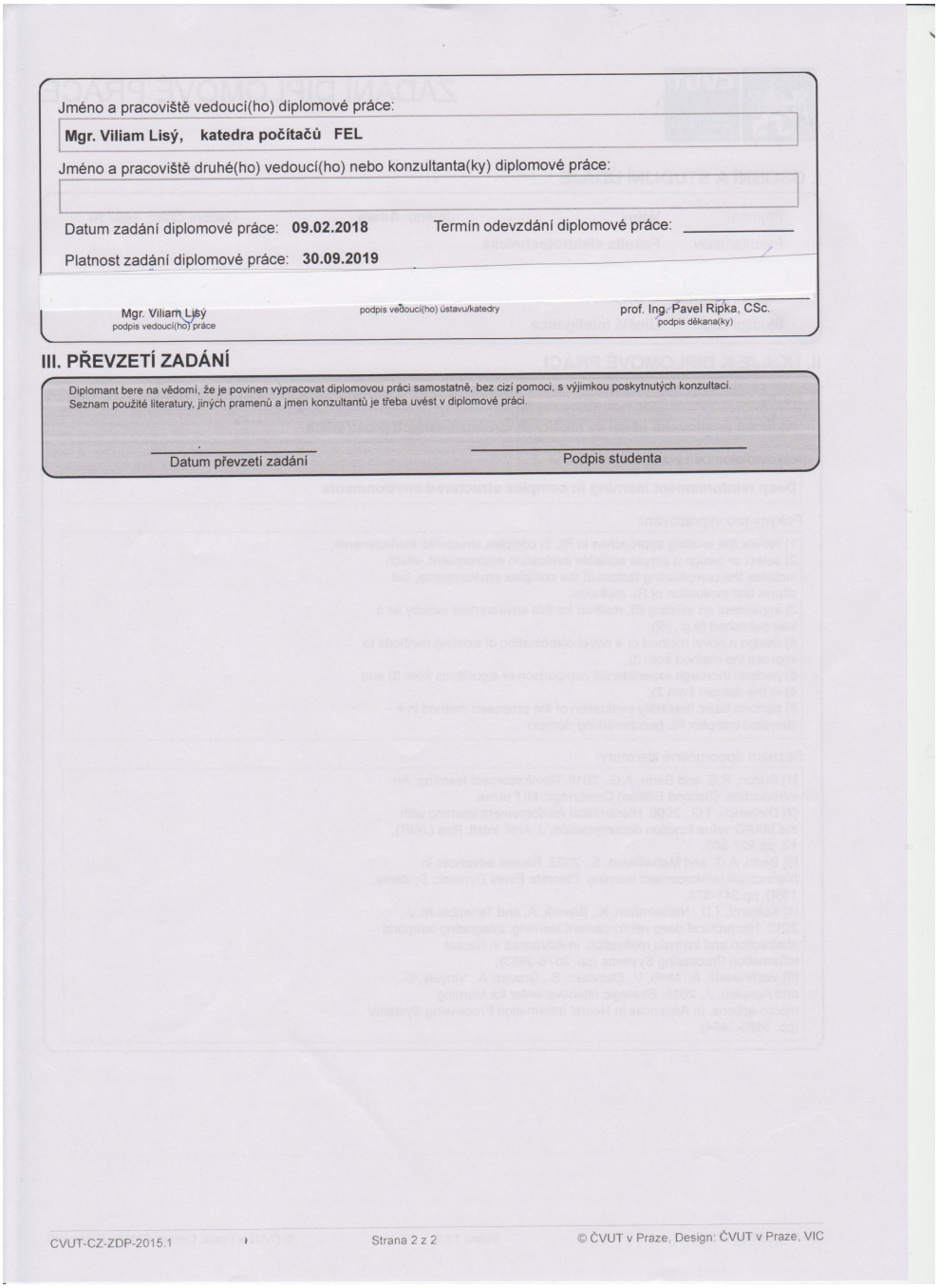
Deep reinforcement learning in complex structured environments

Pokyny pro vypracování:

1) review the existing approaches to RL in complex structured environments;
2) select or design a simple scalable evaluation environment, which includes the complicating factors of the complex environments, but allows fast evaluation of RL methods;
3) implement an existing RL method for this environment exactly as it was published (e.g., [5]);
4) design a novel method or a novel combination of existing methods to improve the method from 3);
5) perform thorough experimental comparison of algorithms from 3) and 4) in the domain from 2);
6) perform basic feasibility evaluation of the proposed method in a standard complex RL benchmarking domain

Seznam doporučené literatury:

[1] Sutton, R.S. and Barto, A.G., 2018. Reinforcement learning: An introduction. (Second Edition) Cambridge: MIT press.
[2] Dietterich, T.G., 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. J. Artif. Intell. Res.(JAIR), 13, pp.227-303.
[3] Barto, A.G. and Mahadevan, S., 2003. Recent advances in hierarchical reinforcement learning. Discrete Event Dynamic Systems, 13(4), pp.341-379.
[4] Kulkarni, T.D., Narasimhan, K., Saeedi, A. and Tenenbaum, J., 2016. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In Advances in Neural Information Processing Systems (pp. 3675-3683).
[5] Vezhnevets, A., Mnih, V., Osindero, S., Graves, A., Vinyals, O. and Agapiou, J., 2016. Strategic attentive writer for learning macro-actions. In Advances in Neural Information Processing Systems (pp. 3486-3494).



Jméno a pracoviště vedoucí(ho) diplomové práce:

Mgr. Viliam Lisý, katedra počítačů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **09.02.2018** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **30.09.2019**

Mgr. Viliam Lisý
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání

Podpis studenta

CVUT-CZ-ZDP-2015.1 Strana 2 z 2 © ČVUT v Praze, Design: ČVUT v Praze, VIC

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for non-profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on 25th May 2018 .....

Acknowledgements

I would like to kindly thank my family and close friends for their tolerance and support during writing this thesis and for standing by me during the difficult times in my life. I would like to thank my alma mater, Czech Technical University and Faculty of Electrical Engineering for providing me with exceptional education. And last but not least, I would like to thank my supervisor, for all the time he invested in me, for giving me the direction I so much needed and for the many thought provoking conversations that have enriched my worldview for years to come. I truly am deeply grateful to all of you.

Czech Technical University in Prague
Faculty of Electrical Engineering

Citation of this thesis

Volný, Adam. *Deep Reinforcement Learning in Complex Structured Environments.* Master's thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, 2018.

Abstrakt

Vytváření obecných agentů schopných naučit se užitečné rozhodovací strategie v prostředích reálného světa je obtížný úkol. Posilované učení je oblast, která se snaží tento problém vyřešit. Poskytuje obecný, rigorózně definovaný framework, ve kterém lze navrhovat algoritmy pro řešení problémů. Komplexní prostředí reálného světa mívají strukturu, které lze využít. Lidé jsou v tomto ohledu vyjímečně schopní. Protože různá prostředí mívají různou strukturu, vytváření agentů schopných tuto strukturu objevit a využít jí, bez předchozích znalostí o daném prostředí, je stále nevyřešený problém posilovaného učení. Hierarchické posilované učení je podobor, který se zabývá nalezením a využitím hierarchické struktury prostředí. V této práci implementujeme a studujeme dvě metody hierarchického posilovaného učení, Strategic Attentive Writer a FeUdal Networks. Představíme modifikaci modelu FeUdal Networks a ukážeme, že funguje lépe, než původní model v komplexním prostředí, navrženém na míru pro testování hierarchických agentů.

Klíčová slova Hierarchické posilované učení, hloubkové učení, umělé neuronové sítě, makro akce, options.

Abstract

Creating general agents capable of learning useful policies in real-world environments is a difficult task. Reinforcement learning is the field that aims to solve this problem. It provides a general, rigorously defined framework within which algorithms can be designed to solve various problems. Complex real-world environments tend to have a structure that can be exploited. Humans are extremely proficient at this. Because the structure can vary dramatically between environments, creating agents capable of discovering and exploiting such structure without prior knowledge about the environment is a long-standing and unsolved problem in reinforcement learning. Hierarchical reinforcement learning is a sub-field focused specifically on finding and exploiting a hierarchical structure in the environment. In this work, we implement and study two hierarchical reinforcement learning methods, Strategic Attentive Writer and FeUdal Networks. We propose a modification of the FeUdal Networks model and show that it performs better than the original model on a complex, customly designed environment.

Keywords Hierarchical reinforcement learning, deep learning, artificial neural networks, macro-actions, options.

Contents

List of Figures

List of Tables

Introduction

Motivation and Objectives

The field of Artificial Intelligence (AI) and more specifically Deep Learning (DL) is today in a state of rapid expansion. The recent successes have flooded the media which lead to a widespread interest in the field. Due to the nature of media to misinterpret information in favor of delivering exciting titles and controversies, the public image of DL is misleading. Despite the tremendous advancements in the narrow AI [3, 4, 5], there is much to be discovered in the field of Artificial General Intelligence (AGI). That can be vaguely summarized as an effort to create machines capable of thinking similarly to humans, including understanding of abstract concepts and common sense. The key ingredients of such machine would have to be strong language and conceptual skills, as language and abstract thinking are closely tied together [6]. If the endeavor were to succeed, that would probably mean another fundamental revolution of human civilization. Hopefully, not in the direction of dystopian societies depicted in many works of art but rather in the direction of usefulness and general prosperity.

This work's main focus is exploring the possibilities of abstraction in Deep Learning models. Recent developments of Convolutional Neural Networks (CNNs)[7] have lead to quite a successful reproduction of spatial abstraction patterns in neural networks (NNs), however, there is one more powerful tool that humans have at their disposal, temporal abstraction. Both work closely together to empower creation of a mental model of the world that, among other things, let's us reason about our actions and their consequences. The common scenario of supervised learning on dataset is *by design* not suitable for researching temporal abstractions.

That's why the Reinforcement Learning (RL) setup has been chosen for this work. It provides useful tools for modeling the real-world in the context of an effort to create intelligent agents. The agent-environment interaction happens in iterative steps and as such it simulates a sequence of events which

breeds a useful framework for learning both spatial and temporal abstractions in conjunction. Also the environment can be chosen arbitrarily to provide an advantage to agent that learns and utilizes meaningful spatial and temporal abstractions. Building agents that are able to learn to take advantage of the intrinsic hierarchical structure of complex environments falls into the field of Hierarchical Reinforcement Learning (HRL).

Problem Statement

In this work we focus on studying hierarchical deep reinforcement learning methods. We implement and study two recent approaches, Strategic Attentive Writer (STRAW)[8] and FeUdal Networks [2]. We design a novel complex environment that is easily scalable to the complexity of the agent and allows for computationally cheap evaluation. We aim to present an improvement of one of the methods and show, in a statistically rigorous manner, that it outperforms the original in the novel complex environment. Furthermore, we perform ablative analysis on a standard benchmarking reinforcement learning environment in an attempt to qualitatively validate the result.

Chapter 1

Background

In this chapter, we build the necessary theoretical and practical foundation for this work.

1.1 Markov Decision Process

For the sake of notational clarity and consistency, for reinforcement learning, we are using the mathematical notation, including most of the definitions, used in the second edition of [9].

In order to reason about agent-environment interaction, we first need a rigorous framework in which we can precisely define and describe it. The most widely used way in reinforcement learning is the Markov Decision Process (MDP). The interaction happens in a sequence of time-steps $t = 1, 2, 3, \ldots$, where at time-step t the agent receives the state of the environment $S_t \in \mathcal{S}$ and based on that produces an action $A_t \in \mathcal{A}(s)$, where $s \in \mathcal{S}$. In turn, the agent receives a *reward* $R_t \in \mathcal{R} \subset \mathbb{R}$ that depends on its action and the present state of environment. Semantically the reward reflects agent's performance (the higher the better) in the context of a given problem. The next state of the environment S_{t+1} depends only on the present state S_t and the agent's action A_t.

The end of the interaction sequence (an *episode*) is marked by reaching a terminal state $s_{terminal}$, the following relation holds, $\mathcal{S}^+ = \mathcal{S} \cup \{s_{terminal}\}$. Note that S_t, A_t and R_t are random variables. In this work we further consider only a *finite* MDP, in which all the sets $\mathcal{S}, \mathcal{A}(s)$ and $\mathcal{R}$ are finite, we will still call it MDP for the sake of brevity.

Trajectory is a sequence of states, actions and rewards that describe an interaction between the agent and the environment:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{1.1}$$

The dynamic of the environment can be described by a function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A}(s) \to [0, 1]$, defined as follows,

$$p(s_t, r_t|s_{t-1}, a_{t-1}) = Pr\{S_t = s_t, R_t = r_t|S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}\} \quad (1.2)$$

Where $s', s \in \mathcal{S}, r \in \mathcal{R}$ and $a \in \mathcal{A}(s)$. Then, the *state-transition function* is,

$$p(s_t|s_{t-1}, a_{t-1}) = \sum_{r \in \mathcal{R}} p(s_t, r|s_{t-1}, a_{t-1}) \quad (1.3)$$

Note that both aforementioned functions are well defined discrete probability distributions because we are considering only the *finite* MDP case.

The adjective Markov in the name is due to the next reward and the next state following a Markov Property. In plain words it means that they are both conditioned solely on the present state and action and not on any of the previous ones. Formally we could express this as,

$$Pr(S_t = s_t|S_{t-1}, A_{t-1}) = Pr(S_t = s_t|S_{t-1}, A_{t-1}, S_{t-2}, A_{t-2}, \ldots) \quad (1.4)$$
$$Pr(R_t = r_t|S_{t-1}, A_{t-1}) = Pr(R_t = r_t|S_{t-1}, A_{t-1}, S_{t-2}, A_{t-2}, \ldots) \quad (1.5)$$

A system that satisfies the Markov property is often called *memoryless*.

1.1.1 Expected Return

The quantity that we are seeking to maximize in solving a problem described by MDP is an expectation of a random variable called return. At time-step t for a sequence of rewards, *return* is quite intuitively defined as,

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \ldots + R_T \quad (1.6)$$

Where T is the index of a last step. That brings up an important notion; many problems solved by reinforcement learning are so called *episodic tasks*, which means that they are carried on in episodes consisting of a finite number of steps [1] (an exemplary task would be chess where each game ends in a finite number of turns with one winning player or a tie).

We call the last state in a trajectory a *terminal state*, let's denote it as $s_{terminal}$. It's useful to differentiate two possibilities for the set of states, $\mathcal{S}$ and $\mathcal{S}^+$, where $\mathcal{S}^+ = \mathcal{S} \cup \{s_{terminal}\}$, allowing us to distinguish a set of all non-terminal states from a set of all states.

We consider a single terminal state because there is no need to have more. No decision is made based on it, thus its value is purely nominal and the outcome of the task may be reflected by the reward accompanying the final state. The initial state can be same in every episode or it can be sampled according to some distribution, let's define a set of all the possible initial states as $\mathcal{S}_{init}$ and the distribution over them as $p(s_{init}) = Pr(S_{init} = s_{init})$.

[1] All the tasks studied in this work are *episodic*

Some tasks, however, are not episodic in nature, consider controlling a valve in an automated industrial process, such task might require an indefinite control thus having an infinite horizon. Then we could encounter a problem as G_t could possibly be infinite and therefore we wouldn't be able to compute its expectation.

To resolve this, a *discount factor* $\gamma \in (0, 1)$ is introduced. It discounts each subsequent reward by increasing amount, reflecting an assumption that present reward is more valuable than reward in the future. The *return* at time-step t is then defined as,

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1.7}$$

Two key observations here are that the coefficients form a geometric series and that the reward is bounded (it comes from a finite set, every finite subset of real numbers is bounded). Those together imply that G_t will indeed be finite. Proof is straightforward, let $M \in \mathbb{R}$ be the upper bound of the set $\mathcal{R}$,

$$\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \leq \sum_{k=0}^{\infty} \gamma^k M = \frac{1}{1-\gamma} M \tag{1.8}$$

Using the formula for computing the sum of a geometric series which holds because $|\gamma| < 1$. Therefore G_t will always be bounded and thus we can maximize its expectation in an attempt to solve the task underlying the MDP.

Even though we are not going to be focusing on tasks with infinite horizon in this work, the notion of *discounted return* is still going to be very useful later on, even in the finite horizon case.

1.1.2 Policy

In order to formalize the agent's decision process, we introduce a function called *policy* which maps the present state s to each possible action a and its probability, formally $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$,

$$\pi(a|s) = Pr(A_t = a|S_t = s) \tag{1.9}$$

An important observation here is that the action of the agent is only conditioned on the present state. It might not be immediately clear why that would be complete. However, due to the Markov property, the dynamic of the MDP is entirely decided by its current state, therefore, considering previous states adds no relevant information whatsoever. This means that the policy also follows the Markov property but rather as a consequence.

Now, that we have defined the notion of *policy*, we need a measure of quality for a given policy, otherwise we wouldn't be able to compare it to other policies and decide which one is best suited to solve our task. For that

purpose, we will reintroduce the *expected return* but this time we'll call it *value function* $v_\pi : \mathcal{S} \to \mathbb{R}$, defined as,

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] \tag{1.10}$$

It denotes the expected return in state s for given policy π. Now, we can finally formalize the problem we are trying to solve:

$$\begin{aligned} \pi^* = \underset{\pi:\mathcal{S}\times\mathcal{A}\to[0,1]}{\mathrm{a}rgmax} \; & \mathbb{E}[v_\pi(s_{init}) | S_{init} = s_{init}] \\ s.t. \sum_{a\in\mathcal{A}(s)} & \pi(a|s) = 1, \forall s \in \mathcal{S} \\ & \pi(a|s) \geq 0, \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s) \end{aligned} \tag{1.11}$$

Where π^* denotes *optimal policy* which is such that it maximizes the expected value function of initial state given the distribution over initial states. Also, the *policy* π itself must be a probability distribution.

It is generally intractable to find the optimal policy analytically, iterative dynamic programming methods can be employed (value iteration, policy iteration and their extensions [9]). When even those cannot be used (due to their space and time complexity), the problem is solved using approximation methods. That is the approach we take in this work.

1.1.3 Partially Observable MDP

The framework we introduced relies on one very strong assumption; that the state of the environment can be fully observed by the agent. Even though this is the case in some environments (fully observable games as chess, tic-tac-toe etc.), it does not hold for many real-world environments. In the partially observable case we assume that the environment's state is a latent variable that cannot be observed by the agent at all. The agent then receives *observations* according to a conditional probability distribution, conditioned on the present state of the environment. This is called Partially Observable MDP (POMDP). To define it formally, let $\mathcal{S}$ be the latent state space and $\mathcal{X}$ the space of observations, we can then write,

$$p(x_t|s_t) = Pr\{X_t = x_t | S_t = s_t\} \tag{1.12}$$

Where $X_t \in \mathcal{X}$ and $S_t \in \mathcal{S}$. It then makes sense to define the agent's policy with respect to the whole history of seen observations as all those carry useful information about the environment. We could express it as follows,

$$\pi(a_t|x_t, x_{t-1}, \ldots) = Pr\{A_t = a | X_t = x_t, X_{t-1} = x_{t-1}, \ldots\} \tag{1.13}$$

No matter which framework, MDP or POMDP, we are going to use, we will always talk about *observations* and denote them as x_t, in the case that

we would be dealing with MDP, we can simply set $\mathcal{X} = \mathcal{S}$ and trivially express MDP as POMDP.

We have laid a formal foundation for defining the problems we will be solving, the interface of agent itself and also ways of measuring its performance. However, we have not yet talked about how to devise its inner workings in a way that would let us find sufficiently good solution. For that we will be using artificial neural networks which are the subject of the next section.

1.2 Artificial Neural Networks

Artificial neural networks are a class of biologically inspired general purpose approximators. They consist of many interconnected units that perform local computations. The units are called *neurons* and connections between pairs of neurons are weighted. The weight decides how much and in which direction (excitatory or inhibitory) will given neuron influence its descendant. For a network with a given topology, tuning those weights is crucial in modifying the internal dynamics and behavior of the network. The key question is how to find such configuration of weights that makes the neural network behave as we would like. The answer is not straightforward and is still an objective of a very active research.

The neural network fits into our context as the component that actually provides the specific policy for the agent based on the observations from environment. Let's describe the neural networks formally (we always mean artificial neural networks, unless stated otherwise).

1.2.1 Artificial Neuron

A single neuron in the network is a function, that maps inputs from multiple neurons to a single output. Let x_i be the i-th input of a neuron, for $i = 1, 2, 3, \ldots, n$, w_i be the weight of the connection from the i-th neuron and bias b. We can then compute a so called *potential* as,

$$\xi = \sum_{i=1}^{n} w_i x_i + b \tag{1.14}$$

The potential is then fed through a so called *activation function* $\varphi : \mathbb{R} \to \mathbb{R}$ to obtain the neuron's output,

$$y = \varphi(\xi) \tag{1.15}$$

The activation function is the central computational element in a neural network and by tuning the weights we change what the neuron computes in relation to other neurons.

The commonly used activation functions are *linear*, *tanh*, *logistic sigmoid*, *rectified linear unit*.

1.2.2 Network Topology

The neurons are usually organized in multiple layers, where each neuron is connected to every other neuron in the preceding and descending layer. The first layer is fed inputs whereas the last layer provides the output of the network. Such architecture is called *fully connected* and the individual layers can be called *dense*. Note that there are endless possibilities for different topologies.

Because each neuron is essentially a function, when they are stacked together, the whole neural network is a function. If the input layer takes n inputs and the output layer gives m outputs, then the network can be described as a function $f : \mathbb{R}^n \to \mathbb{R}^m$. This is the most general definition we can assume, as a parametrized mapping between input and output.

1.2.2.1 Dense Layer

Dense layers are the most basic of types. They assume an input vector $x \in \mathbb{R}^n$, all the inputs are connected to all the units in the layer, thus the name dense. The computation of the layer can be described by dot product,

$$y = f(W^T x + b), \tag{1.16}$$

where $W \in \mathbb{R}^{n \times m}$ is a weight matrix with $W_{(i,j)}$ being the weight of connection between i-th input and j-th unit. $b \in \mathbb{R}^m$ is a bias vector and $f : \mathbb{R}^m \to \mathbb{R}^m$ is the activation function for given layer (for linear layer this would be an identity function).

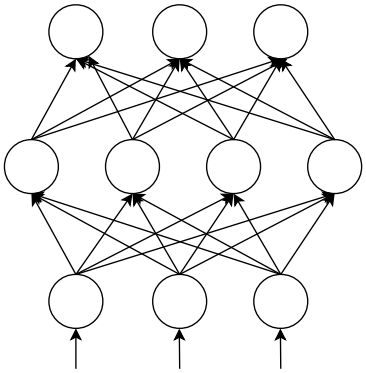


Figure 1.1: Artificial neural network topology with the first layer as an input and then two densely connected layers where the second one could be used as an output.

1.2.2.2 Convolutional Layer

Convolutional layers let us take advantage of the fact that some information in the input may be invariant to its location in the input vector. An example of this would be an image classification task, it usually doesn't matter *where* the object we are trying to classify is located in the image, what matters is its presence. The dense layer is not the best tool for this as all the weights are fixed on specific input point.

The convolutional layer introduces spatial parameter sharing between units. Each convolutional layer is composed of so called *filters* (or *kernels*), which are tensors of weights that have the number of dimensions as the input but not the same shape. For illustration, let us consider 1D convolutional layer (See Figure1.2). That assumes 2D input tensor, so the weight tensor will be a matrix. The last dimension is always matched in the weight tensor, whereas the rest is smaller than the input tensor. The first dimensions are chosen arbitrarily based on the task, so let us consider e.g. filter size of 3. If we had an input matrix $X \in \mathbb{R}^{m \times n}$, then our weight tensor would have shape $W \in \mathbb{R}^{3 \times n}$.

The way output of the layer is computed is that the filter is slid over the input over all but the last dimension with given *stride*. For each position of the filter, we simply multiply each element in the input with its corresponding weight (this will cover only a small portion of X), then we sum those weighted inputs up and add *bias*. This way we have obtained *potential* of our neuron now what remains is to apply the $activation function$ to obtain our output. The most commonly used one is the Rectified Linear Unit (ReLU) for its computational simplicity, yet sufficient expressiveness (it is still non-linear) [3].

What was described above is the case for a single output unit and for a single filter. In order to compute the output of the same filter but for the rest of the units, the filter is slid over the input dimensions with the step size given by *stride* (it is defined for each dimension separately so the step sizes in different dimensions can vary). For our example, let's select stride of 1 which means that the convolutional filter is always slid by one in the first dimension. However, if we use only a single filter, notice that our output is only a one-dimensional tensor while we started with a two-dimensional one. This might be a desired property in some cases but for most applications we want the output have the same number of dimensions as the input because that means that we could stack multiple convolutional layers on top of each other.

To get the output with the same number of dimensions as the input, we simply use multiple filters of identical size and then stack them over the last dimension. This way the number of dimensions is retained.

If we consider an input tensor with N dimensions, we can notice that by applying the $N-1$ dimensional convolutional layer, our output is always shrunk in the first $N-1$ dimensions because we apply the filters only where

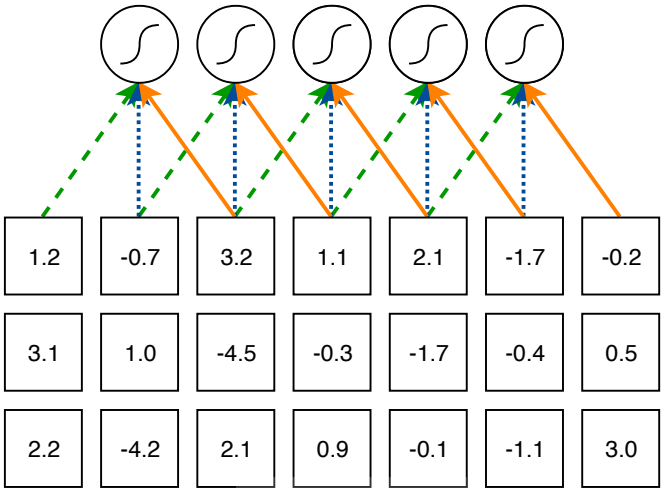


Figure 1.2: Illustration of the 1D convolutional filter for 2D input. The filter has size of 3 and stride of 1. The input's second dimension is 3 (rows are the second dimension) so the weight matrix has shape 3×3. All the illustrated connections that share the same color and style also share the same weight. Note that only one third of the connections is shown, there should also be analogous connections from the second and third rows of the input but those are omitted for clarity. However, even if those were included, the output units still would be arranged in just one dimension. To retain the original two dimensions, we would have to include in the illustration multiple filters.

they are covering the input completely. However, in some applications we want to retain not only the number of dimensions but also the size in the first $N-1$ dimensions as well. To do that we use $zero-padding$. Essentially, we put the filter over a position in the input that normally would not be valid and we fill all the absent values with zeros. See Figure1.3. [10]

1.2.2.3 Recurrent Layer

An option we have not yet talked about are *recurrent neural networks* (RNNs). The notion of a neural network so far assumed a stateless system. We have a function that takes input data and outputs some vector, however, the present output only depends on the present input and not on any of the previous ones. This may be desirable for, let's say, classifying images where each image is unrelated and evaluated separately (i.e. data points presented are i.i.d.). However, when we wish to use our neural network on a task that is sequential and where remembering the previous input points helps getting the right output in the future, a recurrent architecture is much more suitable. Example of such task could be speech recognition [11], where the probability of different

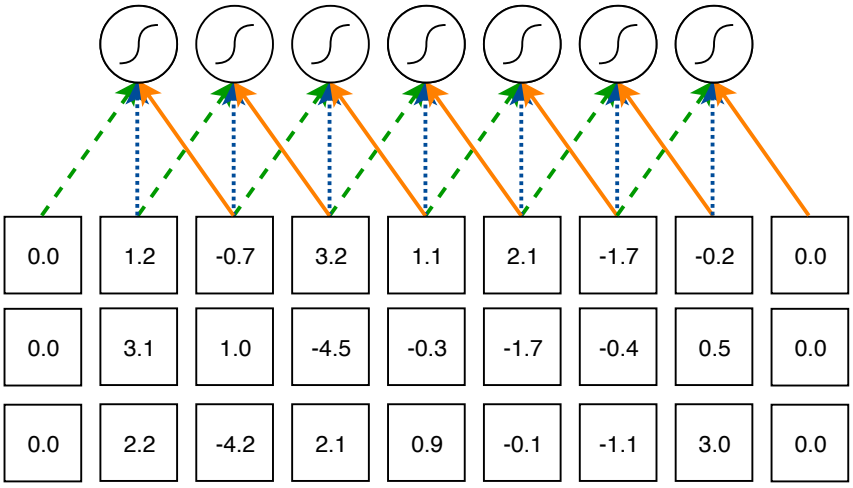


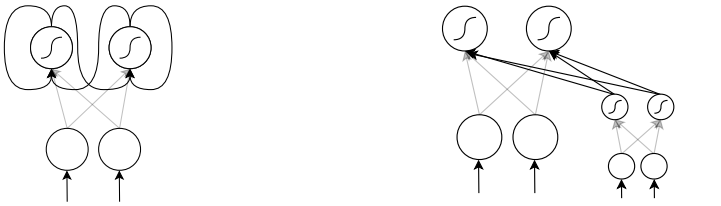
Figure 1.3: An extension of Figure1.2. As we can see, now the number of output units match the input's first dimension exactly. Thus if we stacked multiple such layers, the number of units (within the first dimension) would always remain the same.

syllables and sounds is strongly shaped by what has been already said. Such problem calls for an architecture that maintains and evolves its internal state in some sense and is able to *remember*.

You can think of a recurrent layer as being a common dense layer but on top of the connection to the previous and next layer it is also connected to its own output but from the previous time-step. This means that state is maintained between time-steps and therefore the previous inputs influence the future outputs. The schema of the time-lagged connections could be each neuron only feeding itself its previous state but a more common approach is to connect the previous state to the present *densely*, i.e. each unit influences all the units in the next time-step. This makes the model more expressive.

Because the internal state of the network evolves during computation, we have to maintain the entire history for training the network. Handling the lagged signal in a dynamic manner can be troublesome and prone to implementation bugs. That's why a common approach is to prepare an unrolled model of the recurrent neural network. If we fix *time horizon*, which we treat as the maximum number of steps the network can remember to the past, we can create a large feed forward network that repeats its blocks for each time-step until the time horizon. The weights are completely shared between blocks which means that it indeed is equivalent to recurrently feeding past values through a single layer. An illustration of a single step unroll can be seen in Figure1.4, Figure1.4a depicts the network with recurrent connections, 1.4b shows how would a single unroll step look.

It can also sometimes be difficult to imagine or grasp how to treat recurrent neural network in terms of data, learning, etc. For this, thinking about the unrolled version of the network can be of a great help.



(a) Recurrent layer (b) Unrolled recurrent layer

Figure 1.4: Illustration of single-step graph unrollling.

A popular activation function for recurrent networks used to be the *logistic sigmoid* function. However, there are some serious issues with it in the context of RNNs. Details of training neural networks are discussed later but in principle, after the network was used for inference an error in the estimate is computed. Then this error is distributed through the network through all the previous time-steps. However, the way sigmoid function behaves has a consequence of diminishing the error signals. The more steps in the past, the weaker the learning signal is. Therefore, such vanilla RNNs have difficulties with long-term dependencies in the data. They can operate only within a moderate time-lag. The issue is called the *vanishing gradient* problem.

1.2.2.4 LSTM Layer

To solve the vanishing gradient problem a novel architecture was devised in 1997 by Hochreiter and Schmidhuber [1]. So far, every neuron simply computed dot product between the input vector and the weight vector, added bias term and then fed the value through some mostly non-linear activation function. LSTM units, however, are more complicated.

For a time-step t, LSTM unit consists of a cell state c_t, *input*, *output* and *forget* gate that control the input into the cell state, the output of the unit and the persistence of the cell state respectively. The unit processes input vector x_t, previous output h_{t-1} and previous cell state c_{t-1} into the current output h_t and the current cell state c_t. The parameters of LSTM unit are eight weight matrices and four bias terms, W_q, U_q and b_q), for $q \in \{i, o, f, c\}$ which stand for input, output, forget gate and cell state respectively. The

precise dynamics of the LSTM unit are as follows:

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \tag{1.17}$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \tag{1.18}$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \tag{1.19}$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \tag{1.20}$$
$$h_t = o_t \circ \sigma_h(c_t) \tag{1.21}$$
$$\sigma_g = \frac{1}{1 + \exp -x} \tag{1.22}$$
$$\sigma_c = \sigma_h = \frac{1 - \exp -2x}{1 + \exp -2x} \tag{1.23}$$

For h LSTM units in a layer and d size of the input, the dimensions are:

$$W_f, W_i, W_o, W_c \in \mathbb{R}^{h \times d} \tag{1.24}$$
$$U_f, U_i, U_o, U_c \in \mathbb{R}^{h \times h} \tag{1.25}$$
$$b_f, b_i, b_o, b_c \in \mathbb{R} \tag{1.26}$$
$$x_t \in \mathbb{R}^d \tag{1.27}$$
$$f_t, i_t, o_t, c_t, h_t \in \mathbb{R}^h \tag{1.28}$$

We can see that the activation function for all the gates is logistic sigmoid and for the cell and output state, its hyperbolic tangent. That means that all the gate values f_t, i_t, o_t are bounded by 0 and 1, so in a sense, they approximate boolean logic flow and gates in a continuous and differentiable manner. The cell output h_t is then bounded by -1 and 1 while the cell state c_t does not have an obvious bound due to its incremental nature.

The rationale behind the name Long-Short Term Memory comes from the fact that the model approximates short term memory which can persist for long periods of time. The information is maintained and encoded in the activity of the network, not in the connection weights, which is intuitively analogous to the short term and the long term memory in the brain, respectively.

The basic model has several modifications but none of them significantly outperforms the vanilla version that was introduced here, therefore it is what we will use in this work. [12]

LSTM networks were successfully applied to tasks such as natural language translation and image captioning (describing images by sentences) [13] and continue to be widely used.

One of the fundamental reasons why LSTM architecture works so well in practice is that it solves the *vanishing* and *exploding* gradients problem [1]. That enables the learning algorithm discover and take advantage of long-term dependencies in the input sequence.

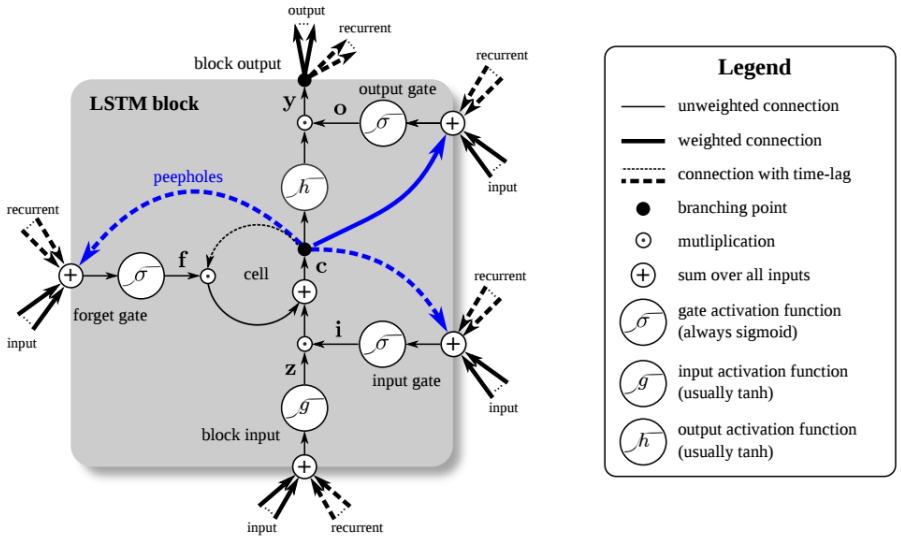


Figure 1.5: LSTM schematic, source: [1].

1.2.2.5 Softmax Layer

In machine learning it is necessary to meaningfully cope with uncertainty. The world we are approximating is stochastic. Therefore, when we train a classifier it is often not quite enough to just consider the best guess of the classifier but it is crucial to also know its confidence. If the classifier were just a little bit more sure about one choice than the other, the output would be the same as if it were the sure choice.

For this purpose, the softmax function works well. It takes a real vector $z \in \mathbb{R}^n$ and *squashes* its values in such a way that they are all *positive* and the vector sums up to 1.

$$SoftMax(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)} \tag{1.29}$$

Those two properties together mean that the softmax of a vector can be interpreted as a probability distribution. In the neural network, the softmax layer is simply a densely connected layer that uses the softmax function as the activation function. Note that this activation function stands apart from all the other commonly used activation functions. The output of each softmax unit depends not only on the input but also on the output of all the other softmax units in the layer, because of the normalizing term $\sum_{j=1}^{n} \exp(z_j)$. Therefore, when gradient flows through one of the softmax units, it flows through all of them. The output of the softmax layer for input x, weights W and bias b, is then given as,

$$y = SoftMax(Wx + b) \tag{1.30}$$

The probability distribution on the network output can be interpreted as approximation of the conditional $Pr(Y = y|X = x, \theta)$ for different label vectors y and model parameters θ. X is the input random variable, Y is the random variable for one-hot encoded label vector sampled according to the distribution given by the softmax output. Then, for the classification task, one can maximize the likelihood of sampling the correct labels w.r.t. the model parameters.

If we consider n classes, training samples $(x^1, y^1), (x^2, y^2), \ldots, (x^N, y^N)$ (where $y^i \in \mathbb{R}^n$ is *one-hot* encoded correct label), and a model parameter vector θ, we can express the likelihood of classifying the samples correctly as follows,

$$L = \prod_{i=1}^{N} Pr(Y = y^i|X = x^i, \theta) \tag{1.31}$$

$$\tag{1.32}$$

The sum simply chooses the probability of the softmax unit that corresponds to the correct label. Now in order to find the optimal parameters we would like to maximize this value. Because the natural logarithm is monotone increasing function, minimizing the negative log-likelihood instead preserves all the solutions.

$$-\ln L = -\ln \prod_{i=1}^{N} Pr(Y = y^i|X = x^i, \theta) \tag{1.33}$$

$$= \sum_{i=1}^{N} -\ln Pr(Y = y^i|X = x^i, \theta) \tag{1.34}$$

$$\tag{1.35}$$

This is where *categorical cross-entropy* loss function comes from. Categorical because the support of the distributions is discrete. Cross-entropy is a term from information theory that describes relationship between two distributions, specifically the average number of bits needed to identify an event drawn from a set. Therefore we can express the optimal parameter vector as,

$$\theta^* = \underset{\theta}{\mathrm{a}rgmin}(-\ln L) \tag{1.36}$$

$$\tag{1.37}$$

1.2.3 Learning Weights

To be clear, the purpose of using a neural network is approximating a function. In practice, usually some complicated high-dimensional function that is not known explicitly. Let θ be a vector of all parameters of a given neural network. In order to be able to reason about a relative quality of θ we introduce an *objective function*, in machine learning often called *loss function*. It is a function that maps the parameter vector to a single real value which we are trying to minimize. Finding the global minimum analytically in a closed form might not be possible and in vast majority of cases it is intractable. One of the common solutions to this are iterative optimization algorithms, namely gradient descent and stochastic gradient descent.

1.2.3.1 Gradient Descent

The Gradient Descent algorithm is based on an assumption that for a function f, in a sufficiently small neighborhood of a point, we can approximate the function by a hyperplane. If the function f is then *well behaved* in the neighborhood, taking a small step in the direction of the steepest descent on the hyperplane will also decrease the value of the function f. Therefore, if we wish to find the minimum of the function f, repeated small decreases will converge (when the step size decreases) to a local minimum.

In order to find the approximating hyperplane, we use derivatives, which by definition give us the slope of the hyperplane,

$$f'_{x_0}(x) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0} \tag{1.38}$$

If we consider a single valued real function for simplicity, we can see that in the fraction, we take a ratio between the difference of values on the y-axis and the difference of values on the x-axis and that is by a very definition a slope of line that crosses both points on a real plane $(x, f(x))$ and $(x_0, f(x_0))$. In the limit, those points are moved infinitely close and that gives us the notion of the tangent of a function. This concept generalizes naturally to multivariate functions and hyperplanes.

Let $J(\theta)$ be a single valued loss function for the network given by parameter vector θ, differentiable w.r.t. θ. $D = (x^1, y^1), (x^2, y^2), \ldots, (x^N, y^N)$ is the dataset of samples that we are trying to correctly classify. Then, we propose trivial parameter update rule using step size α,

$$\theta_t = \theta_{t-1} - \alpha \nabla_\theta J(\theta, D) \tag{1.39}$$

This can be devised into a simple algorithm called Batch Gradient Descent (Algorithm 1), as detailed below. The *batch* adjective is used because we compute the gradient over the whole dataset, so we do all the updates in a single batch. Another way to think of this is that we compute the gradient

for each sample from D separately and accumulate them. After iterating over the whole dataset, we perform a single parameter update and reset the accumulators.

Algorithm 1 Batch Gradient Descent

Input: dataset D, classifier parametrized by θ, loss $J(\theta, D)$ differentiable w.r.t. θ, step size α, maximum training epoch t_{max}

1: initialize epoch counter $t \leftarrow 1$
2: $\theta_0 \leftarrow$ sample $\mathcal{N}(0, 1)$ ▷ arbitrary random initialization
3: **repeat**
4: $\quad \theta \leftarrow \theta - \alpha \nabla_\theta J(\theta, D)$
5: $\quad$ possibly decay the learning rate α
6: $\quad t \leftarrow t + 1$
7: **until** $t > t_{max}$ or $J(\theta, D)$ converges

1.2.3.2 Stochastic Gradient Descent

Stochastic Gradient Descent (Algorithm 2) is a different flavor of the previous algorithm as here we update the parameters right after iterating over a sample. Therefore, we perform many little updates. The term stochastic comes from the fact that we are approximating the true direction of the gradient by using only separate samples. However, it is clear that a single sample drawn is not a very stable estimator of the true gradient, therefore the approximation variance is likely to be large.

To avoid bias from encountering some arbitrary ordering of data (due to factors such as the way the data were collected and organized), the samples are shuffled. Either once at the beginning of the training, or after every epoch, or the samples may be drawn randomly at every step. We present here a generic version that does not specify shuffling.

1.2.3.3 Mini-Batch Gradient Descent

Last version of the Gradient Descent algorithm we present here is the Mini-Batch Gradient Descent (Algorithm 3). It is a compromise between the Gradient Descent and Stochastic Gradient Descent algorithms. It aggregates updates into larger batches but still updates frequently, therefore it has reasonably low variance while still maintaining higher speed of convergence. Once again, it is advisable to shuffle the batches before/during the training to avoid pathological cases.

The dataset comprises of batches $D = B_1, B_2, \ldots, B_{|D|}$ and each batch of samples $B_i = (x^1, y^1), (x^2, y^2), \ldots, (x^1, y^{|B_i|})$.

Algorithm 2 Stochastic Gradient Descent

Input: dataset D, classifier parametrized by θ, loss $J(\theta, x, y)$ differentiable w.r.t. θ, step size α, maximum training epoch t_{max}

initialize epoch counter $t \leftarrow 1$
$\theta_0 \leftarrow$ sample $\mathcal{N}(0, 1)$ ▷ arbitrary random initialization
repeat
    for $(x^i, y^i) \in D$ **do**
        $\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta, x^i, y^i)$
    possibly decay the learning rate α
    $t \leftarrow t + 1$
until $t > t_{max}$ or $J(\theta, D)$ converges

Algorithm 3 Mini-Batch Gradient Descent

Input: dataset D, classifier parametrized by θ, loss $J(\theta, B)$ differentiable w.r.t. θ, step size α, maximum training epoch t_{max}

initialize epoch counter $t \leftarrow 1$
$\theta_0 \leftarrow$ sample $\mathcal{N}(0, 1)$ ▷ arbitrary random initialization
repeat
    for $B_i \in D$ **do**
        $\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta, B_i)$
    possibly decay the learning rate α
    $t \leftarrow t + 1$
until $t > t_{max}$ or $J(\theta, D)$ converges

This version of the Gradient Descent algorithm is the one used for training neural networks and as such we use it in this work as well.

1.3 Reinforcement Learning Methods

In this section we will introduce the basic reinforcement learning approaches used to solve MDPs.

1.3.1 Action Value Methods

First class of RL algorithms are *action-value* methods. They are built on estimating the expected return with respect to given state and actions available in this state. Then, usually the action maximizing expectation is selected and carried on in the environment, this approach is called *greedy policy*. Let us define function $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that expresses our expectation for a given action in a given state,

$$Q_\pi(s, a) = \mathbb{E}[G_t | S_t = s, A_t = a]. \tag{1.40}$$

Then, the *greedy policy* can be expressed as,

$$\pi(a|s) = \underset{a}{\mathrm{argmax}}\, q(s, a). \tag{1.41}$$

In order to support exploration of the whole state space in a more uniform fashion, an *ε-greedy* policy can be utilized. Which means that with probability ε, where $\varepsilon \in (0, 1)$, a random action is selected instead of the greedy one.

There are many methods used to approximate $Q(s, a)$ but none is used in this work therefore discussing it is beyond the scope, for details about *action-value* methods, see [9].

More useful to us is an entity called the *value-function* which is the expected return in a given state over all the actions, weighted by the policy,

$$v_\pi(s) = \sum_a \pi(a|s) Q(s, a). \tag{1.42}$$

Value function is crucial to all the methods used here.

1.3.2 Basic Policy Gradient Methods

Policy gradient methods differ from the action value methods in that they provide meaningful stochastic policies (as *action-value* methods don't have a meaningful way of producing stochastic policies) and that mostly means stronger theoretical guarantees [9].

Let us consider a policy π parametrized by the vector θ,

$$\pi(a|s, \theta) = Pr(A_t = a | S_t = s, \theta_t = \theta) \tag{1.43}$$

In order to achieve such policy that solves our task we need some performance measure $J(\theta)$ that is differentiable with respect to the parameter vector θ.

Then to maximize the performance, an approximate gradient descent can be used.

$$\theta_{t+1} = \theta_t + \alpha \nabla \hat{J}(\theta_t), \tag{1.44}$$

where $\nabla \hat{J}(\theta_t)$ stochastically estimates the gradient w.r.t. θ. How to approximate it is discussed closely in upcoming sections.

Methods discussed here also approximate the value function $v(s, w)$, where w is the parameter vector of the value function approximation. Such methods are called *actor-critic* methods. The *actor* is the policy and the *critic* is the *value function*.

Because we are considering the case of discrete time and discrete actions, our performance measure is the expected return in the initial state (at the start of the episode).

$$J(\theta) = v_{\pi_\theta}(s_0) \tag{1.45}$$

In order to compute the derivative of $J(\theta)$ we take advantage of the **policy gradient theorem**, which applies to our case as,

$$\nabla J(\theta) = \nabla v_{\pi_\theta}(s) = \mathbb{E}_\pi \left[\sum_a q_\pi(S_t, a) \nabla \pi(a|S_t) \right] \tag{1.46}$$

$$= \mathbb{E}_\pi \left[\sum_a \pi(a|S_t) q_\pi(S_t, a) \frac{\nabla \pi(a|S_t)}{\pi(a|S_t)} \right] \tag{1.47}$$

$$= \mathbb{E}_\pi \left[q_\pi(S_t, A_t) \frac{\nabla \pi(A_t|S_t)}{\pi(A_t|S_t)} \right] \tag{1.48}$$

$$= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t|S_t)}{\pi(A_t|S_t)} \right] \tag{1.49}$$

This entity intuitively makes sense. Vector $\nabla \pi(A_t|S_t)$ is the direction in the parameter space that leads to the steepest increase of the probability of action A_t on future visits of state S_t. Now it is scaled by the return G_t which intuitively results in reinforcing the actions that lead to better results. It is also inversely scaled by the present probability of given action in a given state. Without it, actions that are performed more often than others would have more according parameter updates of the same magnitude than others. That could lead to a scenario, where low-return, high-probability action is eventually reinforced over a high-return low-probability action, which would be incorrect. The correct way would be to reinforce the high-return action and increase its probability. The proof of policy gradient theorem is beyond scope of this work and is presented in [9].

1.3.2.1 REINFORCE

REINFORCE [14] is a policy-gradient algorithm that approximates (1.49) for its the parameter updates.

Algorithm 4 REINFORCE [9]

Input: $\pi(a|s,\theta)$ policy differentiable w.r.t. its parametrization, learning rate α

1: **function** LEARN(θ)
2: generate episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ using $\pi(\cdot|\cdot,\theta)$
3: **for** $i \leftarrow 1$ to $T-1$ **do**
4: $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
5: $\theta \leftarrow \theta + \alpha G \nabla \ln \pi(A_t|S_t,\theta)$

The routine is repeated until convergence. In this algorithm the gradient is approximated by using an *eligibility trace* from an episode generated using the policy $\pi(a|s,\theta)$.

1.3.2.2 REINFORCE with Baseline

Even though the algorithm 4 is correct and will converge to a local minimum, it might take a long time [9]. In order to improve this we will reduce the variance of the gradients using a so called *baseline*. Let us follow up on and modify (1.46),

$$\nabla J(\theta) = \mathbb{E}_\pi \left[\sum_a (q_\pi(S_t,a) - b(S_t)) \nabla \pi(a|S_t) \right] \tag{1.50}$$

$$= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t) \right] - \mathbb{E}_\pi \left[\sum_a b(S_t) \nabla \pi(a|S_t) \right] \tag{1.51}$$

$$= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t) \right] - \mathbb{E}_\pi \left[b(S_t) \underbrace{\nabla \sum_a \pi(a|S_t)}_{\nabla 1 = 0} \right] \tag{1.52}$$

$$= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t) \right] - 0 \tag{1.53}$$

We can see that the baseline can be any function, even a random variable, it just has to be independent from a. As long as it is, expectation of it's contribution to the gradient is zero, therefore the equation remains consistent. What it changes is the variance of the gradients. It makes even good intuitive sense because the figure $q_\pi(S_t,a) - b(S_t)$ describes the *advantage* of performing action a against the *baseline*.

From this follows a natural choice for the baseline function; the value function $v_w(S_t)$, where w is the parameter vector that describes the estimator of the true value function as expected return from state S_t, the choice of the estimator will be discussed in the next subsection.

What is interesting is that as the policy $\pi_\theta(a|s)$ changes according to the gradient updates, the true underlying value function $v_{\pi_\theta}(s)$ changes as well, so it is called for to try to adapt w continuously in order to keep the approximation as close as possible. Therefore we expand the original REINFORCE algorithm by the w parameter update, as can be seen in Algorithm 5.

Algorithm 5 REINFORCE with baseline [9]

Input: $\pi(a|s,\theta)$ policy differentiable w.r.t. θ, $v(s,w)$ value function approximation differentiable w.r.t. w, learning rates α,β

function LEARN(θ)
  generate episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ using $\pi(\cdot|\cdot,\theta)$
  for $i \leftarrow 1$ to $T-1$ **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\delta \leftarrow G - v(S_t, w)$
    $\theta \leftarrow \theta + \alpha\delta\nabla \ln \pi(A_t|S_t,\theta)$
    $w \leftarrow w + \beta\delta\nabla v(S_t, w)$

1.3.3 Actor-Critic Methods

Actor-critic methods are an extension of the REINFORCE with baseline algorithm, where the value function is used also as a *critic*. That means that it is used to provide a biased estimate of the expected return for some future state. For a given horizon, instead of completing the whole episode we just finish steps until the horizon (e.g. 10 steps) and estimate the rest of the episode using our learned value function. The important part is that no gradient is passed through the estimate. That's why the term *critic* is used, as it judges the current state and its predicted return is used to adjust the policy.

1.3.3.1 A3C

In a work by Mnih et al. [15] a modification of the REINFORCE with baseline algorithm was introduced, Asynchronous Advantage Actor Critic (A3C). It is devised as an algorithm suited for multi-core systems thanks to its asynchronous nature. It is meant to be used for learning deep neural policies but can be used for any differentiable approximator. However, in this work we are preoccupied primarily with deep neural networks, therefore, we will consider them only.

The main concept is that each thread runs a single instance of the MDP environment and also one local instance of the model. After the episode is over, the eligibility trace is used to accumulate parameter updates (according to the gradients from the local model) and then the *global* model is updated. At the start of each episode, each thread synchronizes its local model with

the global model. It is assumed that it is not a problem to update the global parameters based on a local model because the local models are synchronized very frequently, therefore the gradients are still mostly correct and it doesn't affect the convergence.

This algorithm is fit for deployment in large data-centers, as it takes advantage of the server grade multi-core CPUs and also naturally allows for distributed training across multiple nodes.

Algorithm 6 Asynchronous Advantage Actor-Critic (A3C) [15]

Input: $\pi(a|s,\theta)$ policy differentiable w.r.t. θ, $v(s,w)$ value function approximation differentiable w.r.t. w, learning rate α, global step counter T

initialize thread step counter $t \leftarrow 1$
repeat
  reset gradients: $d\theta \leftarrow 0$ and $dw \leftarrow 0$.
  synchronize local and global parameters $\theta' = \theta, w' = w$
  $t_{start} = t$
  get state s_t
  repeat
    perform a_t according to policy $\pi(a_t|s_t,\theta')$
    receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t or $t_{start} == t_{max}$
  if terminal s_t **then**
    $G = 0$
  else
    $G = v(s_t, w)$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $G \leftarrow r_i + \gamma G$
    $d\theta \leftarrow d\theta + \nabla_{\theta'} \ln \pi(a_i|s_i,\theta')(G - v(s_t,w'))$
    $d\theta \leftarrow d\theta + \partial(G - v(s_t,w')^2/\partial w'$
  perform asynchronous update of θ, w using $d\theta, dw$
until $T > T_{max}$

1.3.4 A2C

A2C is a synchronous version of the A3C algorithm. In its simplest form it can be thought of as A3C on a single thread [16]. More efficient version runs multiple agents but aggregates the updates into batches [17]. In this case there is no need to maintain multiple sets of parameters, we only need one because the agents are synchronous. It means that the gradients are always correct, the

policy that generated the episode is the policy that is being updated (unlike in A3C).

A natural question arises, whether we should expect one version of the algorithm outperform the other. One could argue that the asynchronous updates of A3C introduce some noise that could potentially serve as a regularization. However, in practice, the opposite trend has been observed. The synchronous version performs slightly better [18]. It is probably due to the fact that in A2C, the gradients are always correct and valid, in turn for A3C, the gradients are almost in all cases a little off. That is because agents finish episodes at different times and only copy the global parameters at the start of each episode, therefore the agent's local parameters used to compute gradients differ from the global parameters that are being updated.

The A2C algorithm has a different set of advantages from A3C. It lets us take advantage of another powerful computation tool, the GPU. Because we maintain only a single set of parameters, we can run the policy and its updates in a single instance all on the GPU. The GPU chip is designed in such a way that it supports massively parallel computation. That is very suitable for deep neural networks but also for our multi-agent concept; we can aggregate not only the batch updates but also the policy evaluation during the episode runs.

Chapter 2

Related Work

2.1 Options

In this work we study hierarchical reinforcement learning methods. A widely studied approach to creating hierarchical agents is the *options* framework [9, 19]. Options are sub-policies that are executed until satisfying a terminating condition, they take observations and output actions as regular policies introduced earlier. The agent then executes the sub-policies as if they were elementary actions.

Most of the research in this area in the past focused on either designing the options by hand or by providing explicit sub-goals and auxiliary rewards to the agent [9, 20, 19]. The agent can be then trained using the common methods for regular reinforcement learning setting, because the options can be treated as elementary actions. A recent work has shown that providing predefined sub-goals to the agent combined with deep learning show promise in a Minecraft environment [21].

However, the difficult question is; how to discover options or even sub-goals automatically. That is a long standing question in the hierarchical reinforcement learning [22, 23, 24]. A recent work presents a method of learning the options' internal policies along with the terminating conditions and also the agent's policy-over-options in an end to and manner, using policy gradient methods [25]. Two contemporary approaches to solving this problem of automatic discovery are presented in detail in the following sections.

2.2 Strategic Attentive Writer

Strategic Attentive Writer is a method published in 2017, by Vezhnevets et al. [8]. The options approach has proven in the past that macro-actions can be very useful abstractions. A macro-action is a relaxation of option, simply a sequence of elementary actions. This naturally raises a question of how to discover macro-actions that are useful in terms of achieving return. The

past approaches relied on hand engineered options, while the modern efforts strive to learn those. A significant obstacle is posed by the fact that it is not clear of how many elementary actions a macro-action should be composed. Vezhnevets et al. propose a new deep recurrent neural network architecture that attempts to solve this problem.

The central idea is that the model is creating (using differentiable attention) and maintaining an explicit multi-step action and commitment-plan. While the action-plan decides what the agent's policy looks like, the commitment-plan decides when does the action-plan gets re-planned. This let's the model discover useful macro-actions of varying lengths, where hypothetically a single macro-action corresponds to a plan segment executed between two re-planning steps. The attention read and write methods used for plan creation are based on [26].

2.2.1 The Model

STRAW is a deep recurrent neural network. It can be broken down to 3 modules. A feature detector that learns useful spatial abstractions, an action-plan module which takes the feature representation from the feature detector and turns it into a plan (during the re-planning steps) and finally a commitment-plan module which produces a plan that decides the re-planning steps.

2.2.1.1 State of the Network

The *action-plan* and the *commitment-plan* are maintained in matrices $A^t \in \mathbb{R}^{(A+1)\times T}$ and $c^t \in (0,1)^{1\times T}$, respectively. A is a number of possible actions and T is a maximum planning horizon (a hyper-parameter). The reason why the action-plan matrix has $A+1$ rows is that the value function estimate is also planned ahead, therefore, simply a row to the matrix is added. The elements $A^t_{a,\tau}$ for $a = 1, 2, 3, \ldots, A$ are proportional to the probability of selecting the action a at time $t+\tau$ (in fact, the policy of the model is softmax of the first column, $\pi_t = SoftMax(A^t_{1:A,1})$). Once again, the elements $A^t_{A+1,\tau}$ represent the value function estimate at time $t+\tau$. The commitment-plan is used to trigger the re-planning steps as follows, q_t is a binary random variable sampled from Bernoulli distribution $q_t \sim c_1^{t-1}$ ($Pr(q_t = 1) = c_1^{t-1}$). In re-planning steps $q_t = 1$ and in commitment steps $q_t = 0$. take note that the values in the commitment-plan are always positive and less than one, therefore valid probability values (note: in the original paper [8], the random variable is denoted by g_t, here we use a different letter to avoid collision and confusion, as the letter g is also present in the second paper studied by this work where it refers to goals).

With this in mind, we can define a macro-action as a sequence of actions $a_{t_1}, a_{t_1+1}, a_{t_1+2}, \ldots, a_{t_2-1}$, where $q_{t_1} = q_{t_2} = 1$ and $q_{t'} = 0$ for $t_1 < t' < t_2$. Naturally, the plans are not stationary, they are rolled in each time-step using

the roll operator $\rho : \mathbb{R}^{m \times n} \to \mathbb{R}^{m \times n}$, it takes a matrix and shifts it to the left, dropping the first column and padding with zeros on the right. We can illustrate it as $\rho(M) = \begin{bmatrix} M_{\bullet,2:n} & 0 \end{bmatrix}$, where $0 \in \mathbb{R}^m$ is a column vector of zeros that is concatenated behind the shifted matrix to retain its original dimensions. This operator is applied both to the action-plan and the commitment-plan. When $q_t = 1$, re-planning occurs.

2.2.1.2 Attentive Planning

We have made clear that the network re-plans only in some of the steps. An important part of the design is that there are no typical recurrent units in the network, the only temporal dependence is through the matrices A^t and c^t. Therefore, when $q_t = 1$ the network has only so much available information. It reads out a part of the action-plan (using attention filters that are described below) and it has the current input observation available. Therefore, we have to assume that the current observation contains sufficient information to generate a useful plan for the several next steps. Because the network also has control over the re-planning horizon, it is safe to assume that. If it doesn't hold, the network inevitably receives punishment and has the opportunity to adjust and identify that the commitment horizons should shorten. It can even re-plan right in the next step, emulating the behavior of a typical feedforward or recurrent network.

The attention is realized using arrays of Gaussian filters. It is used in the form of two basic functions, *read* and *write*. The *read* operation takes an input matrix (in this case the action-plan A^t) and parameters for the Gaussian array, and returns a patch that correspond to readout from the matrix using the array. Analogously works the *write* function, it also takes a matrix and the parameters but also a *write patch* and this time it uses the parameters to write the patch into the matrix and return it. The inspiration for this approach comes from [26], where the authors use 2D Gaussian arrays in order to iteratively generate images. Here, the arrays are defined only over the temporal dimension (over columns in A^t, c^t), therefore could be regarded as 1D. However, we apply the same operation to all the rows alike, so the whole matrix is involved.

Let K be the number of filters in the array for the read, write operations on the matrix A^t, it can be seen as the temporal resolution of the patch. The filter array is defined by three parameters, the array mean μ, the filter's variance σ^2 and the filter's stride δ (the filters are spaced uniformly).

The mean of a specific filter i can be expressed as,

$$\mu_i = \mu + (i - K/2 - 0.5)\delta \tag{2.1}$$

The filterbank matrix for the single dimension $F \in \mathbb{R}^{K \times T}$ is,

$$F_{i,a} = \frac{1}{Z_i} \exp\left(- \frac{(a - \mu_i)^2}{2\sigma^2} \right) \tag{2.2}$$

$$Z_i = \sum_{a=1}^{T} \exp\left(- \frac{(a - \mu_i)^2}{2\sigma^2} \right) \tag{2.3}$$

Where i is the index of given filter and a is the temporal index in our matrices. Z is a normalization constant that ensures that $\sum_{a=1}^{T} F_{i,a} = 1$. For matrix $M \in \mathbb{R}^{m \times T}$ and array parameters $\psi = (\mu, \sigma^2, \delta)$, the *read* operation can be then written as,

$$read(M, \psi) = MF^T \tag{2.4}$$

Clearly, the resulting matrix will have shape $m \times K$ which means that we have used the K filters we have to read K values for each row separately. Each filter can therefore be seen as a single pixel in the resulting patch, and each filter is identically applied to each row.

For a write patch $P \in \mathbb{R}^{m \times K}$, the write operation is then,

$$write(M, P, \psi) = M + PF \tag{2.5}$$

The resulting matrix has shape $m \times T$ which is correct if we want to use it to update the matrix M (notice that F is not transposed for write like it is for read). A key point here is that both of the read and write operations are fully differentiable, the parameters are generated using a linear densely connected layer (the details are in the next chapter) and the policy is extracted exclusively on the matrix A^t (which is modified *only* using the write operation). That means, that the write operation has to be differentiable w.r.t. the parameters of the model.

In the model, two sets of parameters are generated, ψ_t^A and ψ_t^c, using the linear embeddings f^ψ and f^c, respectively, that take some vector on input. The former for the read and write on the action-plan and the latter for the write on the commitment-plan. Those parameters are obviously needed only in the re-planning steps (where $q_t = 1$). Take note, that the Gaussian filter array ψ_t^A for the action-plan has K Gaussian filters, while the array for the commitment-plan has only a single filter (we want to define a single point in the future where new plans are produced, the commitment-plan will get re-written so there is no point in using multiple filters).

2.2.1.3 The Plan Updates

The action-plan is updated using the following algorithm.

In Algorithm 7, the function h represents an embedding of two hidden layers (64 units each), f^A is a linear embedding that produces the write patch

Algorithm 7 Strategic Attentive Writer, Plans Update, source:[8],

Input: z_t, A^{t-1}, c^{t-1}
Output: A^t, c^t, a_t, v_t

$q_t \sim c_1^{t-1}$
if $q_t = 1$ **then**
    Compute attention parameters $\psi^A = f^\psi(z_t)$
    Attentively read the current action-plan $\beta_t = read(A^{t-1}, \psi_t^A)$
    Compute intermediate representation $\xi_t = h([\beta_t, z_t])$
    Update $A^t = \rho(A^{t-1}) + q_t \cdot write(f^A(\xi_t), \psi_t^A)$s
    Compute attention parameters $\psi^c = f^c([\psi_t^A, \xi_t])$
    Update $c^t = Sigmoid(\hat{b} + write(e, \psi^c))$
else ▷ $q_t = 0$
    Update $A^t = \rho(A^{t-1})$ ▷ Only roll the plans
    Update $c^t = \rho(c^{t-1})$
Sample an action $a_t \sim SoftMax(A^t_{0:A,0})$
$v_t = A^t_{t+1}$

when re-planning, $\hat{b} \in \mathbb{R}^T$ is a vector filled with a shared learnable bias $b \in \mathbb{R}$ that provides a prior of sorts for the re-planning probability (especially for re-planning in a step earlier than defined by ψ^c). $e \in \mathbb{R}^{1\times 1}$ is a constant (chosen by authors as 40), that represents the write patch for the commitment plan. The reason for using a constant is simple, we want to define a certain sure horizon in which the re-planning will happen for sure. Notice, that we are using the *Sigmoid* function, so providing a sufficiently large constant will result in a value very close to 1 at the mean of the filter (the bias b is usually small) which is equivalent to ensuring re-planning at some future step. Notice that we are sampling q_t from the commitment plan from the previous step. This makes sense, as the element c_1^{t-1} would become c_0^t if we just rolled the plan (it is natural to sample the first value as we do the same thing with the actions and the value function). But it's necessary to realize, that when we are re-planning, the commitment-plan is completely overwritten, therefore, we sample according to what would become the first element in the commitment-plan in this time-step.

An important feature of this model is that in the commitment steps there is no need to execute the policy network. We are not re-planning, therefore, it is not necessary to compute any of the values of the network in those steps. We only need to roll the plans one step forward and that can easily be done without engaging with the model itself. This saves computation. This feature is implemented using the multiplicative gate q_t. Because q_t is a random variable, we wouldn't be able to pass gradient through it, therefore we approximate the gradient as $\nabla q_t \equiv \nabla c_1^{t-1}$.

Also, for stability of the learning, no gradient is passed from the commitment module to the planning module through ψ_t^A and ξ_t

2.2.1.4 Structured Exploration

Finding ways of ensuring consistent and meaningful exploration of the state space during training is one of the key topics of today's reinforcement learning. Aside from the commonly used entropy regularization (It has been shown to be closely related and similar to the ϵ-greedy exploration in Q-learning [27]), STRAW introduces a novel approach to exploration using methods used in Variational Auto-Encoders [28]. Instead of connecting directly the feature vector output from the feature detector (vector z_t) to the planning module, it introduces a noisy communication channel. The vector z_t is fed into a linear embedding that regresses the parameters of a multivariate normal distribution, specifically the means $\mu^t = \mu_1^t, \mu_2^t, \ldots, \mu_n^t$ and a single σ^t. The distribution is then given as $\mathcal{N}(\mu^t, \sigma^t I_n)$. n is the exploration dimension, take note, that we regress individual mean μ_i^t for each dimension, but the σ^t is shared.

In each time-step t and for the feature vector $\hat{z}_t$ produced by the feature detector (when not using structured exploration $z_t = \hat{z}_t$), we regress the parameters μ^t, σ^t and sample the actual feature vector as $z_t \sim \mathcal{N}(\mu, \sigma I_n)$.

The reason, why this produces structured exploration is following, each plan can be interpreted as a single macro-action, between the re-planning steps, the agent carries on with the plan it already has. Therefore, if we randomize slightly the feature vector z_t, the neural network will have a slightly skewed idea of the environment state and it will produce an action-plan (a macro-action) that does not completely correspond to the actual observation but is randomized. However, the plan produced from the noisy information is going to be carried out for several steps, therefore, the agent is forced to experience the outcome of this semi-exploratory step. We are using the macro-actions to meaningfully explore the environment, therefore we can call this structured exploration.

The vanilla exploration schemes (such as ϵ-greedy and entropy regularization) do not usually have this property, as the the agent can make a correcting step in the next step.

The parameters of the Gaussian are shaped using KL divergence. The prior distribution was selected as $\mathcal{N}(0, 1)$, obviously with the same number of dimensions as n. The KL divergence of two normal distributions is,

$$D_{KL}(\mathcal{N}_0||\mathcal{N}_1) = \frac{1}{2}\left(tr(\Sigma_1^{-1}\Sigma_0) + (\mu_1 - \mu_0)^T\Sigma_1^{-1} - n + ln\Big(\frac{\det(\Sigma_1)}{\det(\Sigma_0)}\Big)\right) \quad (2.6)$$

In our case $\mathcal{N}_1 = \mathcal{N}(0, 1)$, and $\Sigma_0 = \sigma^t I$. That simplifies the general equation

as follows (we omit the index t from μ^t, σ^t for simplicity),

$$D_{KL}(\mathcal{N}(\mu,\sigma)||\mathcal{N}(0,1)) = \frac{1}{2}\left(n\sigma^2 + (-\mu)^T(-\mu) - \ln(\sigma^{2n})\right) \tag{2.7}$$

$$= \frac{1}{2}\left(n\sigma^2 - n\ln\sigma^2 + \sum_{i=1}^{n}\mu_i\right) \tag{2.8}$$

In order to pass gradient through the noise embedding, we utilize so called re-parametrization trick where instead of sampling our distribution directly, we sample a proxy distribution and use it in such a way that our output is identical to if we sampled $\mathcal{N}(\mu^t, \sigma^t)$. Let $e \in \mathbb{R}^n$ be a vector with values sampled from $e \sim \mathcal{N}(0,1)$. Then,

$$z_t = \mu^t + e \cdot \sigma^t \tag{2.9}$$

is equivalent to,

$$z_t \sim \mathcal{N}(\mu^t, \sigma) \tag{2.10}$$

But with the difference that 2.9 is differentiable w.r.t. θ.

For the rest of this work we will call the noise embedding a noise layer.

2.3 Training

In order to train the whole model, we have to construct a loss function that will represent the objective we are trying to achieve. The authors have designed the loss function to be as follows,

$$J_t = J^{out}(A^t) + 1_{g_t} \cdot \alpha KL\big(\mathcal{N}(\mu^t,\sigma^t)\,||\,\mathcal{N}(0,1)\big) + \lambda c_1^{t-1} \tag{2.11}$$

$$J^{out} = \ln\pi(a_t|x_t;\theta)\tilde{A}_t + \frac{1}{2}\tilde{A}_t^2 + \beta\pi(a_t|x_t;\theta)\ln\pi(a_t|x_t)) \tag{2.12}$$

$$\tilde{A}_t = G_t - v(x_t;\theta) \tag{2.13}$$

Where $\tilde{A}_t$ is the advantage function, J^{out} is the standard loss function for advantage actor critic, α is the coefficient of the KL divergence loss, and λc_1^{t-1} is a term that penalizes re-planning (if this term wasn't present, the model would have zero incentive to produce longer plans, it would converge to re-planning in every step which would be counterproductive) with the λ as a coefficient.

The training is done using the A3C [15] algorithm.

2.4 FeUdal Networks

The central model studied in this work is called FeUdal Networks (FuN) [2]. It is a hierarchical deep reinforcement learning model.

2.4.1 Feudal Reinforcement Learning

Feudal Reinforcement Learning is a work of Dayan and Hinton, 1993 [20], that is the main inspiration for FeUdal Networks. The driving idea is that it is meaningful to use hierarchy in policy in order to solve most of the problems that we are interested in solving with reinforcement learning. Dayan and Hinton introduced a hierarchy of managers, where each manager has a single super-manager and several sub-managers. Only the managers at the lowest level act directly upon the environment. All the managers above select actions too, but they are appointed to their sub-managers in order to be carried out. Thus each such manager has two points of decision; which action to carry out and which sub-manager will be appointed.

The core philosophy of the paper is *feudal control*. Each manager has absolute control over all the sub-managers and can punish them or reward them as it seems fit. Also, each manager is accountable to its own super-manager. These principles are promoted by *reward hiding* and *information hiding*. In reward hiding, each sub-manager receives a reward for achieving goals set by its manager despite of the goals of the manger. So if sub-manager achieves its sub-goal but it doesn't help the manager in achieving the goal set by its super-manager, it still receives reward. On the other hand, if a sub-manager fails to reach its sub-goal but it helps the manager satisfy its goal, it still receives negative reward. In information hiding, each manager observes the environment only at the granularity scale at which it operates.

In practice they demonstrated the above principles in a simple 2D maze environment. The hierarchical division is created by repeated division to four parts. There is one highest-level manager that has 4 sub-managers, each managing one quarter of the maze. Each of those managers has 4 sub-managers, each minding its own sixteenth of the maze and so on, and so forth.

In the model studied here (FuN), the hierarchy is between a manager and a worker, where manager sets goals for the worker and the worker is in turn rewarded if it's able to fulfill the goals. The manager evolves an internal representation of the environment and the goals take shape of directions in the *space* of the internal representations (a latent state space). To maintain the separation between manager and worker, the gradient doesn't flow from the worker to the manager directly but rather the manager is trained using an extension of the policy gradient theorem where its goals are treated as actions. Furthermore, the worker maintains several sub-policies and when it receives a goal from manager, it turns it into weights for the sub-policies. Those are then combined and the resulting policy is fed through a SoftMax layer.

2.4.2 The Model

The neural network comprises of three main parts: feature detector, worker and manager.

Feature detector receives the observation from the environment x_t and turns it into a feature representation vector z_t. In our case it is always going to be a 2-layer convolutional neural network because it is the suitable choice for all of the tested environments.

The worker uses a recurrent neural network (RNN), namely Long Short-Term Memory (LSTM) [1], to transform z_t into a policy matrix $U_t \in \mathbb{R}^{|a| \times k}$ that represents k different sub-policies. It also receives a pool of recent goal vectors g_t from the manager. It then transforms the goal vectors using a linear embedding into a weight vector $w_t \in \mathbb{R}^{k \times 1}$ which assigns weight to each policy in U_t. Naturally, the worker's policy is then proportional to $U_t w_t$.

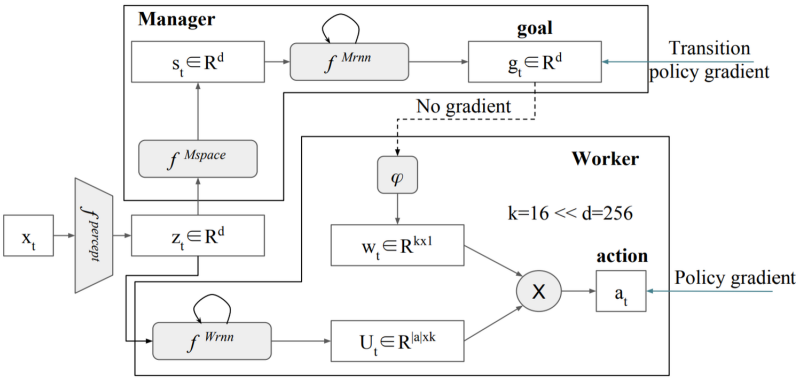


Figure 2.1: FeUdal Networks scheme, source: [2].

The manager first takes z_t and through a non-linearity layer produces manager's internal feature vector, s_t^M it is then passed through a dilated LSTM (dLSTM) layer. dLSTM is a new design of LSTM that lowers the time granularity at which the manager operates. This lets the manager focus more on the big picture and improves long term credit assignment. The details of the dLSTM architecture are explained below. However, it produces a normalized goal vector g_t, that is then pooled with other goal vectors. A key design decision is that no gradient flows through any of the goal vectors. The goals are assigned semantic value as *directions in the manager's latent state space*. This leads us to an important constant for the model, c, it defines the time horizon for the direction of a given goal. So goal g_t is then considered fulfilled if $g_t = s_{t+c}^M - s_t^M$, the measure of goal fulfillment was selected as $d_{cos}(g_t, s_{t+c}^M - s_t^M)$ (which scores the similarity of the two vectors in a natural way), where $d_{cos}(\alpha, \beta) = \alpha^T \beta / (||\alpha|| \, ||\beta||)$. The worker is rewarded or punished in every step based on its ability to fulfill the goals of the manager. Keep in mind that the latent state space representation evolves through time as well,

that's why it's crucial to sever the gradient flow from worker to manager because otherwise we would be only left with a homogeneous model where no meaning could be assigned. The manager is then trained using a modified version of the policy gradient. It assumes the goals as actions.

The whole scheme can be described by following formulas (sourced from [2]):

$$z_t = f^{percept}(x_t) \tag{2.14}$$
$$s_t^M = f^{Mspace}(z_t) \tag{2.15}$$
$$h_t^M, \hat{g}_t = f^{Mrnn}(s_t^M, h_{t-1}^M) \tag{2.16}$$
$$g_t = \hat{g}_t / ||\hat{g}_t|| \tag{2.17}$$
$$w_t = \varphi(\sum_{i=t-c}^{t} g_i) \tag{2.18}$$
$$h_t^W, U_t = f^{Mrnn}(z_t, h_{t-1}^W) \tag{2.19}$$
$$\pi_t = SoftMax(U_t w_t) \tag{2.20}$$

Where h_t^M, h_t^W are the hidden states of the dLSTM and LSTM layers respectively. f^{Mrnn} represents the dLSTM layer, f^{Wrnn} represents the worker's LSTM layer. φ is the learned linear embedding. $f^{percept}$ is the feature detector and f^{Mspace} is the transformation into the manager's latent state space. The function names correspond to the Figure2.1.

2.4.2.1 Goal Embedding

The worker produces as its output a policy matrix U_t that is then multiplicatively turned into a singular policy. The weight vector w_t is produced using linear embedding φ, *without biases*. Because φ has no biases, the embedding can never produce a constant non-zero vector. The consequence of this is that the worker can never fully ignore the manager's output (if the weight vector were always close to zero, no policy could be learned, therefore this situation is highly unlikely). A noteworthy property of the goals is also that they vary smoothly, due to the multi-step pooling

2.5 Training

The setting is similar to our already introduced reinforcement learning framework, the agent receives observations and reward x_t, r_t and makes decisions based on them with the aim to maximize the discounted reward in the current state $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$. The FuN model was trained using the A3C [15] algorithm (as well as all the other models used in this work). The algorithm has to be modified because the manager is trained using a modified version of

the policy gradient theorem (in accordance with the feudal control, manager's goals are treated as its actions). Furthermore, the worker receives intrinsic reward based on its ability to fulfill the manager's goals (also in accordance with the feudal control philosophy), however, unlike in Feudal Reinforcement Learning [2], the worker receives also the regular reward from environment r_t, so this property is weaker than as introduced in the original work.

As we said, the manager is trained using a separate loss function, which is not common. If we allowed the gradient flow from worker to manager through g_t, we could train the model monolithically using the actions from the worker. This, however, the manager's goals would lose any semantic meaning and would just be a part of the network. Instead, the goal of this paper is to train the manager independently, that's why no gradient flows from the worker through g_t. The manager learns to predict directions in its latent state space where it expects favorable returns, as the training goes on, the latent state space is adapted over time to be more useful as a representation and in turn, the manager learns to give better goals to the worker. We can formalize this into a following rule for the manager,

$$\nabla g_t = A_t^M \nabla_\theta d_{cos}(s_{t+c} - s_t, g_t) \tag{2.21}$$

Once again, c is the time granularity parameter of the manager. The intrinsic reward encouraging worker to follow the manager's direction is computed as,

$$r_t^I = 1/c \sum_{i=1}^{c} d_{cos}(s_t - s_{t-i}, g_{t-i}) \tag{2.22}$$

We can see that the intrinsic reward is simply a sum over the horizon of the manger, looking at the c previous goals, computing how well is the worker fulfilling the goals that are presently *active*.

An advantage of the *directional* nature of the goals lies in the fact that the worker is not required to understand the absolute nature of the manager's latent state space, it only needs to be able to navigate it in terms of directions, which is supposedly easier to understand to the worker. Another strong point is that if we use directional goals, the same directional goal may be useful in many different locations of the state space, e.g. a goal with semantic meaning *evade enemy* is useful in any position of the environment where the enemy is encountered. Therefore it helps the generality of the model and its components.

For the worker, the gradient is similar to what we have seen so far,

$$\nabla \pi_t = A_t^W \nabla_\theta \ln \pi(a_t | x_t; \theta) \tag{2.23}$$

2.5.1 Loss Function

The resulting loss function for the worker can be formalized as follows,

$$J_t^W = A_t^W \ln \pi(a_t|x_t;\theta) + \frac{1}{2}(A_t^W)^2 + \beta\pi(a_t|x_t;\theta)\ln\pi(a_t|x_t;\theta) \tag{2.24}$$

$$A_t^W = G_t + \alpha G_t^I - v^W(x_t;\theta) \tag{2.25}$$

$$G_t^I = \sum_{k=0}^{\infty} \gamma_W^k r_{t+k+1}^I \tag{2.26}$$

$$G_t = \sum_{k=0}^{\infty} \gamma_M^k r_{t+k+1}^I \tag{2.27}$$

Where G_t^I is the intrinsic return and α is the coefficient that decides how much is the worker focused on maximizing the internal reward as opposed to the regular environmental reward.

For the manager, the loss function is,

$$J_t^M = A_t^M d_{cos}(s_{t+c} - s_t, g_t) + \frac{1}{2}(A_t^M)^2 \tag{2.28}$$

$$A_t^M = G_t - v^M(x_t;\theta) \tag{2.29}$$

2.5.2 Architecture

The feature detector is dependent upon the environment, f^{Mspace} is 256 units big hidden layer with ReLU nonlinearity. f^{Wrnn} is a 256 units LSTM recurrent layer and f^{Mrnn} is a 256 units dLSTM layer, $k = 16$ and $c = 10$.

2.5.2.1 dLSTM

Another contribution of the FuN paper is a new LSTM architecture. One that partially alleviates the problem of long term credit assignment by pooling states over several time-steps. This essentially results in the manager having a different time resolution than the worker, therefore enabling it to spot more long term dependencies than with just regular LSTM. The idea of dilation is based on dilated CNN architecture [29].

The dLSTM layer is in most ways similar to LSTM layer with one exception, it maintains a pool of hidden states and outputs and updates them one by one maintaining the others. The parameter that decides the extent of pooling is called *radius* and is denoted by r. The whole state of the layer then consists of a set of hidden states $h_t = \{\hat{h}_t^i\}_{i=1}^r$ (note that in h we include both the LSTM hidden vectors h and c, as described in the previous sections, using a single letter simplifies the notation).

At time t, the update rule is as follows,

$$\hat{h}_t^{t\%r}, g_t = LSTM(s_t, \hat{h}_{t-1}^{t\%r}; \theta^{LSTM}) \tag{2.30}$$

Where % denotes the modulo operator. At each time-step one hidden state is updated. Notice that this way, every state is updated only once every r time-steps, this means the LSTM essentially operates on a different time scale to the regular LSTM. Take note that the parameters θ^{LSTM} are invariant of $t\%r$, the parameters are shared for all of the pooled values.

Chapter 3

Used Methods

3.1 A2C

In a previous chapter, the A2C algorithm has been introduced. It is a synchronous version of the policy gradient method A3C [15], suitable for evaluation on a single machine with multi-core CPU and a single GPU. Here, we discuss the implementation details and challenges.

Due to their good affordability and the lowering cost of computation in general, GPUs have become prevalent in deep learning. Neural network algorithms are implemented in such a way that they take advantage of the massively parallel architecture which leads to a significant speedup compared to even optimized CPU implementations. That is also what we use in this work, particularly a general computation framework called TensorFlow (TF).

All the methods studied in this work were trained using the A2C algorithm, so the implementation developed here is general.

If we have a recurrent model, probably the easiest and most efficient way to handle backpropagation through time and also the *monte-carlo* evaluations of the policy, is to maintain two separate TensorFlow graphs that share parameters. For a recurrent model that can be unrolled, if we consider the recurrent connections simply as additional inputs, only feed-forward graph remains, let us call it a single *graph layer*. In the *monte-carlo* evaluation, only a single graph layer is sufficient for computation. We do not do backpropagation, therefore, we can maintain all the previous hidden states in a variable on RAM and feed it to the graph in every step. This is an efficient way to do it, as our graph is relatively small. For the training phase, we need an unrolled graph with more than one graph layer. The number of graph layers determines how many steps into the past is the backpropagation through time going to run. Therefore, if we want to do backpropagation with 20 steps into the past, we would construct a graph with 20 graph layers. A convenient way to do this is to prepare a function for each model, that simply takes as an input all the input tensors and hidden variables from previous step, and outputs all the output

tensors and all the hidden variables for the next step. This way, constructing a graph with arbitrary many graph layers is quite straightforward. However, one still has to be cautious because writing the code is very prone to errors, such as off-by-one errors and invalid handling of boundary conditions for initial hidden states, first and last graph layers. The loss function is then built upon the outputs of the last graph layer, which represents the current time-step. When training, the gradient is propagated through the unrolled graph and because all the parameters are shared, the gradient accumulates at each parameter.

3.1.1 RMSProp

The optimizer used for training in all the experiments in this work was *RMSProp* [30] which is a first order method with first order momentum that is widely used because it strikes a good balance between speed of convergence, computation demands and memory demands. For each parameter θ_i we maintain a momentum variable $MeanSquare(\theta_i, t)$ at time-step t, with following update rule,

$$MeanSquare(\theta_i, t) \leftarrow 0.95\, MeanSquare(\theta_i, t-1) + 0.05 \frac{\partial J}{\partial \theta_i} \tag{3.1}$$

The learning rate is then modified using an inverse square root of the $MeanSquare$ value as follows,

$$\theta_i \leftarrow \theta_i - \frac{\alpha}{\sqrt{MeanSquare(\theta_i, t) + \varepsilon}} \tag{3.2}$$

Where α is the selected global learning rate and ε is a small positive constant that serves both to prevent division by zero and also to *bias* the effective learning rate essentially giving it a ceiling, because the denominator can only get so small. A value used here was $\varepsilon = 10^{-5}$. The update rule implies that if the gradients are consistently bigger, the learning rate is going to be reduced whereas when the gradients are getting smaller, the learning rate is increased.

It is fast enough because it is a first order method, it has good convergence in practice because of individual learning rates for each parameter but that also implies the requirement of an additional variable per graph parameter (still, some other methods require two additional variables). However, the additional variables are required only during training, in deployment, only the model parameters are required.

3.1.2 Episode handling

One of the crucial aspects of the A2C implementation is the way episodes and their sizes are handled. A concept that is fixed for all the variants is that we run multiple agents at once. This takes advantage of the GPU architectures,

which run code in 32-thread quanta called *warps*. So if we use 32 agents, we can expect the evaluation time on GPU (for the monte-carlo part, not the training) be a lot shorter than run time for a single agent repeated 32 times.

3.1.2.1 Variable Batch Size

First variant we present is having a fixed number of agents and running each agent from start to the terminal state. Then we take all the eligibility traces created this way (eligibility trace is a history of the agent's way through the environment in a single episode, i.e. a sequence of observations, actions and rewards) and construct a single batch from them. Then we train the network on this batch. Because the episodes vary in length, each batch is going to have a different size. If we assume, that for a given batch size, the gradient has some fixed variance, during the training, the gradient estimates are going to have different variances.

Let us discuss two cases of environments, one in which if the agent doesn't have a proper policy, the episode is likely to end soon (e.g. the CartPole environment, if the agent's control is not good enough, the pole is going to fall very soon and the episode terminates) and the second in which if the agent doesn't have a good policy, the episodes are going to be drawn out until the episode time-step limit and the episode is terminated (e.g. a maze environment, where each episode has a maximum number of steps and if the agent doesn't reach the goal fast enough, it automatically terminates).

In the first type, the batches are at first going to be small and get larger, as the agent's policy gets better, because the episodes are getting longer and longer. In the second type, the batches are large at first and get smaller as the agent learns to reach the goal faster, thus the episodes are getting shorter.

Now, first disadvantage of the described approach is that it behaves differently under different environment types (in terms of the gradient estimate variance). Furthermore, we measure and compare the temporal progress of training using *the number of processed samples*, in other words, the number of steps in the environment all the agents made in all of the episodes, and for this method, the number of batches during training does not linearly correspond to the number of batches. In fact it is not at all obvious, how those two values correspond to each other, as the evolution of the batch size depends on the environment that is being run. But in the end, what we are interested in is the sample efficiency, not the batch efficiency. We want to know, how many iterations of the environment simulation had to be run, in order to train a given model. Another disadvantage is that we are updating the parameters with a fixed learning rate while the variance of the estimates varies. This is quite unpredictable as the notion of learning rate is tightly connected to the gradient variance. In effect, when we are using large batch size, the variance of our gradient estimate is going to be smaller than when using a small batch size. This means that when we arrive at some estimate of the gradient, our

confidence in it being close to the true gradient depends on the batch size and in a way, based on this expected variance, we choose our learning rate (this is not very true in practice, as the learning rate is being often chosen manually, based on what works well, or is sampled from some distribution with limited range of values and the experiments are repeated with different learning rates). Therefore, it is not really possible to meaningfully reason about the choice of our learning rate, independently of the environment and the model that we use. That's why this method is not really optimal.

3.1.2.2 Variable Batch Size with Learning Rate Scaling

A naive way of trying the solve the problem of connection between batch size and learning rate, while still executing full episodes with each agent is to change the learning rate based on the batch size. This would reflect our belief about the connection of selecting appropriate learning rate for given gradient estimate variance. The most simple way would be to reason as follows, we select a batch size and a learning rate that we expect to perform well if we had batches of given fixed size, then if we receive a batch that has size differing from our prior batch size, we compute the ratio of the real and prior batch size and then scale the learning rate accordingly. Let b_p be the prior batch size, b be the batch size of the actual batch and α the learning rate. Then the update rule can be expressed as,

$$\theta \leftarrow \theta - \frac{b}{b_p}\alpha\nabla_\theta J \tag{3.3}$$

This makes sense in theory but might be troublesome as the relationship between the gradient variance based on batch size and the optimal learning rate is probably not linear and also depends on the task that is currently being learned. Varying the learning rate too much might destabilize the process of training. In order to dampen this effect, we propose raising the coefficient to the scaling power $0 < s < 1$ which will bring the coefficient values *closer* to 1.

$$\theta \leftarrow \theta - \left(\frac{b}{b_p}\right)^s \alpha\nabla_\theta J \tag{3.4}$$

However, this method still suffers from all the other problems mentioned in the previous method. The number of processed samples still doesn't really correspond to the number of processed batches, or in other terms, the number of performed parameter updates.

One more problem of both mentioned methods is in computational efficiency. When we start a new episode for all the agents, let's say we have 32 of them, we are fully using the parallelism of the GPU, however, as the episodes progress, some of the agents begin terminating, therefore we send to the GPU more and more requests with fewer agents. That lowers the overall efficiency of the training.

3.1.2.3 Fixed Batch Size

The solution to all the above problems is to fix the batch size. Let's say we run 16 agents and we set the batch size to 320. That means that each agent will contribute to the batch with exactly 20 samples, which corresponds to 20 steps in the environment. So in each time-step, we always evaluate the action of all the agents, therefore utilizing the full bandwidth of our GPU model. Though, we have the additional implementation complexity of handling the ends of episodes and providing correct hidden states on the input.

In the previous two cases, the model always started from its initial state (we are talking about the initial hidden state of a recurrent architecture) and ended in a terminal state. In this case it gets more complicated, because the batch can be full mid-episode (in fact it is very likely that most of the agents will be somewhere in the middle of their episode after taking the 20 steps), therefore, when evaluating the agents for the next batch, the last hidden state from the previous batch has to be saved and used as an input to the next batch. Second complication arises from the fact that agents are likely to end their episodes mid-batch, e.g. after only 10 steps. In the previous cases, we would have terminated the agent and continued with the rest but here we want to have a constant batch size. What we do is simply save the information that the episode was terminated, compute the expected return estimate from our value function when the final state is not terminal and then initialize all the hidden states of the model for the next episode.

Even though all the mentioned methods are valid (they are all using correct gradient estimation and all are *on-policy*), some of them are more fit for application than others. Namely, the last introduced method is the best one to use as it has consistent batch size and thus removes unpredictable factors. Also, one might think that partitioning the episodes into fixed chunks may reduce the speed of convergence (as we are using recurrent architectures and cutting the episode in half means that the gradient cannot be propagated through the first part of the episode that was performed using the previous policy).

In the evaluation chapter, we will present results of experiments with the three approaches introduced here and compare their convergence on a standard benchmark, the CartPole environment.

3.1.2.4 The On-Policy Pitfall

A notion that was not entirely clear when the author was implementing the A2C algorithm for the first time was the concept of *on-policy* and *off-policy* algorithms. The first implementation was running all the agents at the same time, in that it was similar to the fixed batch size approach but it collected the agent's samples for training only once their episode was terminated. The agent immediately continued with next episode, therefore all the agents ran

at the same time, saving computation. Once enough samples were collected (understand more than the batch size) a batch was constructed with constant size, and the extra samples were offloaded to another batch that was ready for the next round of training. There are 2 problems that make this approach fundamentally incorrect.

First of all, once the batch was filled, the agents that contributed to it early were already in the middle of their next episode which means that once the batch was trained, the agents all of a sudden ran a different policy than in the first part of the episode. This means that the eligibility trace that would be generated for the next batch, was started with the previous policy and at some point continued with the current one. That means that the gradients computed from such eligibility trace are mathematically invalid as the trace was generated using a different policy than the one that we are trying to optimize. This approach was modified into the first method that doesn't fix the batch size and always waits for all the agents to finish the episode to train and then start another one.

Second of all, the samples that overflew the first batch remained in the batch for the next policy but the same logic follows. It was the previous policy that generated these samples, therefore it is incorrect to try to estimate the gradient of the current using the samples of the previous policy.

This boils down to a concept of *on-policy* and *off-policy* algorithms. The former refers to algorithms that are only able to optimize a policy using eligibility traces generated by it (this is the case for A3C and A2C). The latter refers to algorithms that are able to update their policy also using experience from other policies (e.g. Q-learning).

Even though one would expect, that using only a one-train-batch-old traces would have only a small impact and that the model would learn the task in the end, even if more slowly, the reality was different. The incorrect model failed to converge on any of the slightly non-trivial tasks. That is a useful lesson to remember when implementing the reinforcement learning algorithms. Even though some of the algorithms can be realized in multiple possible ways, it is crucial to be sure that the algorithm is still mathematically valid. This mistake was not obvious to the author at all, as his experience with implementing reinforcement learning algorithms was small prior to this work and it took almost 2 months to realize this and fix it.

3.2 Strategic Attentive Writer

One of the goals of this work was successfully re-implementing the Strategic Attentive Writer [8] method. Here we discuss some specific details.

As mentioned in the first chapter, the model creates multi-step plans using arrays of Gaussian filters. The array has three parameters, μ, σ^2 and δ, which correspond to the mean of the array, the variance of each filter and the stride.

The filters are spaced equally, according to δ, therefore, μ is the only locating parameter we need.

3.2.1 Action-Plan

The parameters for the action-plan array, which uses 10 Gaussian filters, are regressed from a linear embedding and are assumed to be generated in the form of $(\tilde{\mu}, \ln \sigma^2, \ln \delta)$, value of μ is then obtained as,

$$\mu = \frac{K}{2}(\tilde{\mu} + 1) \tag{3.5}$$

The reason, why the parameters are not regressed directly lies in the fact, that we want to have a meaningful initial functionality (also, σ and δ have to be positive). Because the values are regressed from values of a linear layer, due to zero-symmetric initialization (the symmetry is present in virtually all used initialization methods), the expected value of each output initially is zero. Therefore, the initial expectation for values δ and σ is 1, which results in a well-behaved configuration. Furthermore, if μ was used directly, mass of the half of the filters would lie outside of the action-plan (left from the zeroth element). Therefore, we compute it in such a way that its initial expectation is half the filter count (K is the number of Gaussian filters in an array). This results in a desirable initial behavior because we are writing into the first 10 steps of the plan with each filter approximately covering one step (because of unit stride and unit variance).

3.2.2 Commitment-Plan

Because the commitment-plan uses only a single Gaussian filter to write (we want to specify a single horizon for re-planning since the commitment-plan gets re-written completely during the re-planning steps, unlike the action-plan), we need only two parameters, μ and σ^2. The parameters are once again regressed from a linear layer in the form $(\tilde{\mu}, \ln \sigma^2)$. What differs is the way we compute the mean,

$$\mu = 20 \cdot Sigmoid(\tilde{\mu}) \tag{3.6}$$

There are multiple reasons for this difference. Firstly, it is absolutely crucial that the μ is positive because otherwise the single Gaussian filter lies outside of the commitment plan and even though the values are normalized, it still results in a very long re-planning horizon (potentially forever). Secondly, giving the re-planning horizon an upper bound serves the method well too, because we avoid unreasonably long plans (it is unlikely that the model would learn to utilize meaningfully such long plans). Another important reason is once again the initial expectation. The initial expected re-planning horizon is 10 steps which fits really well with the above initial conditions for the action

plan (which initially will be 10 steps long). What those three reasons have in common is that they greatly increase the stability of the model. Author of this work has spent a great deal of time implementing, modifying and debugging the STRAW algorithm with little to no luck. The model was simply too unstable, sometimes it would behave well but most of the time the behavior would be pathological. Luckily, the authors of the STRAW paper were contacted and they provided some crucial details into their original implementation (especially the above expression with the sigmoid) which were not mentioned in their original paper. Without these specific details, the author of this work would fail to successfully implement the STRAW method despite working on it fully for several months.

3.3 FeUdal Networks

Another goal of this work was to take some *state-of-the-art* methods and attempt to improve them. The structured exploration in the STRAW model is a really interesting feature that helps to solve one of the toughest problems in reinforcement learning (at least for the STRAW model).

The central effort of this work was to use similar noise injection to aid the structured exploration in the FeUdal Networks model. In the STRAW model, there are two main modules, the feature detector and the planning module. The noise layer is inserted between them. In the FeUdal Networks, we have three modules, the feature detector, the manager and the worker. In addition to obvious connection flow from the feature detector to the manager and from the feature detector to the worker, there is also a connection between the manager and the worker. This gives us three possible locations to put the noise layer in. Furthermore, we can also apply the noise layer to the feature detector output directly, affecting both the manager and the worker, that results in four options.

In the end, all of those could not be tested (due to *heavy* computational demands of this research area) so the author has selected to properly evaluate the option of injecting the noise layer *before the manager* compared against not using the noise layer at all.

The theoretical rationale why it makes sense to put the noise layer right before the manager is following, let us consider a simple LSTM model with some feature detector and a single hidden LSTM layer and an output layer that produces an action and a value function estimate. The question is, does it make sense, *in terms of aiding meaningful exploration*, to put the noise layer between the feature detector and the hidden layer? The answer in this case would be *no*, because the model has a chance to counteract the noisy information from the last step in the next one (technically, its internal state is influenced by the noise and experiences the results of the noisy information, but there is no structure to this exploration, the agent doesn't utilize any

macro-actions to explore, or if it does internally, it is very weak and cannot be assigned semantically like in the more structured models, STRAW and FuN). Same goes for why the common exploration means are not sufficient (namely ϵ-greedy and entropy regularization), they influence the model only very locally (in terms of exploring new states) and if it makes a random step, it can trace back in the next step and continue in its regular policy. On the other hand, the noise layer makes sense in the STRAW model as the model is re-planning only every couple steps and is forced to experience the outcome of the macro-action produced based on information that is slightly off. So the crucial element is whether the model is forced or not to experience some states in the future that are based on some noisy information from the past. That's where the manager in FeUdal Networks fits in. It uses the novel dLSTM architecture to operate over longer time spans, the dLSTM pools several hidden states and updates only one of them in each step, that way it enables credit assignment over really long time spans. Precisely because the states and goals are pooled over multiple steps, we can use the structured exploration scheme. If we insert the noise layer right before manager input, the manager will receive information about the environment that are slightly off, therefore it will have a slightly skewed internal representation (or in other words, an idea about the world) and based on it it will produce slightly skewed goals. Because those goals are pooled over multiple steps, the worker will attempt to fulfill those goals based on slightly misleading information thus forcing the whole model to experience some novel states. Because the goals are used to regress the weighting w_t of the worker's policies, skewed goals mean that different sub-policies than usual are going to be used. This is why this is in fact a *structured* exploration and not only noise regularization, because the workers sub-policies are used to experience the full outcome of a skewed goal, this corresponds to the STRAW model having to experience its plan that it's currently committed to.

The empirical rationale is that the author has performed few ablative experiments that confirmed, that the manager noise option is indeed the most promising of the four mentioned options and is worth exploring fully.

To train the noise layer we add the KL-divergence term as present in Eqn. 2.12 to the loss function of the agent.

3.3.1 Value Function Estimators

In the original paper, no information is shared about how the authors regressed the estimate of the value functions for the manager and the worker. The author of this paper has taken inspiration in a publicly available implementation of the FeUdal Networks model, where they simply add a hidden ReLU layer after the manager's and worker's last layers and on top a single linear neuron whose output is the value function estimate.

3.4 Learning Environments

A crucial aspect of reinforcement learning is the environment in which given method is tested. There are endless possibilities of environments and many different ones used in research in practice. The ones worth mentioning the most are the Arcade Learning environment [31] which are emulations of many of the Atari 2600 games, DeepMind Control Suite [32] which is a 3D environment, where the agent controls a body with multiple joints, OpenAI Gym [33] which is a suite of many various environments suitable for reinforcement learning and last but not least the DeepMind Lab [34] which is a 3D maze-based environment with different tasks. All the mentioned projects are bundles of many environments usable for reinforcement learning research (in fact, the OpenAI Gym contains implementations of many of the Atari 2600 games). Each environment has its specifics and likely will fit some agents better than others. Take as an example two Atari 2600 games, Breakout and Ms. Pac-Man. Breakout is an environment well fit for a simple reactionary agent and more complex models might have trouble beating simple feed-forward baselines while Ms. Pac-Man requires deeper understanding of the game and requires planning ahead, a task in which a simple agent might fail while ideal for some hierarchical methods (example: [8]). Therefore, it is healthy to have a wide range of environments available, where the agents and algorithms can be tested and evaluated against each other.

The OpenAI Gym [33] has the added benefit that its Python API is very straightforward and unified across all the different environments. This means, that code running a simple reinforcement learning agent in a given environment can be easily reused for a different environment with minimal overhead. Furthermore, it is straightforward to implement new environments in the format they came up with, thus the OpenAI Gym has been used as an interface for the environments created by the author of this work. This works well, because all the environments used in this work that were not implemented by the author are already present in the OpenAI Gym framework. It implements two simple functions, reset and step, that are used to reset the episode and to make a step while returning the current observation, reward and information about whether the encountered state is terminal or not.

3.4.1 GridMaze

One of the aims of this work is to design or select a complex RL environment that is scalable and allows fast evaluation. In the end two custom environments are used in this work. GridMaze environment that was used in the Strategic Attentive Writer [8] and MazeRooms that was designed by the author which will be discussed in the next section

In the STRAW paper, the authors use a simple grid maze to demonstrate some properties of their agent. For this reason it was used as the primary

development and environment for the STRAW agent here.

The maze is given by a grid $m \times n$ where both m and n are *odd*. The reason for this is that some of the cells cannot have walls placed on them. If we consider a grid $G \in \{0,1\}^{m \times n}$, where $G_{i,j} = 0$ means there is a wall on i-th row and in $j - th$ column, while $G_{i,j} = 1$ means free space. The boundary is always made of walls, therefore $G_{\bullet,1} = G_{\bullet,n} = G_{1,\bullet} = G_{m,\bullet} = 0$. Furthermore, $G_{i,j} = 0$, for $i = 2k + 1$ and $j = 2l + 1$, $k, l \in \mathbb{N}$ and $G_{i,j} = 1$ for $i = 2k$ and $j = 2l$, $k, l \in \mathbb{N}$. All the cells not covered with the mentioned conditions may or may not have walls. However, each free cell has to be reachable from any other free cell and also, the maze contains no cycles.

The agent has available 4 actions, to move up, down, left and right. The agent receives a reward of -0.01 for each step to the free cell and -0.02 for an invalid step to the wall. When it reaches the goal, a reward of 0.01 is received. The increased punishment for hitting the walls is a slight *reward shaping* (injecting domain specific knowledge), however, the environment can be scaled in such a way to still pose a significant challenge for the agent.

In each episode for each agent a new maze was generated using the Wilson's algorithm [35] which has the property of producing all the possible mazes that satisfy the above conditions with *equal* probability. The agent observes the environment fully.

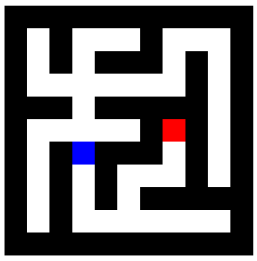


Figure 3.1: An example of a maze in the GridMaze environment, red cell represents the goal and the blue cell represents the agent's location.

The environment is scalable (because the maze size can be adjusted) and can be evaluated rapidly because it is dealing with only small matrices. Also, it is suitable for testing complex agents as the maze always contains only a single shortest path and finding it is not straightforward for an agent without any domain knowledge. Therefore it fits the required criteria.

For the models trained on this environment, the feature detector is a two-layer convolutional neural network, with 3×3 kernels and stride of 1. The first layer contains 16 kernels and the second 32 layers.

3.4.2 MazeRooms

One of the goals of this work was to design a new complex reinforcement learning environment fit for developing and training hierarchical agents, is scalable and allows for computationally fast evaluation. We introduce a novel maze-based environment called MazeRooms. The inspiration for this environment comes from the Atari 2600 game Montezuma's Revenge. In this game a player has to walk through different rooms, collect keys, unlock doors and avoid enemies. It is considered as one of the toughest environments in reinforcement learning and to the author's best knowledge and to this date (May 2018) there has been no model that would be able to solve this environment in a purely general manner without some prior domain-specific knowledge inserted into it. The game is very difficult for agents because one has to understand concept of keys, doors, enemies and their interaction between different rooms.

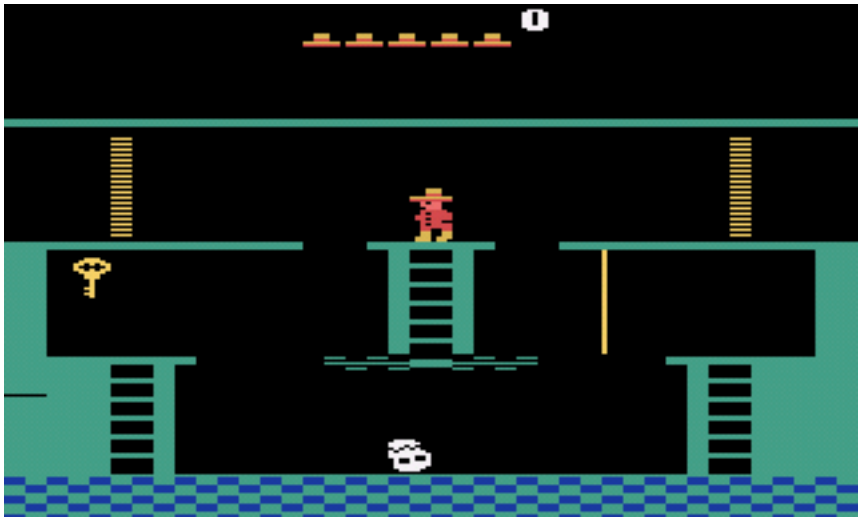


Figure 3.2: The starting screen of Montezuma's Revenge for Atari 2600.

The idea behind the MazeRooms environment is to extract the essence of the complexity of Montezuma's Revenge and create an environment that emulates it in a highly controlled manner. The goal is to have an environment that can be scaled and fine tuned according to the complexity of the model we are testing.

The MazeRooms environment is again a 2D maze environment but this time it is only partially observable. It is defined by a grid of adjacent rooms that have equal dimensions and corridors between each other. The agent then doesn't observe the whole maze but sees only the room it is currently in and a map of the layout of the rooms. The map is a *zoomed out* version of the environment, each cell corresponds to one room and the agent sees which rooms it can walk through, in which room the agent is and also in which room

the goal is. Then it has a second map that is local to the room, so it sees the layout of the room with all the corridors to adjacent rooms and also the goal if it is placed in the same room as the agent. If the agent enters a corridor, it is registered as entering the next room.

The rooms themselves can either be empty, contain a fixed number of randomly generated obstacles or a maze generated by the Wilson's algorithm mentioned above. The obstacles are generated in such a way that no unreachable spots are created. This makes the learning easier as it eliminates the cases when the agent would spawn in a part of the maze separate from the goal. The way it is done is that for each room, a random location is generated and it is tested if all the free cells are reachable from all the other free cells. If it is not, new location is generated. This is repeated for a finite number of times and it is possible that no obstacle is placed.

The agent can move up, down, left and right. The reward scheme is identical to the previous environment. Therefore -0.01 for a valid step, -0.02 for a step to the wall and 0.01 for reaching the goal.

In order to successfully solve the problem, the agent has to learn that the map works on another hierarchical level than the view of the room and it has to be able to integrate information from both levels of abstraction together to produce an optimal policy. Due to this necessity of aggregating information from different levels this is a useful environment for testing hierarchical agents.

Because the size of observations is very small, the environment can be evaluated rapidly and as mentioned. Furthermore, the environment can contain any number of rooms of any size, containing mazes, any number of obstacles or even be empty. Therefore, the environment is scalable according to the complexity of the agent, thus fulfilling the goal of the work.

For the models trained on this environment, the feature detector is also a two-layer convolutional neural network, with 3×3 kernels and stride of 1. The first layer contains 16 kernels and the second 32 layers.

3.4.3 Cart Pole

Another environment used in this work is Cart Pole. It is a simple task of balancing an inverse pendulum on a cart. The environment is 2D, the agent's actions are move left and move right, if the pendulum falls over the episode ends. For each step in which the agent keeps the pendulum from falling over, the agent receives a reward of 1. Each episode is terminated after 200 steps.

The feature detector for this environment is a single densely connected layer with 100 units and with ReLU nonlinearity.

3.4.4 Enduro, Atari 2600

The last environment tested in this work is the game Enduro for Atari 2600. It is an endurance racing game where the player has to try and stay in the race

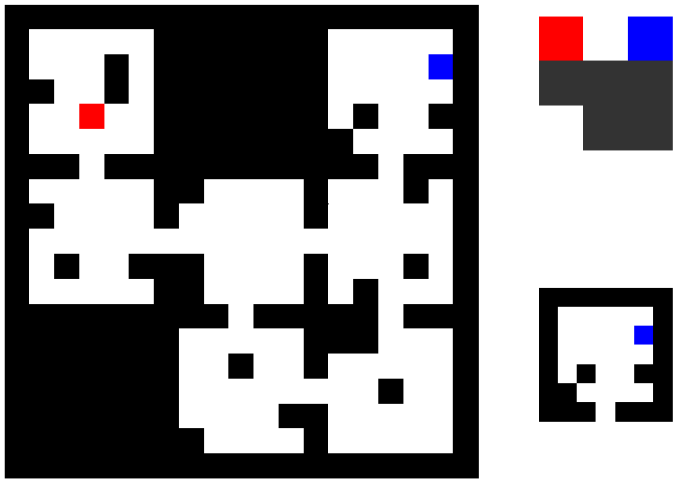


Figure 3.3: The MazeRooms environment with 3 obstacles generated in every room, on the left is the global overview of the whole maze which the agent doesn't see. In the top right corner is the agent's 3×3 map, where the gray cells correspond to existing rooms, white cells to non-existent rooms, red cell is the room with the goal and the blue cell is the one agent's in. Below, is the agent's local view of the room it is in.

Figure 3.4: The CartPole environment.

as long as possible. Time passes in the game and the player has to overtake 200 cars during the first day and 300 cars every other day otherwise the game ends. All the Atari 2600 games have 5 controls, 4 directional actions from the joystick and the action button, therefore the agent has 5 actions available. In the Enduro game the player has to hold the button to accelerate to overtake other cars but if they crash into a car in front of them, the player's car loses speed for short amount of time and has to accelerate again. The player is rewarded for remaining in the race.

The observations from the environment are scaled down to 1/4 of their original size and stacked in groups of 4 frames and each agent's action is repeated 4 times in the following 4 frames. The reward is clipped between -1 and 1. This setup was used in [36] (except they used slightly different scaling due to limiting nature of their implementation).

This environment was selected because it was a part of the experimental evaluation in FeUdal Networks [2] and it was the task that had the largest difference between the performance of FuN and LSTM baseline. Therefore we test our modifications of the FuN on it.

The feature detector for the Enduro environment is a convolutional neural network with 2 layers, first with 16 filters, kernel size 8×8 and stride 4, and the second with 32 filters, kernel size 4×4 and stride 2.

Figure 3.5: The Enduro Atari 2600 environment.

3.5 Statistical Tests

In order to evaluate the results of our experiments we use the Welch's t-test [37]. It is a statistical test that can be used to verify whether two populations

are generated from distributions with identical mean. It is an extension of the common Student's t-test but it is fit for the cases when the two populations have unequal variance and unequal sample size. The advantage of the Student's t-test over confidence intervals lies in the fact, that Student's t-test works even for very small sample sizes, unlike confidence intervals. Due to the nature of our problems and the models we use to solve them, each experiment is highly demanding in terms of time and computation (each experiment took from several hours to multiple days), thus, only a limited sample size could be chosen. Therefore the Welch's t-test is fit for the purpose.

Chapter 4

Experimental Evaluation and Discussion

All the trained models and their configuration files (containing all hyper-parameters) are located on DVD 2 (see Appendix B). The code used to train them is present on DVD 1. All the hyper-parameters used in the following experiments are presented in Appendix A.

4.1 A2C Variants

In order to compare the different A2C implementations, we have devised an experiment comparing all the approaches, namely, fixed batch size, variable batch size with fixed learning rate, variable batch size with learning rate scaling with $s = 1$ and variable batch size with learning rate with $s = 0.5$ (the parameter s was introduced in Eqn. 3.4). For each variant we have executed 5 runs with identical hyper-parameters (aside from the ones tested).

The hypothesis that is being tested is that the fixed batch size approach will converge to better solutions.

The results displayed in Figure4.1 seem to support our hypothesis, as the fixed batch size version has a significant margin over all the other methods in the last 10 epochs. However, to further show that our hypothesis seems correct, the different experiments were compared using the Welch's t-test [37]. We compare the fixed batch size version against all the other versions to see, whether with statistically significant probability, the two distributions underlying the populations have different means. Each time-step is treated as a separate population for the statistical test, each population has sample size of 5 (we ran 5 experiments). Even though the populations in different time-steps are clearly not uncorrelated, assuming that only makes the statistical test stronger.

From Figures 4.2, 4.3, 4.4 it is apparent that there indeed is an indication

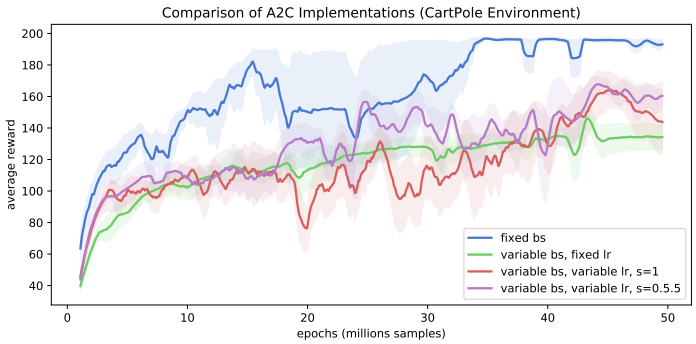


Figure 4.1: Results of the 20 experiments in total, displayed is the mean and sample standard deviation bands of the 5

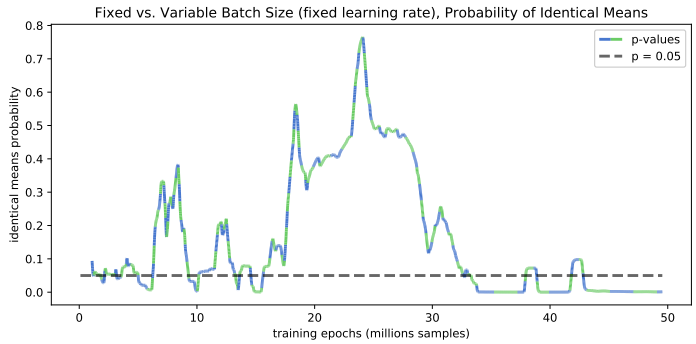


Figure 4.2: Welch's t-test results (fixed vs. variable batch size) against the $p = 0.05$ line.

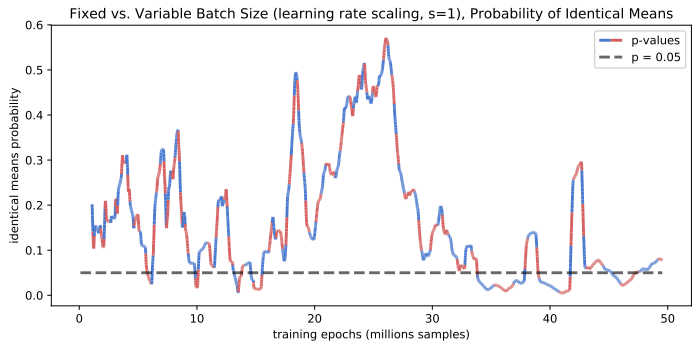


Figure 4.3: Welch's t-test results (fixed vs. variable batch size with learning rate scaling, $s = 1$) against the $p = 0.05$ line.

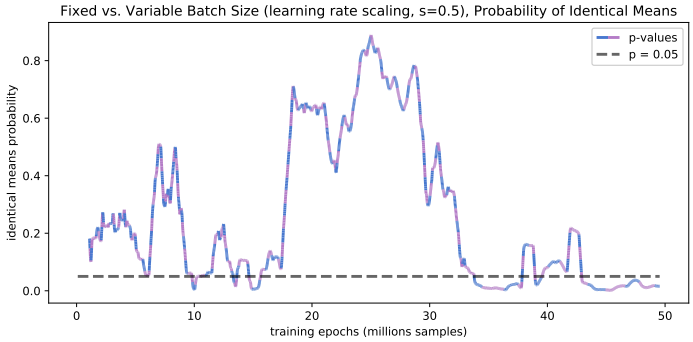


Figure 4.4: Welch's t-test results (fixed vs. variable batch size with learning rate scaling, $s = 0.5$) against the $p = 0.05$ line.

that in the last 15 epochs our hypothesis is correct with $p < 0.05$. In order to fully prove it, more experiments would have to be carried out. However, due to the computation time required to run each experiment, acquiring a larger sample size was outside of the author's capability give.

These experiments were performed mostly as an ablative analysis, so rigorous outcome was not the goal. However, the results demonstrate what has been said in the previous chapter, that the fixed batch size version is the most stable one with the fewest problems. Also the fixed batch size variant is the scheme most commonly used in practice (e.g. in the A2C implementation in OpenAI Baselines [38].

As a result of this, the fixed batch size variant has been selected for all the subsequent experiments performed in this work.

4.2 Strategic Attentive Writer

One of the goals of this project was to implement an existing RL method as is on the complex environment that has been selected or designed by the author. We test the implementation of the agent on the GridMaze environment, which was used in the original paper [8].

To show, that our implementation is correct, we ran it on a 7×7 maze and compared it to a LSTM baseline, as seen in Fig. 4.5. The aim of this experiment was purely qualitative, to show, that the implementation indeed converges. To further prove the point, in Fig. 4.6 is shown the average length of commitment in steps in any given time of the training. The average is computed only over a some recent history, not the entire history (it is a moving average). In the Fig. 4.6 we can see, that the agent is indeed initialized to creating 10-step plans, as described in Section 3.2, then during training, it eventually shortens the macro-actions up until an average length of 5.5. This

means that the model indeed does learn to create useful macro-actions that enable it to successfully solve the environment.

Let us discuss the possible reasons, why the STRAW agent takes so much longer to converge than the LSTM baseline. First of all, the initialization. The agent starts with re-planning every 10 steps, without having any prior knowledge about the environment. Notice, that after the training is finished, the average re-planning horizon is 5.5 steps, which strongly suggests that for this environment shorter re-planning horizons are better than longer ones. It takes time for the agent to discover this and eventually reduce the plan length, as seen in Fig. 4.6. A second argument we present, is that unlike the LSTM agent, the STRAW agent has to discover useful macro-actions. If we considered only purely deterministic policies, with 4 possible actions and average re-planning horizon of 5, this gives us $4^5 = 1024$ possible 5-step plans or macro-actions. Clearly, the agent has to deal with this combinatorial explosion and it does so successfully, however, at the cost of speed of convergence.

Even though the agent takes longer to converge, it has been shown by the authors, that indeed those learned macro-actions can lead to much higher average rewards in later stages of training (when this discovered structure can be fully exploited) in more complex environments where the LSTM model doesn't cope well. The complex environments in question are several Atari 2600 games, unfortunately, due to the overwhelming computational demands to perform thorough quantitative analysis on such environments (in the paper they did 200 experiments per model, per environment), it is beyond scope of this work.

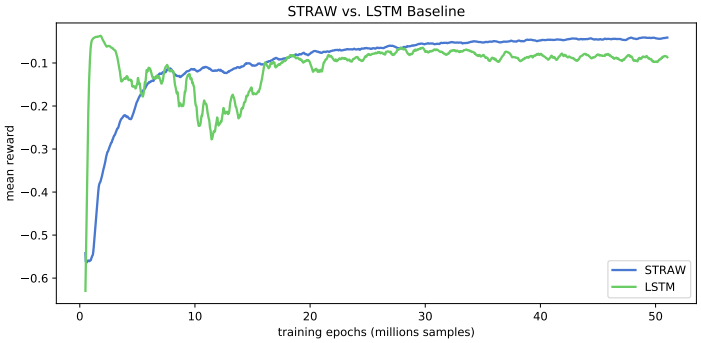


Figure 4.5: Comparison of the STRAW model and the LSTM baseline.

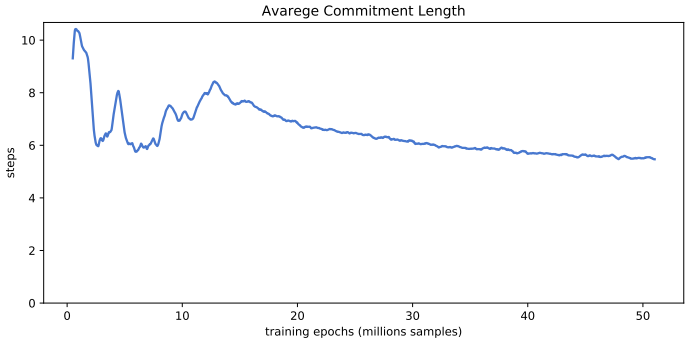


Figure 4.6: The average plan length of the agent.

4.3 Feudal Networks with Structured Exploration

Another goal of this work was to take one of the implemented methods and suggest a meaningful modification in attempt to improve the original method. This modified agent is then tested on the novel MazeRooms environment thoroughly, and we also perform a basic qualitative the analysis on the standard Enduro Atari 2600 environment. As introduced in the previous chapter, the specific modification is introducing the structured exploration feature from STRAW model. The noise layer is placed between the feature detector and manager.

4.3.1 MazeRooms

The hypothesis that we are testing is that the agent which uses the structured exploration will have better convergence rate on the MazeRooms environment than the one which doesn't use it. Furthermore, we claim that this improvement is not simply due to an increased number of parameters, neither is it due to an additional nonlinearity but thanks to using the noise layer.

In order to check the hypothesis, four different experiments have been devised. All test the FeUdal Networks agent on the MazeRooms environment (specifically a setup with 3×3 map, containing all the rooms, with each room being 5×5 and containing 2 randomly generated obstacles) but with slightly different architectures. To test the first part of our hypothesis we execute an agent that uses the noise layer and as a baseline we execute an agent that corresponds to the original version of the model. In order to test the second part of the hypothesis, we train an agent that replaces the noise layer with a linear layer (because the noise layer can be seen as a linear layer with added noise from $\mathcal{N}(0, \sigma)$ as is illustrated in Eqn. 2.9, therefore one of the tests is to remove the noise and just use the generated means), we call it a linear dummy agent. And to test the last part, that it is not just any non-linearity

that brings increase in performance, we execute an agent similar to the linear dummy, but this time we put the ReLU non-linearity behind it.

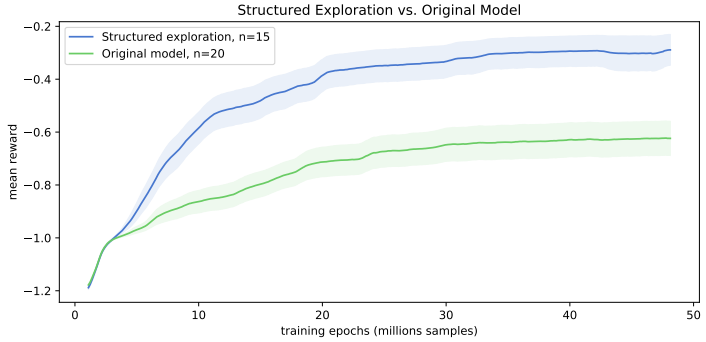


Figure 4.7: Comparison of the means of both populations along with sample standard deviation bands (n is the sample size for each experiment).

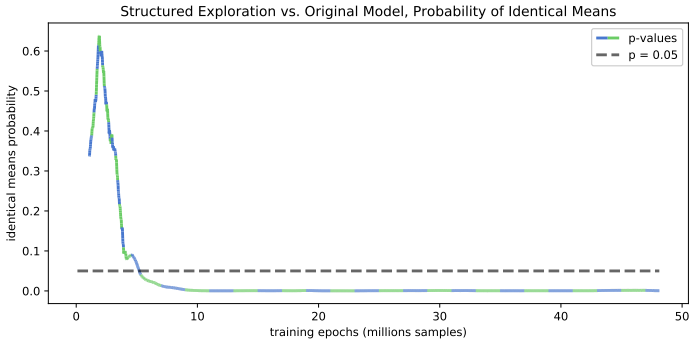


Figure 4.8: Welch's t-test results against the $p = 0.05$ line.

Fig. 4.7 suggests that indeed the first part of our hypothesis is true, to rigorously test this, once again, we use the Welch's t-test, with the result in Fig. 4.8. Clearly, only after 5 epochs, the hypothesis is true with $p < 0.05$, in fact, after 10 epochs, it holds true even with $p < 0.01$. This confirms that the structured exploration indeed improves the performance of the model.

In Fig. 4.9 we can see a comparison of the baselines used for validating the second part of our hypothesis. Once again, the plots are strongly indicative, that it holds true. To rigorously confirm this, we once again use the Welch's t-test [37], to compare the means of the structured exploration agent and the linear dummy agent (Fig. 4.10), and to compare the means of the structured exploration agent and the ReLU dummy agent (Fig. 4.11). Both figures once again tell us, that the means of the underlying distributions of the populations

are different with $p < 0.05$ (once again even $p < 0.01$ holds). This confirms the second part of our hypothesis.

Let us deem the hypothesis as confirmed. The structured exploration scheme indeed improves the performance of the FeUdal agent on the Maze-Rooms environment with statistical significance. Furthermore, we have confirmed that this improvement is not due to the increased number of parameters, neither is it due to additional non-linearity but really can be assigned to the structured exploration scheme.

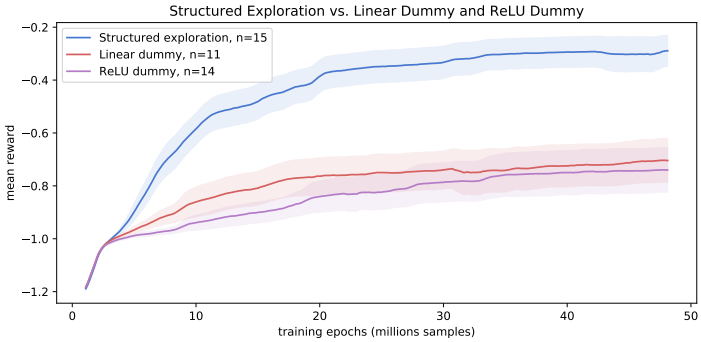


Figure 4.9: Comparison of the means of all the populations along with sample standard deviation bands (n is the sample size for each experiment).

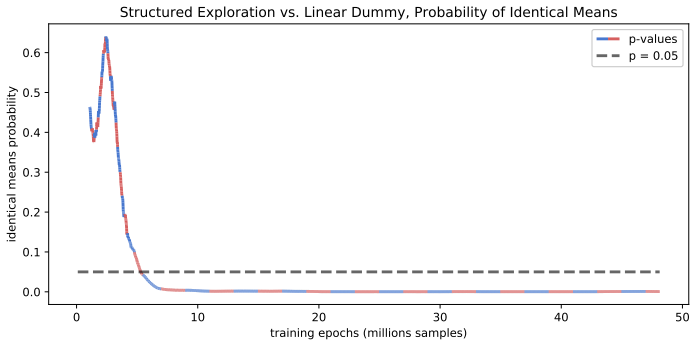


Figure 4.10: Welch's t-test results against the $p = 0.05$ line.

For the sake of completeness, Fig. 4.12 displays all the individual runs of the four different configurations studied here. We ran 15 experiments with the structured exploration agent, 20 experiments with the original model, 11 experiments with the linear dummy model, and 15 experiments with the ReLU dummy model. An interesting phenomenon that can be seen in the Fig. 4.12 is, that the agents often reach certain levels of performance that are similar

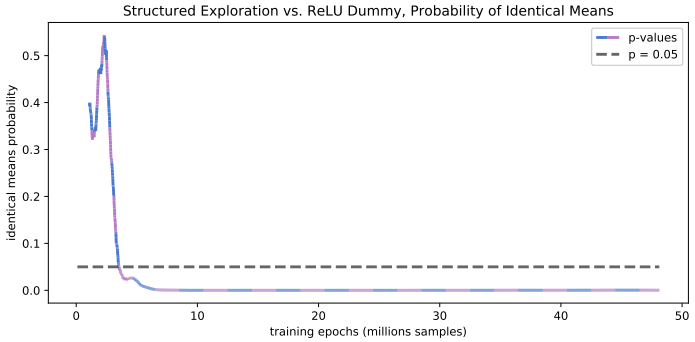
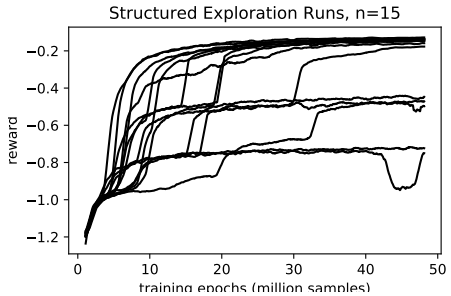
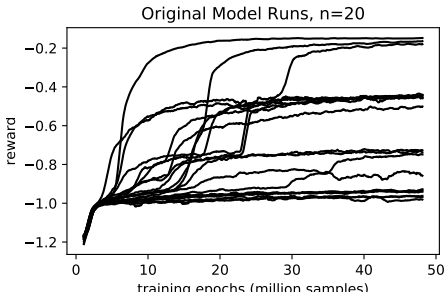


Figure 4.11: Welch's t-test results against the $p = 0.05$ line.

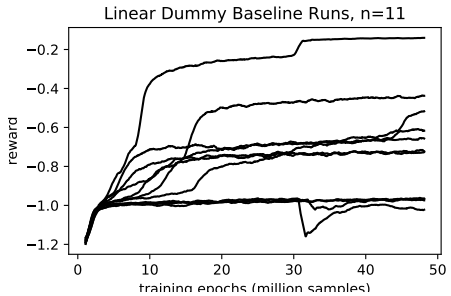
between runs and improve in quick qualitative leaps. This can be attributed to the structure of the environment. Recall, that the environment is composed of a 3×3 grid of rooms. These levels can *probably* be assorted to agents that learn to walk through different parts of the environment, the basic level could be an agent that can reach the goal only if it spawns in the same room as the agent, next level might be an agent that can reach goals only in the adjacent rooms, etc.



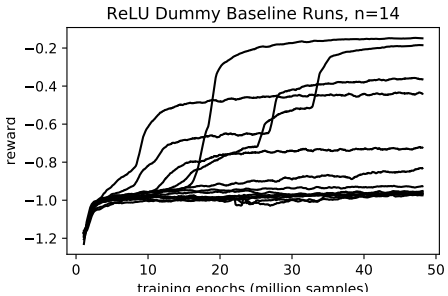
(a) Structured exploration runs.



(b) Original model runs



(c) Linear dummy runs.



(d) ReLU dummy runs.

Figure 4.12: Individual runs of all the experiments presented here.

In addition to the validation of our hypothesis, we also ran experiments on the same MazeRooms environment but with an LSTM agent, that has the same number of parameters as the structured exploration model.

Last but not least, as a part of the qualitative analysis of the algorithm, we also ran experiments with an LSTM model on the same MazeRooms environment, as a baseline for the general performance of the agent. The results are displayed in Fig. 4.13 along with the mean of the structured exploration agent runs. We can see that in fact, the LSTM baseline on average outperforms our best agent quite significantly. A possible explanation is that the FeUdal Networks model doesn't scale down in a linear fashion (in terms of its performance). Because the authors were using the A3C [15] algorithm, which utilizes only CPU, they could provide the agent a lot more memory. We used a GPU implementation, and GPUs are quite limited in this aspect. Therefore, in order to use models with reasonable unroll lengths and batch sizes, we had to scale down the model significantly; we reduced the number of units in each layer by the factor of 4 (another crucial reason for the scale-down was achieving a reasonable computation time). It might be, that this reduces the capabilities of the model in a *non-linear* manner. This was not a problem in our experiment as we compared only slightly modified versions of a single agent to each other.

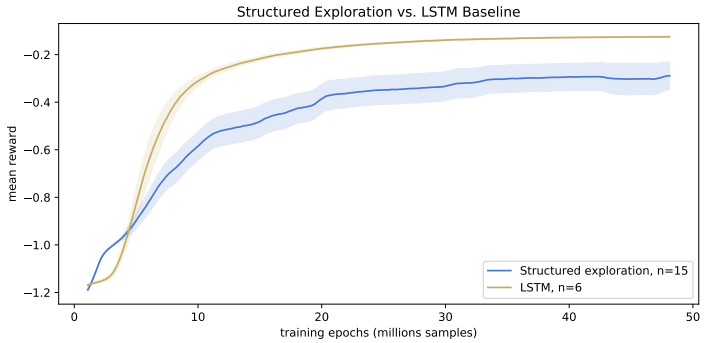


Figure 4.13: Comparison of the structured exploration and the LSTM baseline (n is the sample size for each experiment).

4.3.2 Enduro, Atari 2600

Last goal of this work was to perform a qualitative analysis of our improved method on a standard benchmarking reinforcement learning domain. As was mentioned earlier, we selected the game Enduro for Atari 2600 because the FeUdal Networks agent performed exceptionally well on it, in the original paper [2].

The Atari 2600 environments are quite computationally demanding (in comparison to our simpler environments) which poses a significant challenge for the implementation. Unfortunately, the implementation, that was used for all the other experiments presented in this work was not efficient enough to fit the whole model into GPU memory, while maintaining a reasonable batch size and unroll length, thus, both the A2C algorithm and the FeUdal Networks algorithm had to be re-implemented for this particular experiment. This was problematic and took a long time to debug as it required a different approach to unrolling and forming batches. In the end, the author was successful in the implementation but not a lot of time remained to perform the experiments (combined with the fact, that the Atari experiments took from 3 to 20 times longer than the other ones, and required a lot more computational resources). Therefore, the author managed to perform only two experiments. One with the structured exploration agent and one with the original model. The results are displayed in Fig. 4.14, and seem inconclusive, neither of the agents significantly outperforms the other, more experiments would have to carried out.

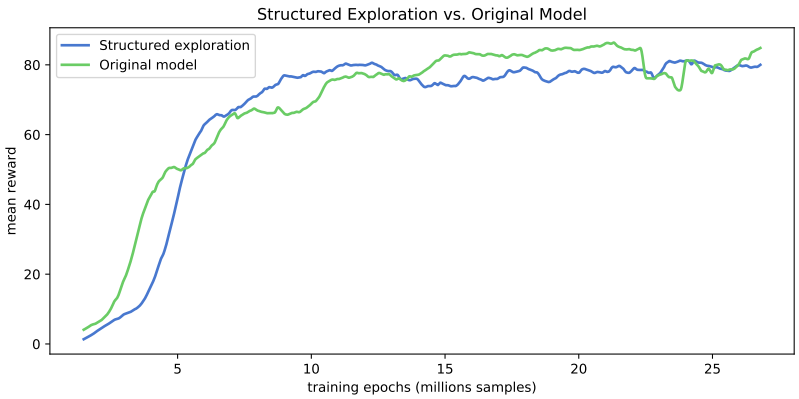


Figure 4.14: Comparison of the structured exploration and the original model.

4.4 Future Work

Integration of the structured exploration scheme from Strategic Attentive Writer into the FeUdal Networks model is a promising direction. We have only tested inserting the noise layer before manager but in Section 3.3 we have presented three more options to where the noise layer could be added. Furthermore, it is also possible to put the noise layer into the model on multiple locations at once. We have shown an undeniable increase in performance only for a scaled-down version of the original model and on a custom environ-

ment, naturally it would be interesting to explore the structured exploration scheme in FeUdal Networks but with a full-sized model and on multiple games in the Atari 2600 domain (which is considered a benchmarking standard in reinforcement learning).

The structured exploration scheme as presented in STRAW [8] draws its inspiration from Variational Auto-Encoders [28], which utilize the noise layer in the middle of the model. The application of noise in Auto-Encoders is based on strong mathematical reasoning. This work's aim was mainly empiric and even though many things were explained in terms of deeper intuitive concepts, rigorous theoretical reasoning on *why* the structured exploration scheme works well, is beyond the scope of this work. Studying this phenomenon theoretically could potentially bring new insight into the exploration-exploitation dilemma.

In the noise layer, we used a normal distribution and for the KL-divergence prior we used a normal distribution with zero mean and unit variance. It would be interesting to study, whether different distributions would work as well or how would a different prior affect the performance.

Conclusion

In this work, we studied hierarchical reinforcement learning methods. First, we have discussed two contemporary approaches, the Strategic Attentive Writer and FeUdal Networks. We have proposed a complex structured environment, MazeRooms, that emulates some elements of difficult environments such as Montezuma's Revenge, also it is highly scalable and adjustable and allows for cheap evaluation of the reinforcement learning agents.

We have implemented both of the mentioned methods. We have tested the STRAW model on the GridMaze environment. We proposed using a structured exploration scheme from STRAW in order to improve the performance of FeUdal Networks. We have shown, that the structured exploration scheme significantly improves performance of the FuN agent on the MazeRooms environment. We have performed a basic qualitative analysis of the proposed method on a standard reinforcement learning benchmarking domain, Atari 2600, specifically on the game Enduro. The results of this analysis were not conclusive and require further study.

Deep reinforcement learning today is a rapidly growing area. As computation is getting cheaper, larger and more powerful models than ever are being conceived and tested. Because of the scarred and bumpy history of AI (the two AI winters) a lot of researchers were careful about voicing their excitement and realizing their visions, afraid of funding cuts and ridicule. Today, however, this is changing, people are openly and systematically tackling the problem of developing AGI and real measurable progress is being made. Let us hope, that the people and the organizations, that might successfully develop such systems use them for the benefit of us all and not just for their selfish agenda. Let us hope, that the future won't bring autonomous warfare, or personalized surveillance and censorship, but rather systems capable of solving humanities' toughest problems for the universal good.

Bibliography

[1] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[2] Tom Schaul Nicolas Heess Max Jaderberg David Silver Alexander Sasha Vezhnevets, Simon Osindero and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning, 2017.

[3] Ilya Sutskever Alex Krizhevsky and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[4] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[5] Chris J. Maddison Arthur Guez Laurent Sifre George van den Driessche Julian Schrittwieser Ioannis Antonoglou Veda Panneershelvam Marc Lanctot Sander Dominik Grewe John Nham Nal Kalchbrenner Ilya Sutskever Timothy Madeleine Leach Koray Kavukcuoglu Thore Graepel David Silver, Aja Huang and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[6] Lila Gleitman and Anna Papafragou. *Relations Between Language and Thought*. 2005.

[7] Jason Kuen Lianyang Ma Amir Shahroudy Bing Shuai Ting Liu Xingxing Wang Jiuxiang Gu, Zhenhua Wang and Gang Wang. Recent advances in convolutional neural networks. *CoRR*, abs/1512.07108, 2015.

[8] John Agapiou Simon Osindero Alex Graves Oriol Vinyals Alexander Sasha Vezhnevets, Volodymyr Mnih and Koray Kavukcuoglu. Strategic attentive writer for learning macro-actions, 2016.

[9] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning : An Introduction*. MIT Press, 1998.

[10] Andrej Karpathy and Justin Johnson. Cs231n convolutional neural networks for visual recognition.

[11] Andrew Senior Haşim Sak and Françoise Beaufays. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition, 2014.

[12] Jan Koutník Bas R. Steunebrink Klaus Greff, Rupesh Kumar Srivastava and Jürgen Schmidhuber. Lstm: A search space odyssey. 2015.

[13] John Berkowitz Zachary C. Lipton and Charles Elkan. A critical review of recurrent neural networks for sequence learning, 2015.

[14] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256, May 1992.

[15] Mehdi Mirza Alex Graves Timothy P. Lillicrap Tim Harley David Silver Volodymyr Mnih, Adrià Puigdomènech Badia and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. 2016.

[16] Dhruva Tirumala Hubert Soyer Joel Z Leibo Remi Munos Charles Blundell Dharshan Kumaran Jane X Wang, Zeb Kurth-Nelson and Matt Botvinick. Learning to reinforcement learn, 2016.

[17] Prafulla Dhariwal Alec Radford John Schulman, Filip Wolski and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[18] OpenAI. Openai baselines: Acktr & a2c, Nov 2017.

[19] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In Sven Koenig and Robert C. Holte, editors, *Abstraction, Reformulation, and Approximation*, pages 212–223, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[20] Peter Dayan and Geoffrey E. Hinton. Feudal reinforcement learning. In *Advances in Neural Information Processing Systems 5, [NIPS Conference]*, pages 271–278, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[21] Tom Zahavy Daniel J. Mankowitz Chen Tessler, Shahar Givony and Shie Mannor. A deep hierarchical approach to lifelong learning in minecraft. *CoRR*, abs/1604.07255, 2016.

[22] Amy McGovern and Andrew G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pages 361–368, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[23] Shie Mannor Ishai Menache and Nahum Shimkin. Basis function adaptation in temporal difference reinforcement learning. *Annals of Operations Research*, 134(1):215–238, 2005.

[24] David Silver and Kamil Ciosek. Compositional planning using optimal option models, 2012.

[25] Jean Harb Pierre-Luc Bacon and Doina Precup. The option-critic architecture. *CoRR*, abs/1609.05140, 2016.

[26] Alex Graves Danilo Jimenez Rezende Karol Gregor, Ivo Danihelka and Daan Wierstra. Draw: A recurrent neural network for image generation, 2015.

[27] Xi Chen John Schulman and Pieter Abbeel. Equivalence between policy gradients and soft q-learning, 2017.

[28] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

[29] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions, 2015.

[30] T. Tieleman and Geoffrey E. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[31] Joel Veness Marc G. Bellemare, Yavar Naddaf and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. 2012.

[32] Alistair Muldal Tom Erez Yazhe Li Diego de Las Casas David Budden Abbas Abdolmaleki Josh Merel Andrew Lefrancq Timothy Lillicrap Yuval Tassa, Yotam Doron and Martin Riedmiller. Deepmind control suite, 2018.

[33] Ludwig Pettersson Jonas Schneider John Schulman Jie Tang Greg Brockman, Vicki Cheung and Wojciech Zaremba. Openai gym, 2016.

[34] Denis Teplyashin Tom Ward Marcus Wainwright Heinrich Küttler Andrew Lefrancq Simon Green Víctor Valdés Amir Sadik Julian Schrittwieser Keith Anderson Sarah York Max Cant Adam Cain Adrian Bolton Stephen Gaffney Helen King Demis Hassabis Shane Legg Charles Beattie, Joel Z. Leibo and Stig Petersen. Deepmind lab, 2016.

[35] David Bruce Wilson. Generating random spanning trees more quickly than the cover time. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 296–303, New York, NY, USA, 1996. ACM.

[36] David Silver Alex Graves Ioannis Antonoglou Daan Wierstra Volodymyr Mnih, Koray Kavukcuoglu and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[37] B. L. WELCH. The generalization of 'student's' problem when several different population variances are involved. *Biometrika*, 34(1-2):28–35, 1947.

[38] Oleg Klimov Alex Nichol Matthias Plappert Alec Radford John Schulman Szymon Sidor Prafulla Dhariwal, Christopher Hesse and Yuhuai Wu. Openai baselines. `https://github.com/openai/baselines`, 2017.

Appendix A

Experiment Details

Here we share specific architecture details, paths to iPython notebooks used to generate all plots, run IDs and hyper-parameter tables. The run IDs are internal IDs used in the framework developed as a part of this work. Those can be used to identify specific runs in the folder `src/fs-learning/runs` or to test the specific models using the script `src/fs-learning/fs_learning/test.py`. For example running `python -m fs_learning.test -r 1551 -b -nc 2 -m 0.33 -g 0 -re` executes the test script that loads the best model (`-b`) from the run 1551 (`-r`), using the GPU with id 0 (`-g`), assigning 2 cores to TensorFlow (`-nc`) and running the test script in render mode (`-re`), which runs just a single agent and displays output of environments. The render mode works only for the GridMaze and MazeRooms environment.

An important remark is that in all of our experiments we are using a linear learning rate decay schedule from the initial value to zero.

A.1 LSTM Architecture

All the LSTM baselines in this work use the same architecture. A feature detector connected to the input (which is environment-specific, the details of each feature detector are presented in Chapter 3) a single LSTM layer on top and then an output layer consisting of a single linear neuron for the value function estimation and several SoftMax units for the policy (their number naturally is given by the number of available actions in a given environment).

A.2 A2C Experiments

The iPython notebook used to evaluate the runs and extract the plots is located in `src/fs-learning/fs_learning/notebooks/a2c_comparison.ipynb`.

A.2.1 Run IDs

Table A.1: The run IDs of the A2C experiments.

Experiment	**Run IDs**
Fixed batch size	1551, 1556, 1560, 1568, 1572
Variable batch size, fixed learning rate	1553, 1555, 1558, 1561, 1563
Variable batch size, variable learning rate, $s = 1$	1552, 1564, 1571, 1573, 1574
Variable batch size, variable learning rate, $s = 0.5$	1554, 1557, 1559, 1562, 1567

A.2.2 Hyper-Parameters

Table A.2: The hyper-parameters used for the four LSTM experiments.

Hyper-Parameter	**Value**	**Comment**
environment name	CartPole-v1	Code for OpenAI gym
max episode steps	200	Number of steps before episode terminates (T_{max} in A3C 6)
gamma	0.99	
RMSProp decay	0.95	
learning rate	0.001	
entropy regularization loss weight	0.001	
LSTM units	50	
unroll steps	20	Depth of the unrolled recurrent graph (BPTT steps)
number of agents	16	
batch size	320	This is also the reference batch size used in the learning rate scaling experiments

A.3 Strategic Attentive Writer Experiments

The iPython notebook used to evaluate the runs and extract the plots is located in `src/fs-learning/fs_learning/notebooks/straw.ipynb`.

A.3.1 Run IDs

Table A.3: The run IDs of the STRAW experiments.

Experiment	**Run IDs**
STRAW	4450
LSTM baseline	4454

A.3.2 Hyper-Parameters

Table A.4: The STRAW hyper-parameters.

Hyper-Parameter	**Value**	**Comment**
environment name	GridMaze7x7-v0	Code for OpenAI gym (the custom environments are using it too)
max episode steps	50	Number of steps before episode terminates (T_{max} in A3C 6)
gamma	0.99	
use structured exploration	True	
noise layer units	128	
RMSProp decay	0.95	
learning rate	0.001	
re-plan penalty loss weight	$5 \cdot 10^{-5}$	λ from 2.12
KL-divergence loss weight	10^{-6}	
entropy regularization loss weight	0.001	
STRAW hidden units (in the embedding h)	256	
plan size	64 steps	T
bias b initial value	0	
action-plan filters	10	K
unroll steps	20	Depth of the unrolled recurrent graph (BPTT steps)
number of agents	16	
batch size	512	

Table A.5: The hyper-parameters for the LSTM baseline.

Hyper-Parameter	**Value**	**Comment**
environment name	GridMaze7x7-v0	
max episode steps	100	Number of steps before episode terminates (T_{max} in A3C 6)
gamma	0.99	
RMSProp decay	0.95	
learning rate	0.001	
entropy regularization loss weight	0.001	
LSTM units	32	
unroll steps	10	Depth of the unrolled recurrent graph (BPTT steps)
number of agents	16	
batch size	160	

A.4 FeUdal Networks MazeRooms Experiments

The iPython notebook used to evaluate the runs and extract the plots is located in `src/fs-learning/fs_learning/notebooks/feudal_mgrnoise_vs_nonoise.ipynb`

A.4.1 Run IDs

Table A.6: The run IDs of the FeUdal MazeRooms experiments.

Experiment	**Run IDs**
Structured exploration	3206, 3209, 3211, 3213, 3215, 1471, 1473, 1475, 1477, 1479, 1481, 1484, 1485, 1489, 1492
Original model	3207, 3208, 3210, 3212, 3214, 3216, 3235, 1470, 1472, 1474, 1476, 1478, 1480, 1482, 1483, 1486, 1487, 1488, 1490, 1491
Linear dummy	3223, 3243, 3246, 3251, 1501, 1512, 1515, 1518, 1538, 1540, 1544
ReLU dummy	3224, 3236, 3241, 3245, 3249, 1500, 1513, 1516, 1519, 1522, 1530, 1535, 1539, 1543
LSTM baseline	3228, 3237, 3242, 3248, 3252, 3254

A.4.2 Hyper-Parameters

Table A.7: The hyper-parameters for the four FeUdal Networks configurations.

Hyper-Parameter	**Value**	**Comment**
environment name	MazeRooms Obstacle4-v0	This environment is also integrated into the OpenAI Gym API
max episode steps	100	Number of steps before episode terminates (T_{max} in A3C 6)
manager gamma	0.99	
worker gamma	0.95	
intrinsic reward weight	0.1	
RMSProp decay	0.99	
learning rate	0.001	
entropy regularization loss weight	0.001	
KL-divergence loss weight	10^{-5}	
noise layer units	128	
manager f^{Mspace} units	64	
manager dLSTM f^{Mrnn} units	64	
worker f^{Wrnn} units	64	
worker sub-policies count	8	
manager value function hidden layer units	32	
worker value function hidden layer units	32	
dLSTM radius	6	c, r
unroll steps	50	Depth of the unrolled recurrent graph (BPTT steps)
number of agents	8	
batch size	400	

Table A.8: The hyper-parameters for the LSTM baseline.

Hyper-Parameter	**Value**	**Comment**
environment name	MazeRooms Obstacle4-v0	
max episode steps	100	Number of steps before episode terminates (T_{max} in A3C 6)
gamma	0.99	
RMSProp decay	0.99	
learning rate	0.001	
entropy regularization loss weight	0.001	
LSTM units	105	This number was chosen to match the number of parameters of the structured exploration agent
unroll steps	50	Depth of the unrolled recurrent graph (BPTT steps)
number of agents	8	
batch size	400	

A.5 FeUdal Networks Atari 2600 Experiments

The iPython notebook used to evaluate the runs and extract the plots is located in `src/fs-learning/fs_learning/notebooks/atari.ipynb`

A.5.1 Run IDs

Table A.9: The run IDs of the FeUdal Atari experiments.

Experiment	**Run IDs**
Structured exploration	1603
Original model	1602

A.5.2 Hyper-Parameters

Table A.10: The hyper-parameters for the two FeUdal Networks configurations.

Hyper-Parameter	**Value**	**Comment**
environment name	Enduro-v0	The OpenAI Gym environment ID
max episode steps	1200	Number of steps before episode terminates (T_{max} in A3C 6)
manager gamma	0.99	
worker gamma	0.95	
intrinsic reward weight	0.1	
RMSProp decay	0.99	
learning rate	0.0001	
entropy regularization loss weight	0.001	
KL-divergence loss weight	$3.3 \cdot 10^{-6}$	
noise layer units	128	
manager f^{Mspace} units	64	
manager dLSTM f^{Mrnn} units	64	
worker f^{Wrnn} units	64	
worker sub-policies count	8	
manager value function hidden layer units	32	
worker value function hidden layer units	32	
dLSTM radius	10	c, r
unroll steps	100	Depth of the unrolled recurrent graph (BPTT steps)
number of agents	2	
batch size	200	

Appendix B

Contents of DVDs

Here we describe the basic directory structure of the two attached DVDs. The first one contains all the source code and some of the trained models along with their configuration files. The second DVD contains the rest of the trained models along with configuration files used for each run. In order to use the framework, the folders from both DVDs have to be merged together.

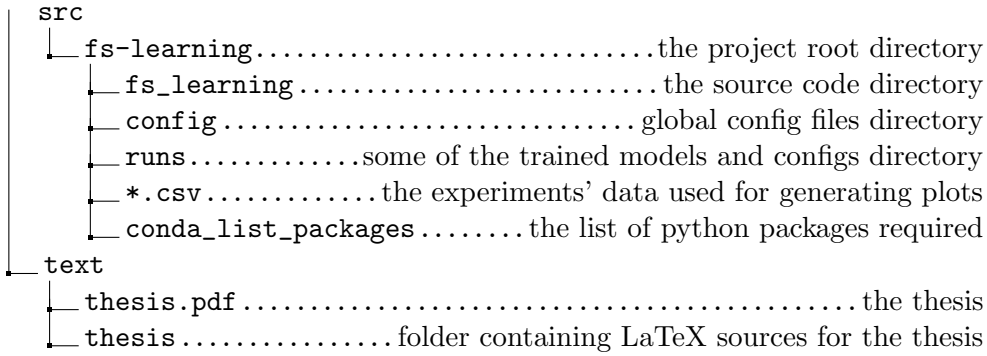


Figure B.1: The directory structure of the first DVD

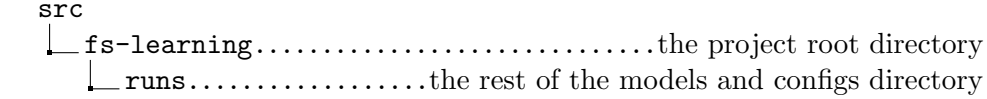


Figure B.2: The directory structure of the second DVD