Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Architecture Optimization for Multiple Instance Learning Neural Networks

Bc. Nikita Tishin

Supervisor: Ing. Jan Drchal, Ph.D.
Field of study: Artificial Intelligence
August 2020

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Tishin Nikita**	Personal ID number:	**452982**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence**		

II. Master’s thesis details

Master’s thesis title in English:

Architecture Optimization for Multiple Instance Learning Neural Networks

Master’s thesis title in Czech:

Optimalizace architektury neuronových sítí pro multiple instance learning

Guidelines:

The task is to develop, implement, and experiment with algorithms optimizing specialized neural networks that solve multiple instance learning (MIL) problems. MIL networks allow to directly process variable-length inputs in the form of multisets (bags). Moreover, these can be extended to work with general tree structures (such as JSON files). Finding optimal MIL network architecture with respect to a given dataset can be time-consuming; hence we search for methods that will automate it.
1) Familiarize yourself with neural network approaches to multiple instance learning (MIL) and neural architecture search (NAS) methods.
2) Propose a NAS method aimed to optimize architectures of neural network MIL models with tree-data extensions.
3) Evaluate your method on available benchmark datasets.

Bibliography / sources:

[1] Using Neural Network Formalism to Solve Multiple-Instance Problems, Tomáš Pevný, Petr Somol, 2016
[2] Discriminative models for multi-instance problems with tree-structure, Tomáš Pevný, Petr Somol, 2016
[3] Elsken, Thomas, Jan Hendrik Metzen, and Frank Hutter. &quot;Neural architecture search: A survey.&quot; arXiv preprint arXiv:1808.05377 (2018).

Name and workplace of master’s thesis supervisor:

Ing. Jan Drchal, Ph.D., Artificial Intelligence Center, FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **04.02.2020** Deadline for master's thesis submission: **14.08.2020**

Assignment valid until: **30.09.2021**

Ing. Jan Drchal, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

Jsem velmi vděčný své rodině a přátelům za jejich podporu. Rád bych také vyjádřil svou vděčnost svému vedoucímu a všem členům Fakulty Elektrotechnické za jejich práci.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 14. August 2020

Abstract

As Deep Learning continues to make significant progress in solving various complex tasks, there is a growing need to discover novel neural network architectures, which is a complicated and time-consuming process. Neural Architecture Search (NAS) methods have emerged to automate network architecture design utilizing advanced Machine Learning techniques, such as Reinforcement Learning and Deep Learning itself. Currently, NAS has been applied mostly on convolutional and recurrent types of architectures. This thesis aims to transfer existing NAS methods to design network architectures capable of solving tasks defined on sets of instances, referred to as Multiple Instance Learning (MIL) tasks.

To evaluate the effectiveness of NAS methods in the MIL setup, we implement selected search algorithms and conduct extensive experiments on benchmark datasets. We conclude that while Neural Architecture Search provides a practical framework to optimize MIL network architectures, search space design is crucial to architecture optimization.

Keywords: Machine Learning, Multiple Instance Learning, Neural Architecture Search, AutoML, Optimization

Supervisor: Ing. Jan Drchal, Ph.D.
centrum umělé inteligence FEL

Abstrakt

Hluboké učení dosahlo značného pokroku při řešení různých složitých úkolů, a je stále nutné objevovat nové architektury neuronových sítí, což je komplikovaný a časově náročný proces. Za tímto účelem roste zájem o metody automatického vyhledávání architektur neuronových síti (Neural Architecture Search, NAS). Nové metody vyhledávání využívají pokročilé techniky strojového učení, jako je například posilované a hluboké učení. V současné době se NAS používá převážně pro konvoluční a rekurentní typy architektur. Cílem této práce je přenést stávající metody NAS do návrhu síťových architektur schopných řešit úlohy definované na množinách instancí (Multiple Instance Learning).

Pro vyhodnocení účinnosti metod automatického vyhledávání architektur pro Multiple Instance Learning jsme implementovali vybrané vyhledávací algoritmy a provedli jsme rozsáhlé experimenty na několika datových sadách. Došli jsme k závěru, že zatímco NAS poskytuje praktické nástroje pro optimalizaci architektur neuronových sítí, pro optimalizaci architektury je rozhodující návrh prohledávácího prostoru.

Klíčová slova: Strojové účení, Multiple Instance Learning, Neural Architecture Search, AutoML, Optimalizace

Překlad názvu: Optimalizace architektury neuronových sítí pro multiple instance learning

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Motivation and Goals

This work combines two recently emerged machine learning topics: Neural Multiple Instance Learning and Neural Architecture Search. A typical Machine Learning algorithm is designed to operate on fixed-length vectors of inputs and outputs. However, in some cases, it is desirable to represent the data as a set of items, which requires models capable of processing inputs of variable length without imposing a specific ordering of items. This setting is referred to as **Multiple Instance Learning**. The problem was around for some time, but recent progress in Deep Learning has motivated new methods based on Neural Networks designed to operate on sets of data samples.

Designing a neural network architecture requires much expertise and can be time-consuming, making this process desirable to automate. As Deep Learning continues to make progress in various tasks, the topic of automated **Neural Architecture Search** has attracted increasing attention. Recent developments in NAS show that automatically designed architectures can outperform prior art manually designed architectures in image classification, semantic object segmentation, and other problems.

Currently, NAS has been applied mostly on convolutional and recurrent types of architectures. In this work, we transfer state of the art knowledge to design MIL networks automatically. Employing NAS can be beneficial for understanding novel architectures as well as just increasing raw performance on the tasks.

The main goal is to adapt existing NAS techniques to the domain of Multiple Instance Learning, evaluate them in different settings, and propose a method that works best.

Chapter 2

Theoretical background

This chapter contains an overview of the theoretical results related to Multiple Instance Learning and Neural Architecture Search.

2.1 Multiple Instance Learning with Neural Networks

Multiple Instance Learning (**MIL**) is a type of a learning problem where a model operates on sets of instances instead of just operating on individual instances as most common machine learning models do. In MIL terminology, a multi-set of instances is called a *bag*, since the items it contains are not necessarily unique. Typically, MIL problems involve supervised learning where the target variable is available only on the level of bags and is unknown for individual instances. Modeling the bag-level target variable can be quite challenging since many factors should be taken into account, such as complex interactions between individual instances in a bag or a presence of a single instance significant to the task. Problems that are naturally suitable for MIL formulation arise in different domains, such as drug discovery, classification of text documents, anomalous image detection, speaker identification, and network traffic classification. For a detailed discussion of MIL and its aspects, the reader is referred to [1].

The recent success of Deep Learning in a wide range of problems has motivated developments on Artificial Neural Networks capable of solving

Multiple Instance Learning problems. As a result, new types of network architectures that can handle sets were discovered. It has been shown that to perform supervised tasks on sets, it is both necessary and sufficient to use a permutation-invariant function, which maps input bags to the target variable[33]. Permutation-invariant mapping preserves its output with respect to the different orderings of the same input set. If such a mapping is also differentiable, it can be used in a neural network and be effectively trained with gradient descent algorithms.

Network architecture proposed by [23] incorporates instance-level processing layer with a permutation-invariant aggregation layer followed by a classifier. This formulation is focused on classification but is easily extended to any supervised task using a suitable loss function. Let $\mathcal{X}$ be a non-empty instance space. The formulation assumes that each bag $B \subseteq \mathcal{X}$ is a realization of some random variable with probability distribution $P(p_b, y)$, where p_b is a probability distribution producing individual instances and $y \in \mathcal{Y}$ is the bag label. The goal of the model is to learn a discrimination function $f : \mathcal{B} \rightarrow \mathcal{Y}$, where $\mathcal{B}$ is the set of all possible realizations of all possible distributions over $\mathcal{X}$. The forward pass of the proposed architecture is formulated as follows:

$$F(B; \theta, \theta_f) = f(g(\{\phi(x_i; \theta) | x_i \in B\}), \theta_f) \tag{2.1}$$

where $\phi : \mathcal{X} \rightarrow \mathbb{R}^m$ is a parameterized mapping which embeds individual instances into the space of real vectors, $g : \mathcal{P}^{\mathbb{R}^m} \rightarrow \mathbb{R}^m$ is an aggregation function and $f : \mathcal{R}^m \rightarrow \mathcal{Y}$ is a classifier network. Such units can be nested to process increasingly complex data structures, such as bags of bags[22].

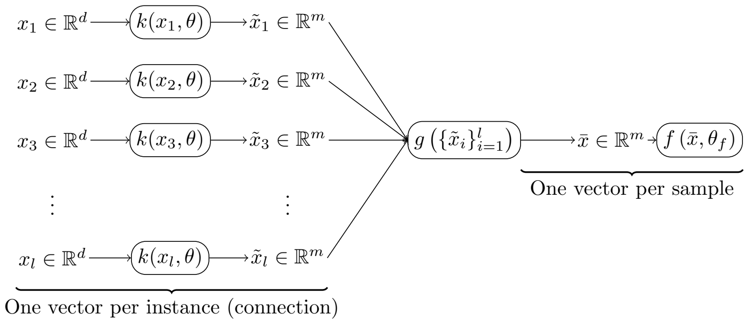
$x_1 \in \mathbb{R}^d$
$k(x_1, \theta)$
$\tilde{x}_1 \in \mathbb{R}^m$
$x_2 \in \mathbb{R}^d$
$k(x_2, \theta)$
$\tilde{x}_2 \in \mathbb{R}^m$
$x_3 \in \mathbb{R}^d$
$k(x_3, \theta)$
$\tilde{x}_3 \in \mathbb{R}^m$
$x_l \in \mathbb{R}^d$
$k(x_l, \theta)$
$\tilde{x}_l \in \mathbb{R}^m$
$g(\{\tilde{x}_i\}_{i=1}^l)$
$\bar{x} \in \mathbb{R}^m$
$f(\bar{x}, \theta_f)$
One vector per sample
One vector per instance (connection)

Figure 2.1: Sketch of the neural network optimizing the embedding in embedding-space paradigm. Source: [23].

Another type of processing unit proposed to solve a MIL problem is called the permutation-equivariant layer[33]. Permutation-equivariant layer maps an input set to an output set of the same size such that the output set preserves the ordering of the input set, i.e. $f([x_{\pi(1)}, \ldots, x_{\pi(M)}]) =$

$([f(x_{\pi(1)}), \ldots, f(x_{\pi(M)})])$ for any permutation π. It is shown that the only form this operation can take is

$$f(\mathbf{x}) = \sigma(\mathbf{x}\Lambda - g(\mathbf{x})\Gamma) \tag{2.2}$$

where $\mathbf{x} \in \mathbb{R}^{m\times d}$ is a bag of m instances stacked into a matrix, $g : \mathbb{R}^{m\times d} \to \mathbb{R}^{1\times d}$ is a commutative polling function, σ is a point-vise non-linearity and $\Lambda, \Gamma \in \mathbb{R}^{d\times d'}$ are model parameters. The output of this layer is a bag of transformed input instances $\mathbf{y} \in \mathbb{R}^{m\times d'}$. The advantage of permutation-equivariant layers is that their combination preserves permutation-equivariance property, which allows stacking them into deep networks.

A typical aggregation function used in MIL networks is one of the summation, the mean or the element-wise maximum function. However, the choice does not have to be limited to these simple functions. Attention-based aggregation functions have been shown to outperform standard pooling functions in some tasks[11]. Attention mechanism applied to the MIL problems enable importance-weighted aggregation. In general, attention-based pooling has the form of the weighted average:

$$g(\mathbf{B};\theta) = \sum_{\mathbf{x}\in\mathbf{B}} \alpha(\mathbf{x};\theta)\mathbf{x} \tag{2.3}$$

where $\alpha : \mathcal{X} \to [0, 1]$ is an attention function which serves as an importance weight which depends on the particular instance, and it holds that $\sum_{\mathbf{x}\in\mathbf{B}} \alpha(\mathbf{x};\theta) = 1$ for each bag $\mathbf{B}$. Attention can also provide useful insights into the importance of the individual instances for the bag classification.

To summarize, a general recipe for applying a neural network to bagged inputs is to combine the instance-level learning with the bag-level learning, and the main ingredient to it is the permutation-invariant aggregation function. Using neural networks as universal approximators to encode the instances before the aggregation provides a powerful mechanism to end-to-end learning of the bag embedding jointly with the task-specific post-processing network.

2.2 Neural Architecture Search

The term neural network architecture refers to the arrangement of neurons into layers and the connection patterns between layers, activation functions, and learning methods. The architecture of a neural network determines how a network transforms its input into an output and how efficient the network is in performing the task.

Since designing and fine-tuning a neural network architecture requires a lot of expertise and can be time-consuming, there is growing interest in automated **Neural Architecture Search** (**NAS**) methods. Recent developments in NAS show that automatically designed architectures can outperform prior art manually designed architectures in image classification, semantic object segmentation and other problems. NAS is closely related to hyperparameter optimization and AutoML paradigm.

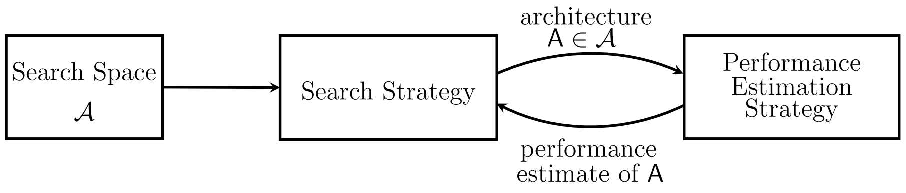

Figure 2.2: Abstract illustration of Neural Architecture Search methods. Source: [8]

NAS methods can be differentiated by three main components which constitute a search algorithm:

- **Search space** defines a set of architectures which can be discovered in principle.

  Generally search spaces can easily be of infinite size, which makes complete exploration not just impractical, but simply impossible. One of the popular and methods to produce a more compact search space is to optimize "micro architecture" - the architecture of a smaller unit ("cell") which are stacked in deep learning fashion to produce a "macro architecture". Although this method of search space design has proved successful, it introduces *human bias* into the search procedure, preventing discovery of completely novel architectures. The trade-off between exploitation (and efficiency) and exploration is quite common for search procedures in general and should be considered when developing a new NAS method. The choice of the search space determines the optimization problem and suitable optimization methods.

- **Search strategy** guides the exploration of the search space. An example of a popular and effective strategy for small search spaces is random search – uniform sampling from the set of all possible configurations. Search strategies are discussed in more detail in the following section.

- **Performance estimation strategy** is the process of estimating the performance of candidate architectures.

  A naive way to estimate the performance is to train and evaluate the network, which takes a prohibitive amount of time in the NAS setting, where estimation procedure is run thousands of times. Various techniques are employed to improve performance estimation time. *Proxy metrics* are lower bound estimates of real architecture performance. They can be obtained e.g. by training for a small amount of epochs or on a smaller subset of data. Other possibilities include curve extrapolation, sharing weights across multiple candidates and using surrogate models which can learn and predict performance of networks in search space. Performance estimation process is often governed by the search strategy.

A good NAS algorithm thus ideally should satisfy following characteristics:

- ability to handle large or even unbounded search spaces;
- potential to find novel architectures which are not necessarily based on current conventions in manual design;
- it should be equipped with a suitable performance estimation strategy to avoid computational overhead of training a large amount of candidate architectures.

In general the problem of NAS is intractable unless some compromises are made. For a survey of NAS methods, reader is referred to [8].

2.2.1 State-Of-The-Art Review

This section provides a short overview of most notable research directions in NAS.

Evolution

Evolutionary optimization is a traditional method of finding neural architectures. The process follows a general evolutionary algorithm: population of architectures is evolved by producing offsprings via mutation of selected architectures. Mutation can be done with operations such as adding a new layer or a connection, altering hyperparameters of the existing layer and as well as altering training hyperparameters. Offsprings are then trained and added to the population if their fitness (e.g. performance on the validation set) is sufficient. Evolutionary scheme provides a lot of space for customization, raising a whole family of NAS methods. For a survey of evolutionary NAS, reader is referred to [27].

Model-based Optimization

Bayesian Optimization (**BO**) has already been successfully applied to the hyperparameter search for various algorithms, including Machine Learning[9][25]. Bayesian Optimization is an iterative search strategy for optimization of black-box functions which are typically costly to evaluate, which is the case for the validation loss of an untrained neural network. The main idea of the approach is to construct a *surrogate* probabilistic model which approximates actual objective function while being easier to optimize. Algorithm starts with a *prior* (uninformed) distribution as a surrogate model. At each iteration, surrogate objective is optimized to obtain candidate hyperparameters which are used to evaluate actual objective. Resulting value is used to update probabilistic model using Bayes Theorem, resulting in the *posterior* distribution. **Sequential Model-Based Optimization** [10] (**SMBO**) approach generalizes over Bayesian Optimization. It uses a surrogate function to estimate real objective and acquisition function to select candidate hyperparameters. Gaussian Process and Tree Parzen Estimators are popular choices for implementing a surrogate model, and Expected Improvement can be used as the acquisition function[3]. These procedures are useful to optimize relatively small real-valued or discrete hyperparameter spaces. A general search space in NAS problem is too large for this case.

Auto-Keras system [15] adapts Bayesian Optimization to the search space of neural architectures. It uses Gaussian Process as the surrogate model. To circumvent the restriction of Gaussian Processes which are defined only on real domains, authors use a kernel trick and introduce an edit-based distance kernel for neural networks. They use an Upper-confidence Bound as an acquisition function to balance between exploration and exploitation.

The method is provided with an efficient implementation, utilizing parallel computations.

Progressive NAS (PNAS) [18] uses a SMBO strategy which searches architectures in order of increasing complexity. However, it is unconventional in a way that it uses a Recurrent Neural Network as a surrogate model. Search space of PNAS consists of neural cells which are composed from successively stacked blocks. Acquisition of new architecture parameters is done in a progressive fashion: starting with a set of candidate cells consisting of B blocks, it iteratively updates K most promising architectures by expanding them with all possible blocks creating cells consisting of $B + 1$ blocks. Search space is limited by the maximum number of blocks in the cell.

Reinforcement Learning

NAS can be formulated as a **Reinforcement Learning** (**RL**) problem. The case when agent's action space is the same as the search space is essentially a multi-armed bandit problem. At each iteration agent generates an architecture and receives a reward based on the estimated architecture performance. Another option is to formulate NAS as a sequential decision process in which agent samples architecture sequentially, constructing it from building blocks such as layers. However, there is no interaction with the environment during this process and the reward is obtained only after the last action.

Zoph and Le [34] use a **Recurrent Neural Network** (**RNN**) controller to sequentially sample a string that encodes the architecture. The controller is auto-regressive: hyperparameters are predicted one at a time based on the previous network outputs, so that it can make the most suitable addition to the architecture generated so far. RNN stops after the predefined number of iterations and undergoes a simple post-processing step in case if the generated graph is invalid. Resulting network is trained for a reduced amount of epochs and its performance on the validation set is used as the reward signal to the controller and its parameters are updated with REINFORCE method. RNN controller is powerful and can generate architectures of variable length, which makes it suitable for discovering deep architectures. This method has shown good empirical performance, however, it requires a few GPU days to reach convergence due to the inefficient performance estimation strategy.

Efficient NAS (ENAS) [24] uses weight sharing among generated models in order to reduce the search time significantly. The main idea is that the search space can be viewed as a supergraph (also referred to as parent model or

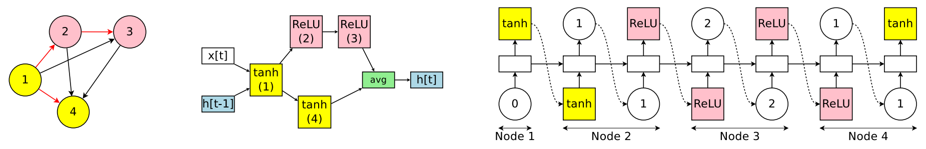

Figure 2.3: An example of a recurrent cell in our search space with 4 computational nodes. *Left*: The computational DAG that corresponds to the recurrent cell. The red edges represent the flow of information in the graph. *Middle*: The recurrent cell. *Right*: The outputs of the controller RNN that result in the cell in the middle and the DAG on the left. Note that nodes 3 and 4 are never sampled by the RNN, so their results are averaged and are treated as the cell's output. Source: [24]

hypernetwork), and all discovered architectures are its subgraphs. In ENAS, search discovers cells which are then stacked to get the deep network, so this is the case of discovering micro-architecture while the macro-architecture is fixed. The RNN controller samples N decision blocks, each one specifying the hyperparameters of the DAG node (e.g. activation function) as well as with which edges to take as an input. Each edge is associated with the set of network parameters and since the edges come from a common supegraph, the weights are shared across sampled models. This way, the results of the training phase are not discarded and used in the further search. While preserving the performance of the RNN controller, this method speeds up the search process by 1000x in terms of GPU-hours compared to previous RL-based method. Performance estimation technique where all models share their parent's weights is referred to as *One-shot NAS*.

Differentiable Architecture Search

Differentiable Search uses continuous search space representation to enable gradient-based optimization of both network weights and architecture. Architecture is represented by a computational supergraph similarly to the ENAS search space. In differentiable search methods, each node of the parent graph is parameterized by the set of coefficients, so that the output of the node can be computed as $f(x; w) = \sum_{i=1}^{m} \alpha_i o_i(x; w), \alpha \geq 0, \sum_{i=1}^{m} \alpha_i = 1$, where α denotes architecture coefficients, w are the models weights and $o \in \mathcal{O}$ are possible operations to choose from, e.g. convolution. Network weights and architecture parameters are optimized jointly using backpropagation to compute the gradients as in standard Deep Learning. Let $\mathcal{L}_{val}$ and $\mathcal{L}_{train}$ denote validation and training loss, respectively. The problem which is solved by differentiable search methods can be formulated as:

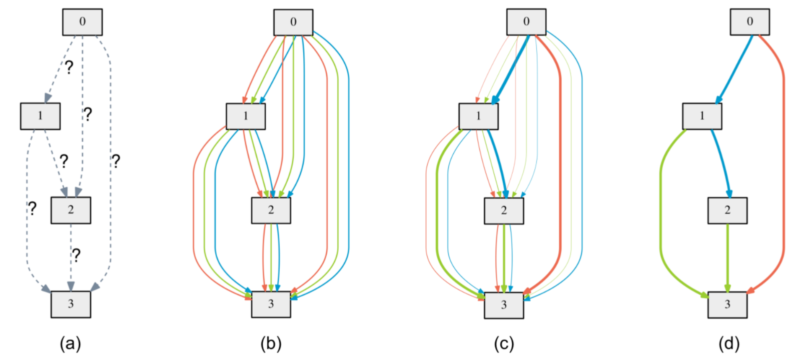

Figure 2.4: An overview of DARTS: (a) Operations on the edges are initially unknown. (b) Continuous relaxation of the search space by placing a mixture of candidate operations on each edge. (c) Joint optimization of the mixing probabilities and the network weights by solving a bilevel optimization problem. (d) Inducing the final architecture from the learned mixing probabilities. Source: [19]

$$\begin{aligned} \min_{\alpha} \quad & \mathcal{L}_{val}(w^*(\alpha), \alpha) \\ \text{s.t.} \quad & w^*(\alpha) = \arg\min_{w} \mathcal{L}_{train}(w, \alpha) \end{aligned} \tag{2.4}$$

To avoid overfitting the architecture, it is common that weights and parameters are optimized on the training and validation datasets, respectively. Final architecture can be extracted e.g. by choosing the operations with maximized architecture weights: $o^* = \arg\max_{o_i} \alpha_i$. Resulting network has to be retrained from scratch since the optimization procedure essentially learns to *rank* candidate architectures. Performance on the validation set during training does not directly correspond to the real performance of the network, however, these quantities are strongly correlated.

Differentiable Architecture Search (DARTS) [19] uses a simple approximation scheme to avoid nested optimization in (2.4). Optimization alternates between applying gradient descent update to the architecture parameters α and weights w. Convex combination of the operators is formulated with softmax transformation of α. DARTS have managed to achieve competitive results while using substantially less computational resources compared to RL-based NAS methods. Convolutional cells learned on CIFAR-10 dataset were also transferable to the ImageNet problem.

Although DARTS has shown good performance, it has been argued that it introduces bias to the search due to the fast convergence of some child models. Shallow or smaller subgraphs can converge quickly which leads the their exploitation and harms the exploration of other architectures. Self-Evaluated Template Network (SETN) [6] algorithm addresses this problem by treating learned architecture parameters as discrete probability distributions over graph edges. Instead of selecting a single final architecture after convergence, it samples N candidate architectures according to the probability distribution learned during the optimization and evaluates them to select the best one.

Stochastic NAS (SNAS) [31] uses a blend of stochastic and differentiable search strategies. Operation choices are encoded as categorical random variables. Choosing the architecture is then equivalent to sampling from the joint distribution of all operation choices, similar to the ENAS algorithm. Distributions of these variables are learned during the fully differentiable search procedure. As in DARTS, these choices are embedded directly into the computation graph of the search space. In order to make samples from the categorical distribution differentiable with respect to the class probabilities, the algorithm employs the *Gumbel-Softmax* trick[14]. It also uses a scheduled temperature parameter to control the exploration-exploitation trade-off during the learning procedure. The parameterization of the search space is similar to the DARTS version:

$$\begin{aligned} f(x; w) &= \sum_{i=1}^{m} \alpha_i o_i(x; w) \\ \alpha_i &= \frac{\exp((\log(\pi_i) + g_i)/\tau)}{\sum_{j=1}^{m} \exp((\log(\pi_j) + g_j)/\tau)} \end{aligned} \tag{2.5}$$

where π_i are parameters of the categorical distribution, g_i are i.i.d. samples drawn from $\text{Gumbel}(0, 1)$ and $\tau > 0$ is the temperature parameter. For low temperature values, softmax output is close to the true one-hot samples from the categorical distribution with class probabilities π. SNAS eliminates the bi-level optimization problem of DARTS since the choices are sampled directly during the forward pass. Architecture probabilities are adjusted during the backward pass without the need of any additional mechanisms such as policy gradient updates. This procedure is shown to be equivalent to sampling candidate architectures from the hypernetwork-induced search space with RNN as in ENAS.

Analysis of the One-Shot approach suggests that obtaining promising architectures does not require optimization of hypernetworks or RL. Search procedure described in [2] trains a parent network where all possible output paths in the cell are aggregated and randomly samples candidate architectures after convergence. Most promising architectures (having highest performance on validation set) are trained and evaluated from scratch. To stabilize training and avoid co-adaptation of the parameters, authors use *path dropout* to randomly "turn off" parts of the graph. Experimental results suggest that network learns to optimize weights on the most promising operations and ignore the rest of them, even without additional architecture parameterization.

Chapter 3

Methodology

This chapter describes the process of applying Neural Architecture Search methods to find MIL network architectures. We implement several state of the art search methods and design a search space for MIL architectures.

3.1 Search Methods

To study performance and efficiency of NAS techniques we select Reinforcement Learning based method, Differential Architecture Search method and the Stochastic Search, referred to as ENAS, DARTS and SNAS, respectively. These methods are selected for the reason that at their core they rely only on the gradient descent optimization, which is in line with the end-to-end learning paradigm of Deep Learning. Although ENAS, DARTS and SNAS are closely related in theory[31], in practice they can produce significantly different results. To control the effect of these methods, we use Random Search algorithm as a baseline. All of these methods can be seen as a form of the guided (unguided in the case of the Random Search) architecture sampling.

3.1.1 Random Search

Random Search has proven to be a strong baseline method for hyperparameter optimization and architecture search[17]. The main advantages of the algorithm are simplicity and good exploration properties. However, in the case of neural networks it is inefficient, since every sampled architecture should be trained from scratch which is a time-consuming process. We can address that by using a smarter performance evaluation strategy without compromising the simplicity of the algorithm. Firstly, training until convergence might be excessive to rank architectures. It should be sufficient to train the network for a small fixed amount of epochs and employ early stopping, which cuts off the learning process when the loss stops decreasing on a validation data set reserved specifically for this purpose. Second option is to use weight sharing similarly to ENAS and other One-Shot search strategies. While it introduces some bias to the search, it greatly reduces computational costs. We use both of these options and refer to them as Random Search (RS) and Random Search with Weight Sharing (RS+WS), respectively.

Algorithm 1: Random Search

```
Input: Number of iterations $n$, Search space $\mathcal{S}$, performance
       evaluation function Eval
Output: Neural Network Architecture $\alpha$
$p \leftarrow -\infty$;
$\alpha \leftarrow \emptyset$;
for $i \leftarrow 1$ to $n$ do
    $\alpha_i \leftarrow$ SampleUniform ($\mathcal{S}$);
    $p_i \leftarrow$ Eval ($\mathcal{S}$, $\alpha_i$);
    if $p_i > p$ then
        $p \leftarrow p_i$;
        $\alpha \leftarrow \alpha_i$;
    end
end
```

3.1.2 Reinforcement Learning

For the Reinforcement Learning approach, we adapt the ENAS[24] algorithm. Original paper does not include all the details of the training process, so our version is not an exact reproduction of it, but an adaptation. In the Reinforcement Learning NAS formulation, state space and action space are identical and consist of successive architecture choices. Actions are produced by the controller. Controller is a policy network build from an encoder, which embeds individual actions into the vector space, a recurrent unit which processes an action embedding and a set of decoders, one for each action.

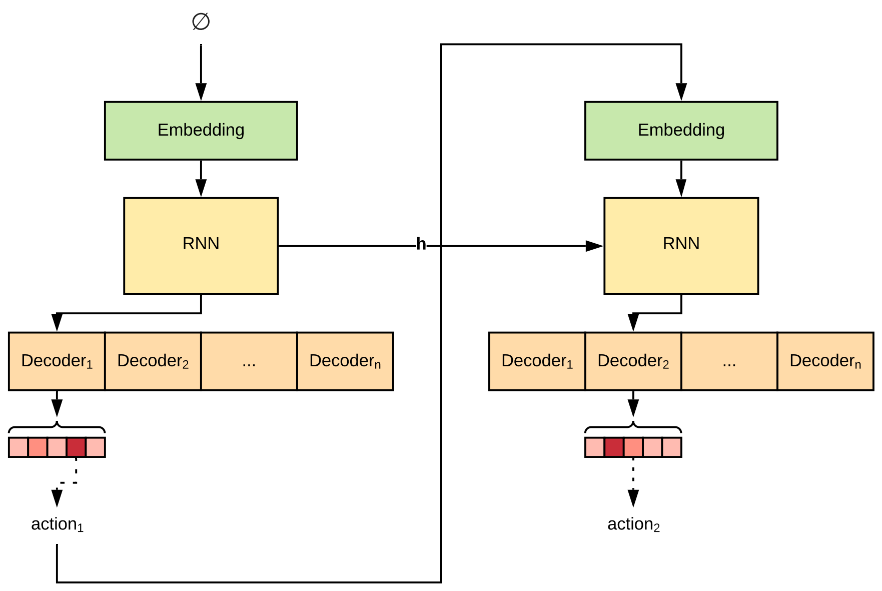

Figure 3.1: A step of a controller network unrolled through time. Dotted lines represent sampling from a set of candidate actions according to the probabilities estimated by the network.

To produce an action, the output of the recurrent network is fed into the corresponding decoder which produces a vector of action probabilities. Action is then sampled from a categorical variable distributed according to these probabilities. This process is repeated in an autoregressive fashion, each sampled architecture choice is fed into the embedding layer on the next step, until all choices are sampled and the architecture is complete. Sampled architecture is trained for a fixed amount of epochs on the training set and evaluated on the validation set. Performance measured on a validation set is used as a feedback signal to the controller network. To train the controller with backpropagation, REINFORCE[29] policy gradient rule is used. This process is equal to a single controller epoch and can be repeated for a fixed number of times or until the convergence. All sampled architectures are assumed to be a part of the hypernetwork and share trained weights across controller epochs.
As in the original paper, we use REINFORCE rule with a reward moving average baseline to reduce the variance of the gradient estimate. We add entropy regularization to the REINFORCE loss which penalizes over-confident decisions of the policy network. It helps to prevent premature convergence to a local minimum and encourage exploration. We use a single LSTM cell

as a recurrent network and a single fully-connected layer per each decoder. Hidden dimension size is kept across embedding, RNN and decoder input. Size of the decoder's output is determined by the number of choice options.

Algorithm 2: ENAS

```
Input: Number of epochs $n$, Search space $\mathcal{S}$, performance evaluation
       function Eval, learning rate $\eta$
Output: Trained Controller network $\pi_\theta$
Initialize the controller network $\pi_\theta$ using $\mathcal{S}$;
for $i \leftarrow 1$ to $n$ do
    // Sample architecture probabilities
    $\pi_i \leftarrow$ SampleController ($\pi_\theta$);
    // Sample architecture choices
    $\alpha_i \leftarrow$ WeightedSample ($\mathcal{S}$, $\pi_i$);
    $R_i \leftarrow$ Eval ($\mathcal{S}$, $\alpha_i$);
    // Update the Controller using REINFORCE rule
    $\theta \leftarrow \theta + \eta R_i \nabla_\theta \ln \pi_\theta(\alpha_i)$;
end
```

3.1.3 Differentiable Search

We adapt two differentiable search methods: DARTS[19] and SNAS[31]. In these methods, each choice is represented as a parameterized mixture of candidate operations. The difference is that in DARTS the coefficients of the combination are computed in a purely analytical way, while in SNAS the coefficients approximate one-hot samples from the categorical distribution, which is closer to sampling a single operation from the candidate set. Another difference is the optimization scheme. To enforce architecture generalization, DARTS alternates between optimizing the weights of the model and the architecture parameters on different data sets. SNAS optimizes both the model and architecture parameters in a single pass. Weights of candidate models are naturally shared since they constitute one large hyper-network. DARTS and SNAS at high temperature values are prone to the effect of weight co-adaptation, which is caused by operations in a mixture getting dependent on each other which may lead to overestimating candidates scores. To combat this effect, we can select a single operation by introducing the argmax function into the forward pass of the mixture. In some works this modification of DARTS algorithm is referred to as the Single-Path NAS[26]. Similar trick for SNAS is the Straight-Through Gumbel-Softmax estimator[14]. We refer to these variations as SP-DARTS and ST-SNAS in the experiments.

Algorithm 3: DARTS

```
Input: Number of iterations $n$, Architecture parameters $\alpha$, Network
       parameters $w$, Loss function $\mathcal{L}$, learning rate $\eta$
Output: Neural Network Architecture $\alpha$
for $i \leftarrow 1$ to $n$ do
    $\alpha \leftarrow \alpha - \eta \nabla_\alpha \mathcal{L}(w, \alpha)$;
    $w \leftarrow w - \eta \nabla_w \mathcal{L}(w, \alpha)$;
end
```

Algorithm 4: SNAS

```
Input: Number of iterations $n$, Architecture probabilities $\pi$, Network
       parameters $w$, Loss function $\mathcal{L}$, learning rate $\eta$
Output: Neural Network Architecture $\alpha$
for $i \leftarrow 1$ to $n$ do
    $(w, \pi) \leftarrow (w, \pi) - \eta \nabla_{w,\pi} \mathcal{L}(w, \pi)$;
end
```

3.1.4 Evaluation Methodology

In order to compare different search methods, a properly crafted evaluation procedure is required. Due to the stochastic nature of most search algorithms, we evaluate them through architecture sampling. Some of the methods result in ready to use models, while others require post hoc training of the model parameters.

We evaluate search methods through the following procedure. After the training is finished, K architectures are sampled from the resulting model. Architecture sampling is natural for stochastic methods like ENAS and SNAS. For DARTS, we treat learned architecture weights as parameters of categorical distribution and draw i.i.d. samples at each choice node according to them. In Random Search, K samples are drawn from N top performing architectures discovered during the search. Sampled architectures are then trained from scratch on the same data set that was used for search. Finally, resulting models are evaluated on a held-out test set. This approach enables us to estimate the expected performance of models produced by the search method. It also gives us the information about performance distribution and variance.

3.2 Adapting NAS to MILNNs

In order to use NAS algorithms in MIL architecture search, the most important step is to develop a suitable search space. Another thing to consider is that current NAS methods are designed with Convolutional and Recurrent architectures in mind, as well as with imaging and textual data sets, respectively. In MIL, data sets and learning dynamics may differ from conventional ones which might affect the behavior of the search procedures.

3.2.1 Search Space Design

Arguably, search space design is more important than the search method because it defines which architectures can be discovered in principle. When defining the search space we introduce potentially promising operations and structures, while disregarding ones that are not useful at the risk of introducing bias to the search procedure. However, a completely unbiased search space is impossible to achieve since it would not be practical. The search space has to be reasonably constrained so that the search algorithm can explore it in a practical time budget. Another thing to consider is that for the different search methods to be comparable, search space should be agnostic to the algorithm using it. It means that conceptually different algorithms should be equally exposed to the search space and be able to find any architecture it contains.

In order to define the search space for MIL networks, we separate the architecture into units which constitute MIL blocks. Overall architecture can consist of one ore more blocks combined in parallel or serial fashion. Parallel combination allows networks to process hierarchically structured inputs, while serial combination can be employed to build deep MIL networks. MIL block consists of an encoder network, an aggregation function and a decoder network, corresponding to the formulation in the equation 2.1. Encoder and decoder units can have an arbitrary search space depending on the task being solved. As an example, when solving an image processing task, encoder architecture would most likely be a Convolutional Network. Aggregation operations are chosen from two groups of options. One group consists of standard pooling functions, such as sum, mean, etc. The second group is attention-based pooling functions. The dimensionality of the search space of one block is $\dim(\text{block}) = \dim(\text{encoder}) \times (|\mathcal{O}_p| + |\mathcal{O}_a|) \times \dim(\text{decoder})$, where $\mathcal{O}_p$ and $\mathcal{O}_a$ are sets of pooling and attention operations, respectively.

Following table summarizes all aggregation functions used in the experiments.

Pooling	Formula
sum	$p = \sum_{i=1}^{k} \phi(\mathbf{x_i})$
mean	$p = \frac{1}{k} \sum_{i=1}^{k} \phi(\mathbf{x_i})$
elementwise max	$p = [\max\{\phi(x_{ij}) \mid i = 1 \dots k\} \mid j = 1 \dots m]$

Table 3.1: Pooling functions

Following table lists all candidate attention functions used in the experiments. Variations denoted as mlp and gated are introduced by [11]. Linear attention is a simple dot product between the encoded input instance and the learnable "context vector".

Attention	Formula
linear	$s_i = \mathbf{w}^T \phi(\mathbf{x_i})$
mlp	$s_i = \mathbf{w}^T \tanh(\mathbf{V}\phi(\mathbf{x_i}))$
gated	$s_i = \mathbf{w}^T (\tanh(\mathbf{V}\phi(\mathbf{x_i})) \odot \mathrm{sigm}(\mathbf{U}\phi(\mathbf{x_i})))$

Table 3.2: Attention functions

Attention coefficients are obtained with a softmax function over all scores in a bag:

$$a_i = \frac{\exp(s_i)}{\sum_{j=1}^{b} \exp(s_i)} \tag{3.1}$$

Search space of the permutation-equivariant MIL layers differs from that of the permutation-invariant layers in that it is more restricted. The only variable parameters of the layer are an aggregation function and an activation function. We adopt a simplified version of the layer which consists of a single matrix multiplication:

$$f(\mathbf{x}) = \sigma(\beta + (\mathbf{x} - g(\mathbf{x}))\Gamma) \tag{3.2}$$

Choices for the aggregation operation g are the same as in the permutation-invariant case.

It has been shown that MIL architectures can benefit from so-called residual connections[28] which are widely adopted e.g. in convolutional architectures.

Thus it would be reasonable to incorporate the possibility of such connections into the search space. We add a bag-level skip connection choice into the search space of both MIL layers. Sum of multiple aggregation unit outputs is passed as an input to the decoder unit. For each potential residual connection, a search algorithm chooses between two candidate operations: an identity $f(\mathbf{x}) = \mathbf{x}$ or a zero operation $f(\mathbf{x}) = \mathbf{0}$, which correspond to the inclusion and exclusion of the residual connection, respectively. It is also possible to add connection choices into encoder and decoder networks if it is reasonable for the task being solved. Residual connections do not violate theoretical properties of MIL networks.

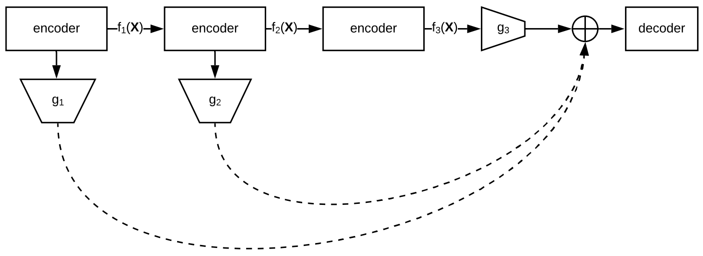

Figure 3.2: An illustration of the residual connections included into the network architecture. Optional residual connections are depicted with dashed lines.

Individual MIL blocks can be combined into complex networks. Nested permutation-invariant blocks allow processing of tree-structured data. In this case blocks at the inner level share search space and model parameters. Permutation-equivariant blocks can be stacked sequentially and retain their equivariance property. It is possible to add residual connections to these networks in a similar way as with permutation-invariant networks.

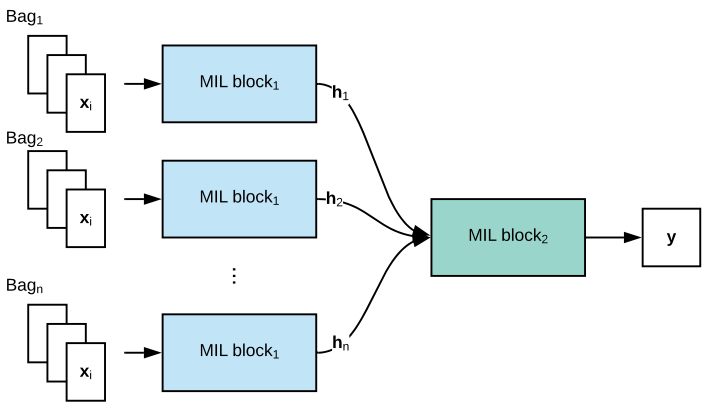

Figure 3.3: Two permutation-invariant MIL blocks nested in order to process hierarchical data structures. Identical blocks have the same color.

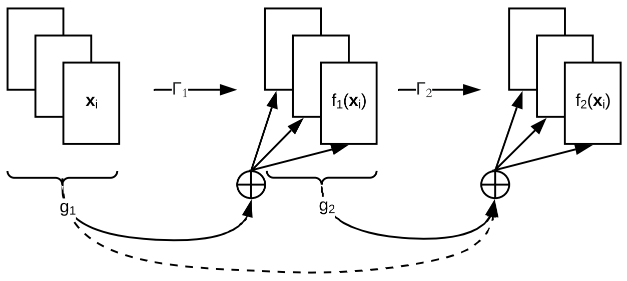

Figure 3.4: Two sequentially stacked permutation-equivariant MIL blocks constituting a deep MIL network. Residual connections are depicted with dashed lines.

In general, state of the art NAS methods cannot operate with hierarchical search spaces, where some parts of space are dependent on the choice of a certain architecture parameter. This problem can arise for example when working with fully-connected networks, which are still wide-spread in MIL applications. Each consecutive layer of the fully connected network is dependent on the output size of the previous layer. Another problem with these types of search spaces is that weight sharing among heterogeneous architectures is impossible, which significantly increases memory demands of the search procedure. For these reasons we do not optimize hidden dimension sizes of the fully connected layers and keep them fixed. Instead, we focus on the search of the best combination of activation and aggregation functions, as well as the optimal connectivity structure between the operations.
Due to the fact that encoder and decoder units of the permutation-invariant MIL block are largely domain-dependent, it is also impossible to discover a transferrable MIL block suitable for a broad range of tasks.

3.2.2 Practical Considerations

Training of the search methods requires a lot of tricks and subtle adjustments which may seen unimportant, however they can be crucial to convergence of the learning procedure. Some papers describing the search algorithms do not come with source code and an exhaustive list of all parameters and details of the search, which makes them hard to reproduce. We use several techniques to stabilize the training and improve search convergence. They are not directly related to any algorithm but are nonetheless important.
We found the batch size of 1 to be enough in MIL problems tested in our setup. This is equivalent to processing one bag per iteration of the gradient descent. This also enables usage of Batch Normalization layers[13], since a single bag can be seen as a batch of instances. Our hypothesis is that batch

normalization stabilizes training in a setup with combinations of multiple activation functions in the search space.
A special regularization scheme is employed to decay the weights that are being currently active and avoid unnecessary decrease of the weights that are "turned off" by the search. This effect arises in Random Search with Weight Sharing and ENAS while using some kind of weight decay. Since it is part of the loss function, all weights are penalized at each iteration, even though just a small part of them is being active at the current step. This can lead to poor performance of network paths that are chosen rarely, which results in accumulating "winner-takes-all" effect.
A similar effect can be observed when a candidate set contains operations that are not trainable or have few parameters and heavily parameterized operations at the same time. It is the case for aggregation operation choice which includes parameterized attention and plain pooling functions. Operations that perform reasonably from the start or take a few iterations to do so accumulate higher architecture parameters, while other operations do not get enough gradient updates and can not perform at their full power. One possible solution to this problem is to use a number of iterations at the beginning of the search as a warm-up for parameterized operations and keep parameter-less operations disabled for this time. Another option is to regularize architecture parameters. We can use the KL divergence between the architecture distribution and a uniform distribution as a regularizer and anneal the strength of that term as the number of iterations increases to assist convergence. The modified loss is formulated as follows:

$$\mathcal{L}(\mathcal{T};\alpha,\theta) = \mathcal{L}(\mathcal{T};\theta) + \xi \sum_{\alpha\in\mathcal{S}} D_{\mathrm{KL}}(A(\alpha)\|Q) \tag{3.3}$$

where A is a categorical distribution induced by the architecture parameters $\alpha \in \mathcal{S}$, Q is a discrete uniform distribution and ξ is a regularization strength coefficient. We compare both of these options in a separate experiment.

3.3 Implementation

We implement a minimalistic library for Neural Architecture Search in the Julia programming language[4]. It is built on top of the Flux.jl[12] framework, which provides a broad range of functionality for Machine Learning and Differentiable Programming. Our library contains a set of abstractions and primitives for creating search spaces, as well as implementations of selected search strategies. Search spaces provide a general interface that can be used

by a wide range of search algorithms.
All the code, including experiments, is written in the Julia language. It is a general-purpose language designed with scientific programming in mind. Julia's type system allows first-class support of Automatic Differentiation at compile-time, making it an excellent choice for Deep Learning paradigm. However, it is worth noting that as of today, Julia's Machine Learning ecosystem is still in development and is far from maturity.
Appendix A contains a detailed description of the implementation as well as the instructions on reproducing experimental results.

Chapter 4

Experimental Results

In this chapter we demonstrate the conducted experiments. In the experiments we compare selected search methods on several benchmark MIL problems, and analyze the effects of their modified variations. We then discuss the results and propose a search method best suited for MIL networks based on the experimental results.

4.1 Comparison of Search Methods

4.1.1 Experimental Setup

The goal of the experiment is to compare different NAS methods in terms of performance and efficiency on MIL problems. We compare Random Search, DARTS, SNAS, ENAS adapted to the MIL setting as described in the previous section. When available, we add a published human-designed model to the comparison.
We use the following metrics for the comparison:

- Task-specific performance metric (e.g. classification accuracy)
- Training time (relative to the Random Search)

To make the comparison between the search algorithms more general, we evaluate them on a diverse set of problems:

Dataset	Task	Inputs
Musk[7]	classification	real vectors
MNIST-bags	regression	images
IoT	classification	hierarchical features
ModelNet40[30]	classification	point clouds

Table 4.1: Benchmark problems

Although majority of the datasets concern with solving a classification task, each of them provides a unique input structure which requires distinct search spaces.

4.1.2 Musk

This dataset describes a set of 92 molecules classified by humans as musks or non-musks. The goal is to learn to predict whether new molecules will be musks or not. Each molecule is represented by a bag of it's possible low-energy spatial conformations, which are obtained by rotating bonds of the molecule. If one of these conformations is marked as musk, then the molecule is considered to be a musk.

Search space for this task contains a single MIL block. The encoder consists of three fully connected layers with 128 neurons per each one, last layer is followed by a dropout operation with probability 0.5. Their activation functions σ_i are chosen from $\{\tanh, \mathrm{ReLU}, \mathrm{ELU}[5]\}$. There are three aggregation function choices, all of them are connected to the output classifier by optional residual connections. Since the dataset is quite small to properly learn the attention pooling, aggregation functions are chosen from the set of basic pooling operators $\mathcal{O}_p$ only. The dimensionality of that space is $3^3 \times 3^3 \times 2^3 = 5832$. The search space is inspired by the basic MI-Net model[28], which is also used as a human-designed baseline.

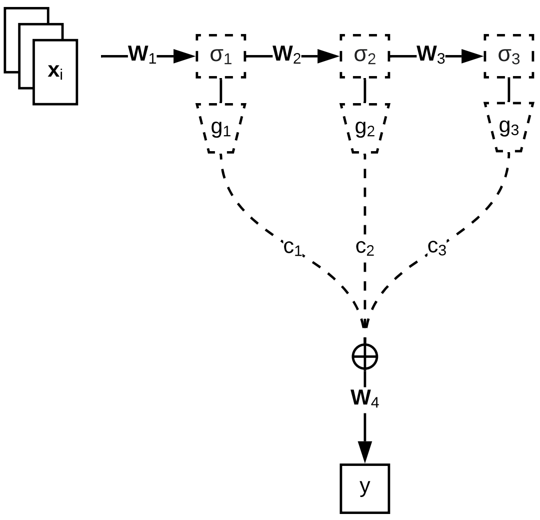

Figure 4.1: Search space for the Musk classification task. Solid lines denote fixed operations. Architecture choices are denoted with dashed lines. Activation and aggregation function choices are represented by σ_i and g_i nodes respectively. $\mathbf{W}_i$ denotes a fully-connected layer.

MI-Net[28] and all architectures sampled for the evaluation procedure are trained for 10 epochs with batch size equal to one using SGD with Nesterov momentum[21] and L2 weight regularization.
Random Search and Random Search with Weight Sharing were run for 100 iterations. Evaluation strategy is to train the sampled model and validate on a small held-out portion of the training set. Training is done for 1 epoch for Random Search with Weight Sharing and 10 epochs for vanilla Random Search.
ENAS was run for 50 epochs. Controller network with hidden dimension 100 was trained with an ADAM[16] optimizer with learning rate $\eta = 10^{-4}$. Evaluation strategy is to train the sampled model for 1 epoch and validate on a small held-out portion of the training set.
DARTS was run for 25 epochs, SGD with learning rate $\eta = 5 \cdot 10^{-6}$ was used to optimizer network weights and ADAM with learning rate $\eta = 5 \cdot 10^{-5}$ was used to optimize architecture parameters.
SNAS was run for 25 epochs with an ADAM optimizer and a learning rate $\eta = 50^{-5}$.
Due to the small dataset size, training procedure has shown to be sensitive to the train-test split. To combat this effect, we evaluate all methods using 10-fold cross-validation and list averaged results along with standard deviation values.

Results are summarized in the following tables:

Method	Accuracy	Relative training time
MI-Net	0.88 ± 0.097	-
RS	0.88 ± 0.18	100%
RS+WS	0.88 ± 0.14	13.1%
ENAS	0.9 ± 0.11	3.1%
DARTS	0.89 ± 0.11	**0.08%**
SNAS	0.89 ± 0.1	0.29%

Table 4.2: Accuracy and training time comparison for Musk dataset.

Choice	chosen value
σ_1	tanh
σ_1	tanh
σ_3	ELU
g_1	max
g_2	max
g_3	mean
c_1	enable
c_2	disable
c_3	enable

Table 4.3: An example of the architecture found by DARTS for Musk dataset.

In general we observe that most sampled architectures include residual connections and tend to mix different aggregation functions. Furthermore, ELU activations are preferred over ReLUs. However, this is merely an observation, since the search space seems to consist mostly of local optima. Slightly modified resulting architectures sometimes can reach better performance than original ones. There is no clear winner in terms of accuracy among search methods.

4.1.3 Sum of MNIST Digits

We create a dataset for the task of estimating the sum of digits in a bag of 28×28 digit images originally obtained from MNIST dataset. Bags are formed by random samples of the original images. Each training bag consists of up to 10 images and the size of testing bags is limited by 50.

Search space for this task contains a single MIL block. Encoder is a

Convolutional Neural Network which plays the role of the feature extractor. Encoder network consists of 5 operations which are connected arbitrary forming a directed acyclic graph. This is achieved by optional residual connections: j-th operation can take a sum of outputs of the nodes from 1 to $j-1$ as an input. Operations are chosen from the set of options $\{\text{identity}, \text{conv 3x3}, \text{conv 5x5}, \text{max pool}, \text{avg pool}\}$. Non-linearity is fixed to the ReLU function. Number of channels and spatial dimensions are preserved across the operations to make them compatible. The last node is followed by a fully-connected layer and a flattening operation to produce a one-dimensional vector. Bag of these representations is aggregated by a pooling function chosen from $\mathcal{O}_p \cup \mathcal{O}_a$. Finally, the last fully-connected layer produces an estimate of the bag sum. Dimensionality of the search space is $5^5 \times 6^5 \times 2^{10} = 2.48 \cdot 10^{10}$.

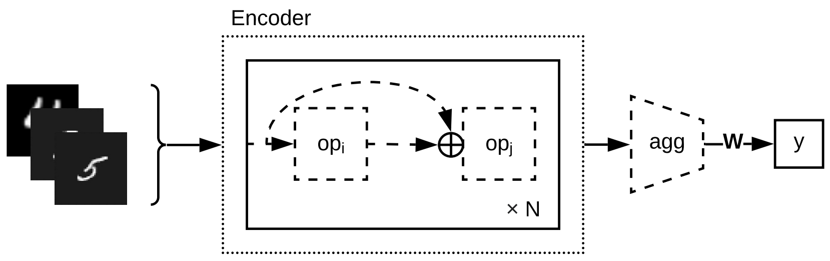

Figure 4.2: An illustration of the search space for the Sum of MNIST-bags task. Encoder is build by repeating the pattern of the layer fed in with a sum of residual connections from the previous layers. Dashed lines and containers represent architecture choices.

Random Search and Random Search with Weight Sharing were run for 5000 iterations. Evaluation strategy is to train the sampled model and validate on a small held-out portion of the training set. Training is done for a 1000 *iterations* for Random Search with Weight Sharing and 1 epoch for vanilla Random Search. Low number of iterations in the evaluation phase indented to make random search methods feasible.
ENAS was run for 200 epochs. Remaining setup is the same as in the Musk experiment.
DARTS was run for 200 epochs, SGD with learning rate $\eta = 10^{-4}$ was used to optimizer network weights and ADAM with learning rate $\eta = 10^{-5}$ was used to optimize architecture parameters.
SNAS was run for 200 epochs with an ADAM optimizer and a learning rate $\eta = 50^{-5}$.
All architectures sampled for the evaluation procedure are trained for 100 epochs with batch size equal to one using ADAM optimizer and L2 weight regularization. We sample 10 architectures and select the best-performing one to measure performance of the method.

We use accuracy of the rounded predictions as the evaluation metric since it is more intuitive than MAE loss.

Results are summarized in the following table:

Method	Accuracy	Relative training time
RS	0.82	100%
RS+WS	0.80	14.5%
ENAS	0.86	5.1%
DARTS	0.87	**0.55%**
SNAS	0.89	0.89%

Table 4.4: Accuracy and training time comparison for MNIST-bags dataset.

Sampled architectures employ residual connections and mostly use simple sum pooling, which might be the most suitable choice in the task of estimating the sum of bag labels. Contrary to the Musk experiment, advanced search methods were able to outperform the Random Search by a significant margin. This may be the sign that the proposed search space is more complex and contains a lesser number of effective candidate architectures.

4.1.4 ModelNet40

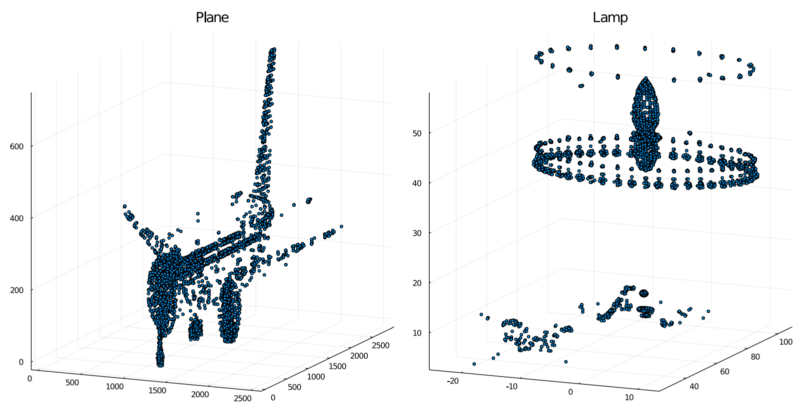

Figure 4.3: Examples of point clouds generated for the task.

ModelNet40[30] dataset provides CAD models from the 40 categories. It can be formulated as a point cloud classification task, which can be seen as a MIL task, since point clouds are just sets of real vectors. To do so, we randomly sample 5000 points in the $\mathbb{R}^3$ space from each model. Note that a lot of structural information about the faces of models is discarded and the resulting representation of the object is very compact. Permutation-equivariant model used for the same task in DeepSets[33] is used as a baseline. In DeepSets, point clouds used for training undergo some data augmentation techniques such as random rotations and scaling. We do not employ these techniques to avoid them contributing to the higher accuracy score and focus only on the architecture instead.

Search space for this task consists of three permutation-equivariant layers followed by bag aggregation and a 40-way fully-connected layer to predict the categories. Number of channels at all layers is set to 512. Each of the layers contains a choice of the activation and aggregation function. Search space also includes optional residual connections between aggregation units. Activation functions are chosen from the set $\{\tanh, \mathrm{ReLU}, \mathrm{ELU}\}$, and aggregation functions are chosen from $\mathcal{O}_p \cup \mathcal{O}_a$. The dimensionality of the search space is $3^3 \times 6^3 \times 2^2 \times 6 = 1.4 \cdot 10^5$.

Random Search and Random Search with Weight Sharing were run for 2500 iterations. Evaluation strategy is to train the sampled model and validate on a small held-out portion of the training set. Training for model evaluation is done for 1 epoch in RS and for 1000 iterations in RS+WS.
ENAS, DARTS and SNAS were all run for 150 epochs. Remaining setup is the same as in the previous experiment.
All architectures sampled for the evaluation procedure are trained for 50 epochs with batch size equal to 4 using Adam optimizer and L2 weight regularization. As in the previous setup, we report the best accuracy among 10 sampled architectures.

Results are summarized in the following table:

Method	Accuracy	Relative training time
DeepSets + transformation	0.9 ± 0.3	-
RS	0.87	100%
RS+WS	0.88	16.8%
ENAS	0.91	5.1%
DARTS	0.89	**0.12%**
SNAS	0.9	0.48%

Table 4.5: Accuracy and training time comparison for Modelnet40 dataset.

Again, random search methods are on par with advanced search techniques. We observe that sampled architectures tend to utilize simple pooling functions such as max in intermediate layers, whereas attention pooling seems to be the best option for the last aggregation which precedes the classifier. Residual connections have negligible impact on the performance of the permutation-equivariant architecture.

4.1.5 IoT

The dataset describes network scans of Internet-of-Things devices from 13 classes. Each scan contains a set of open ports and some other information, e.g. the list of registered DNS services on the device. A simple MIL task for this dataset would be to classify the device according to the network scan. To make this a hierarchical MIL task, we group scans into bags and label these second-level bags according to the presence of the IP security cameras turning the task into the binary classification of hierarchical data.

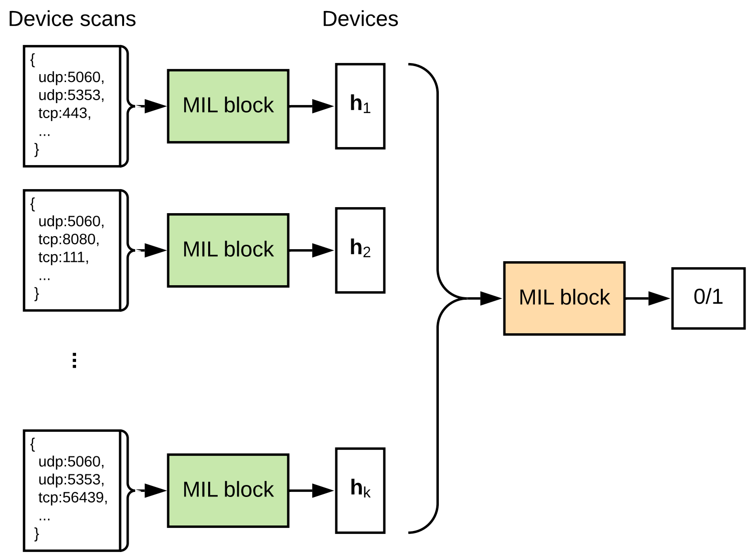
Device scans
Devices
{ udp:5060, udp:5353, tcp:443, ... }
{ udp:5060, tcp:8080, tcp:111, ... }
{ udp:5060, udp:5353, tcp:56439, ... }
MIL block
MIL block
MIL block
h_1
h_2
h_k
MIL block
0/1

Figure 4.4: Schematic representation of the hierarchical model for the IoT task.

Search space for this task contains two MIL blocks, inner and outer one. Inner block operates on the level of individual device scans, and outer block outputs a prediction based on outputs of the inner block. Open ports are embedded into 64-dimensional real vector space and are process by a DAG of linear layers and non-linear activation functions. Non-linearities are chosen from the set $\{\text{identity}, \tanh, \text{ReLU}, \text{ELU}[5]\}$. For each of the activations, an

input is chosen from one of the outputs of the previous activation nodes. All outputs are tied up with aggregation functions which are connected to the block output by optional paths.
Outer block of the search space consists only of the aggregation function and a final fully-connected layer to produce the prediction. The dimensionality of such search space is $4^3 \times 4 \times 6^3 \times 2^3 \times 6 = 2.6 \cdot 10^6$.

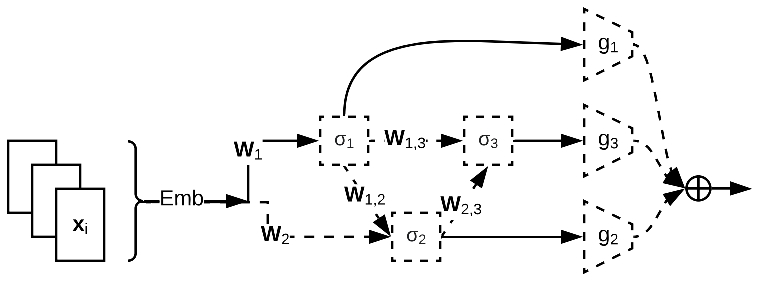

Figure 4.5: Search space for the inner MIL block. Solid lines represent fixed connections, dashed lines represent connection choices. Choice nodes for activation and aggregation operations are denoted with σ_i and g_i, respectively.

Random Search and Random Search with Weight Sharing were run for 2500 iterations. Evaluation strategy is the same as in the previous experiment. ENAS, DARTS and SNAS were all run for 250 epochs. Remaining setup is the same as in the previous experiment.
All architectures sampled for the evaluation procedure are trained for 100 epochs with batch size equal to 1 using SGD with the learning rate $\eta = 10^{-6}$ optimizer and L2 weight regularization. We report the best accuracy among 10 sampled architectures.

Results are summarized in the following table:

Method	Accuracy	Relative training time
RS	0.82	100%
RS+WS	0.83	17.2%
ENAS	0.82	7.4%
DARTS	0.84	**0.22%**
SNAS	0.86	0.89%

Table 4.6: Accuracy and training time comparison for IoT dataset.

Similarly to the most conducted experiments, guided search methods outperform Random Search only by a small margin. In this task, search methods have found attention operations to be beneficial for the classification. This is a natural discovery since not all bags are relevant to the classification of the single type of device. Optional connections between the aggregations and the output in the inner MIL block are typically enabled. This can be interpreted as some sort of mixture of different attention heads.

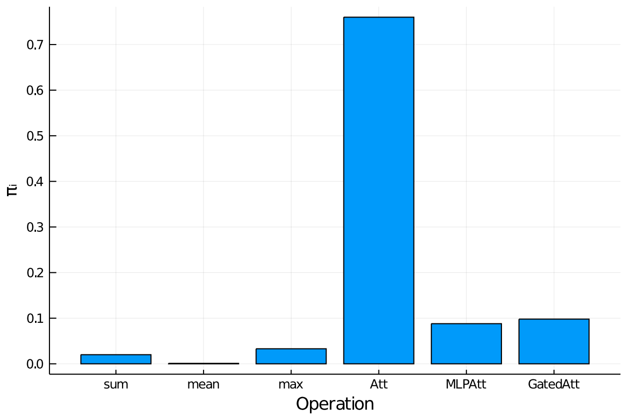

Figure 4.6: Parameters of the categorical distribution at the outer aggregation function in SNAS after convergence. Plain attention pooling is clearly leading.

4.1.6 Summary

According to the results of the conducted experiments, we conclude that no single search method is best in terms of performance for the search spaces used in the experiments. However, DARTS is consistently outperforming other algorithms in terms of convergence speed - it is orders of magnitude faster than Random Search baseline, even if the latter is augmented with the weight sharing strategy. It is still much faster than guided non-differentiable search represented by ENAS. Although SNAS has some appealing theoretical properties, it is on par or slightly worse than DARTS, so there is no strong reason to use it. In general, search methods are able to discover adequate architectures able to perform well.

4.2 Regularization of Architecture Distributions

We compare the DARTS algorithm augmented with Kullback–Leibler divergence regularization term, as described in the section 3.2.2. Two methods were trained in the IoT task setting, where KL divergence term could potentially aid the exploration and give more weight to attention functions. Results show that the effect is negligible as it does not affect the performance and the choice distribution remains nearly the same.

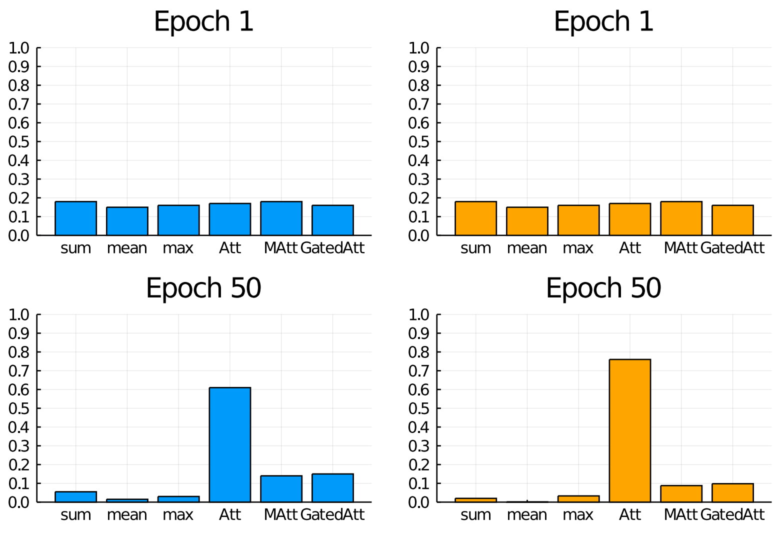

Figure 4.7: *Left*: Architecture distribution regularized by the KL divergence term. *Right*: Architecture distribution of the standard DARTS.

4.3 Single-Path Effect

Another small-scale experiment is conducted to measure the effect of single-path search modification of DARTS. We compare the methods in the setting of Sum of MNIST-bags task, as it is the most computationally intensive among all experiments.

Method	Accuracy	Relative training time
DARTS	0.88 ± 0.12	100%
SP-DARTS	0.87 ± 0.17	**89.2%**

Table 4.7: Comparison of the DARTS and SP-DARTS on the MNIST-bags data.

Results show that there is no significant drop in performance of SP-DARTS compared to the standard DARTS, whereas the former is somewhat faster to train, which makes SP-DARTS a good candidate when the computing power is limited. However, more extensive experiments should be conducted to reach a definitive conclusion.

4.4 Discussion

According to results of the search strategy comparison, we advice to use DARTS algorithm mainly by virtue of its fast convergence speed. Other advantages of DARTS are the straightforwardness of the implementation and its simplicity, which leaves a lot of room for customization. It might be beneficial to use KL divergence regularization on the architecture parameters or employ the single path variant of the algorithm to achieve further speed up. A good alternative is the SNAS algorithm, which is comparable with the DARTS in terms of performance and speed, and has built-in exploitation-exploration trade-off control in a form of the Gumbel-Softmax temperature parameter.
Random Search has proven to be a sensible baseline in terms of performance, even though it lacks the speed of differentiable methods. We hypothesize that this result serves as the evidence in favor of importance of the search space design over the choice of the particular search algorithm. When the search space contains enough adequately performing architectures, which can be considered as local optima, Random Search will eventually pick one of them given enough iterations. Other, more advanced methods can easily get stuck in these local optima or oscillate between them. Thus, an important research direction is the development of general and task-agnostic search spaces for Multiple-Instance Probelms. There are some developments in this area[32][20], however, they are mostly centered around convolutional architectures.
Random Search with Weight Sharing improves upon the speed of the standard RS algorithm since weight sharing allows to estimate the performance of candidate architecture efficiently without retraining them from scratch. It is worth noting that in our experiments ENAS outperformed RS in terms of speed because of the lower number of architecture evaluations. It is possible that RS could reach comparable results in a shorter amount of time. This requires more extensive experiments focused on the Random Search algorithm.
It is clear that search methods are constrained by the capacity of the search space. The bias that is introduced by adopting patterns from the human-designed architectures can potentially limit the performance of discovered architectures. Novel network architectures can not be discovered if the search space is too constrained, and in this case NAS algorithms reduce to an efficient hyper-parameter optimization method.

Chapter 5

Conclusions

This thesis contains a theoretical overview of the state-of-the-art Neural Architecture Search methods and the experimental study of the performance of selected search algorithms on Neural Networks for Multiple Instance Learning Neural Networks, which are designed to operate on sets of instances. We also briefly review the general idea of MIL networks and some recent developments in this area, such as attention pooling.

The goal of Neural Architecture Search methods is to automate the design of neural network architectures, which is a time-consuming task that requires extensive human expert knowledge. NAS is a novel technique and is still in active research. State-of-the-art methods in this area were evaluated mainly on convolutional and recurrent architectures. We transfer these methods to the Multiple Instance Learning domain by developing a search space for MIL networks. Next, we run extensive experiments evaluating selected NAS algorithms and their variations on four different MIL tasks and analyze the results, providing practical recommendations on applying NAS methods to design MIL neural networks. We implemented selected search algorithms and an interface for search spaces in the Julia programming language to conduct the experiments.

We conclude that search space design is a critical component to the successful application of NAS methods. Thus, potential future work could be aimed at designing general and expressive search spaces agnostic to the particular learning task and unifying search spaces for MIL and other networks.

Appendix A

Implementation Details

This appendix contains the technical information about the implementation and experiment reproduction. The code is available as the supplementary material to the text and in the GitHub repository `https://github.com/nikitati/Nas.jl`.

A.1 Library Usage

We implement a minimalistic package for neural architecture search `Nas.jl` using the Julia language. The main primitive is the `ChoiceNode` which represents an architecture choice. Using choice nodes, users can build arbitrary search spaces in a similar fashion of how models defined in the `Flux.jl` library. Choice node can be parameterized by either `StatefulSoftmax`,`GumbelSoftmax` or `nothing`. The last case is intended for non-differentiable search algorithms. Library provides implementations of the Random Search, DARTS, SNAS and ENAS algorithms. They are available via corresponding structures `RandomSearch`, `DARTSearch`, `SNASearch` and `ENASearch`. Once configured, they can be run with the `optimize!` method. Optimized search space can be sampled with the `sample_architecture` method to obtain candidate architectures.

Listing A.1: Nas.jl usage example

```
using Flux
using Nas
```

```
searchspace = Chain(
    Dense(28*28, 256, Flux.relu),
    ChoiceNode(GumbelSoftmax, [
        Dense(256, 128, Flux.relu),
        Dense(256, 128, Flux.tanh)
    ]),
    )

snas = SNASearch(epochs=10, tmpdecay=0.01, opt=ADAM())
data = ...
loss = ...
optimize!(snas, searchspace, loss, data)

model = sample_architecture(searchspace)
Flux.train!(model, ...)
```

A.2 Running the Experiments

We tried to make the experiments as reproducible as possible. It is achieved with the Julia packaging system Pkg and docker technology. Docker is used to automate the process of installing the Julia language and downloading and compiling necessary packages.
The musk dataset is included in the supplementary meterial, however, other datasets have to be downloaded from sources and pre-processed with the `generate_data.jl` script.

Listing A.2: Running the musk experiment from docker

```
docker build -t nas . # run in the Nas.jl directory
docker run -it --rm nas julia --project musk.jl
```

Appendix B

Bibliography

[1] Jaume Amores. Multiple instance classification: Review, taxonomy and comparative study. *Artif. Intell.*, 201:81–105, August 2013.

[2] Gabriel M. Bender, Pieter jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *Proceedings of Machine Learning Research*, 2018.

[3] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, page 2546–2554, Red Hook, NY, USA, 2011. Curran Associates Inc.

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Rev.*, 59(1):65–98, January 2017.

[5] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[6] Xuanyi Dong and Yi Yang. One-shot neural architecture search via self-evaluated template network, 2019.

[7] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[8] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey, 2018.

[9] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated

machine learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2962–2970. Curran Associates, Inc., 2015.

[10] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, pages 507–523, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[11] Maximilian Ilse, Jakub Tomczak, and Max Welling. Attention-based deep multiple instance learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2127–2136, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[12] Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 3(25):602, 2018.

[13] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[14] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

[15] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining - KDD '19*, 2019.

[16] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *arXiv preprint arXiv:1902.07638*, 2019.

[18] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. *Lecture Notes in Computer Science*, page 19–35, 2018.

[19] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search, 2018.

[20] Renato Negrinho, Darshan Patil, Nghia Le, Daniel Ferreira, Matthew Gormley, and Geoffrey Gordon. Towards modular and programmable architecture search, 2019.

[21] Y. E. NESTEROV. A method for solving the convex programming problem with convergence rate o(1/k^2). $Dokl.Akad.NaukSSSR, 269 : 543 - -547, 1983.$

[22] Tomás Pevný and Petr Somol. Discriminative models for multi-instance problems with tree structure. *ArXiv*, abs/1703.02868, 2016.

[23] Tomas Pevny and Petr Somol. Using neural network formalism to solve multiple-instance problems, 2016.

[24] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing, 2018.

[25] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms, 2012.

[26] Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. Single-path nas: Designing hardware-efficient convnets in less than 4 hours. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 481–497. Springer, 2019.

[27] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *ArXiv*, abs/1712.06567, 2017.

[28] Xinggang Wang, Yongluan Yan, Peng Tang, Xiang Bai, and Wenyu Liu. Revisiting multiple instance neural networks. *Pattern Recognition*, 74:15–24, 2018.

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

[30] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1912–1920, 2015.

[31] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. Snas: Stochastic neural architecture search, 2018.

[32] Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search, 2019.

[33] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3391–3401. Curran Associates, Inc., 2017.

[34] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning, 2016.