Ben-Gurion University of the Negev

Faculty of Engineering Sciences

The Department of Software and Information Systems Engineering

Thesis

DeepLine: AutoML Tool for Pipelines Generation using Deep Reinforcement Learning and Hierarchical Actions Filtering

Yuval Heffetz

Thesis submitted in partial fulfillment of the requirements

for the Master of Sciences degree

Under the supervision of **Supervisor**

Month year

Ben-Gurion University of the Negev
The Faculty of Natural Sciences
The Department of **Computer Science**

Thesis Title

Author

Thesis submitted in partial fulfillment of the requirements
for the Master of Sciences degree

Under the supervision of **Supervisor**

Signature of student: _____ Date: _____
Signature of supervisor: _____ Date: _____
Signature of chairperson of the
committee for graduate studies: _____ Date: _____

Month Year

Abstract

Automatic machine learning (AutoML) is an area of research aimed at automating machine learning (ML) activities that currently require human experts. One of the most challenging tasks in this field is the automatic generation of end-to-end ML pipelines: combining multiple types of ML algorithms into a single architecture used for end-to-end analysis of previously-unseen data. This task has two challenging aspects: the first is the need to explore a large search space of algorithms and pipeline architectures. The second challenge is the computational cost of training and evaluating multiple pipelines. In this study we present DeepLine, a reinforcement learning based approach for automatic pipeline generation. Our proposed approach utilizes an efficient representation of the search space and leverages past knowledge gained from previously analyzed datasets to make the problem more tractable. Additionally, we propose a novel hierarchical-actions algorithm that serves as a plugin, mediating the interaction between the environment and the agent in deep reinforcement learning problems. The plugin significantly speeds up the training process of our model. Evaluation on 56 datasets shows that DeepLine outperforms state-of-the-art approaches both in accuracy and in computational cost.

Contents

1 Introduction

The explosion of digital data has made the use of machine learning (ML) more ubiquitous than ever before. Machine learning is now applied to almost any aspect of organizational work, and used to generate significant value. The growth in the use of ML, however, was not matched by a growth in the number of people who can effectively apply it, namely data scientists. This shortage in skilled practitioners has spurred efforts to automate various aspects of the data scientist's work.

Automatic machine learning (AutoML) is a general term used to describe algorithms and frameworks that deal with the automatic selection and optimization of ML algorithms and their hyperparameters. Examples of AutoML include automatic hyperparameter selection for predefined algorithms [31], automatic feature engineering [35], and neural architecture search [6]. While effective, the above mentioned studies sought to optimize only specific steps of the overall process undertaken by human data scientists. In recent years, studies exploring the problem of *automatic ML pipeline generation* have sought to automate the process end-to-end by generating entire ML pipelines. An example of such pipeline can be seen in figure 1.1

The creation of entire ML pipelines is challenging because it involves a

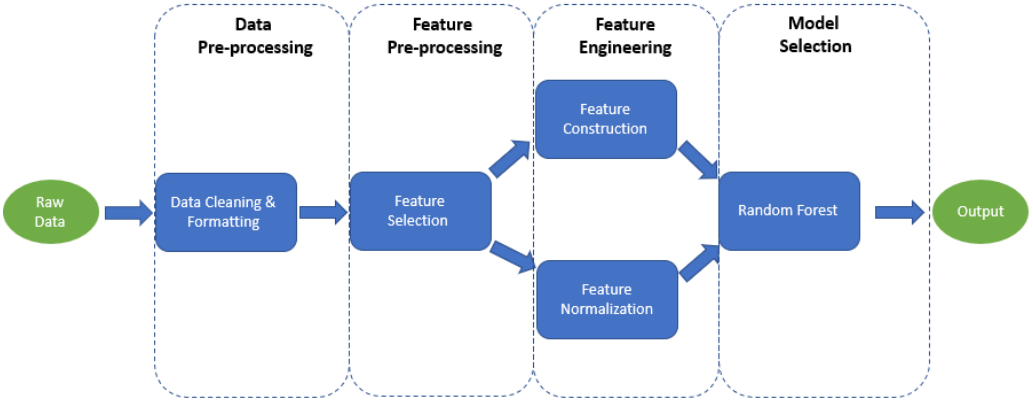

Figure 1.1: pipeline

large and complex search space. Even simple pipelines usually involve multiple steps such as data preprocessing, feature selection, and the use of a classifier. Complex pipelines can both contain additional types of algorithms (e.g., feature engineering) and multiple algorithms from each type. The large number of available algorithms and the fact that the performance of each component of the pipeline is highly dependent on the input it receives from previous component(s) further complicates this task.

Existing approaches for automatic pipeline generation can be roughly divided into two groups: *constrained space* and *unconstrained space*. Constrained space approaches generally create a predefined pipeline structure and then search for the best algorithms combination to populate it. Studies that utilize this approach include Auto-Sklearn [21] and Auto-Weka [70]. This approach narrows the search space, but it prevents the discovery of novel pipeline architectures. The unconstrained space approaches place little or no restrictions on the structure of the pipeline, but they come at a higher computational cost. Approaches of this kind include TPOT [49] and AlphaD3M [18].

In this study we propose DeepLine, a novel *semi-constrained* approach for AutoML pipeline generation. While our approach constrains the maximal size of the pipelines, it supports the inclusion of multiple algorithm of the same type (e.g, classification), as well as the creation of parallel sub-pipelines. In addition, any compatible algorithm(s) can serve as the input of another, ensuring that novel and interconnected architectures can be discovered.

Another important advantage of DeepLine over previous work is its ability to learn across multiple datasets. We apply deep reinforcement learning (DRL) techniques that enable our approach to perform all of its learning offline. This fact considerably speeds up performance for new datasets while enabling us to leverage past experience and improve DeepLine's performance over time.

Our contributions in this study are as follows: (1) We present DeepLine, a novel approach for automatic ML pipeline generation. Our approach learns across multiple datasets, enabling it to efficiently produce pipelines for new datasets; (2) We propose a novel action-modeling approach, which enables us to use a fixed-size representation to model dynamic action spaces. This hierarchical solution not only speeds up the training process of the DRL agent but also enables the use of DRL methods that only support a fixed number of actions. We implement our solution on the OpenAI Gym platform and publish the code, and; (3) We conduct an extensive evaluation on 56 datasets and show that DeepLine achieves comparable or better results than state-of-the-art methods at a fraction of the computational cost.

2 Background and Related Work

In this chapter we review the relevant literature in machine learning, AutoML and reinforcement learning, which are the main focus of this study.

2.1 Machine Learning

Before diving into the domain of automatic machine learning, we provide a short review and explanation of the machine learning field.
The machine learning (ML) field evolved from the broad field of artificial intelligence, which aims to mimic intelligent abilities of humans by machines. In the field of machine learning one considers the important question of how to make machines able to ”learn”. Learning in this context is understood as inductive inference, where one observes examples that represent incomplete information about some ”statistical phenomenon” [47]. Machine learning systems automatically learn programs from data. In the last decade the use of machine learning has spread rapidly throughout computer science and beyond [17]. Machine learning is used in web search, spam filters, recommender systems, ad placement, credit scoring, fraud detection, stock trading, drug design, and many other applications. A 2011 report from the McKinsey Global Institute asserts that machine

learning will be "the driver" of the next big wave of innovation [42].
In ML, we differentiate between supervised and unsupervised learning. Supervised learning occurs when the data used to train the model is labeled. Common algorithms include regression models (linear, logistic, polynomial etc.), neural networks, and decision trees. A label refers to the correct value of output associated with the input of a specific sample. In supervised learning, the goal is to find some hypothesis (a mapping h from input to output) that will agree with the training set (correctly maps the given input to its respective output); this hypothesis will be a good mapping for the general case and correctly classify or predict outputs for previously unseen inputs. The larger and more variant the original training set, the better h will generalize to new examples. In unsupervised learning, the data associated with the problem lacks these above-mentioned labels, so the data is a large set of input vectors. Our task, then, is to find structure or trends in this input data that may help us achieve understanding or explanation. Different outputs of such algorithms may be clustering information, density estimation of the data or other explanation oriented outputs [1]. Before applying ML models on a dataset, a certain amount of data manipulation and preprocessing is usually needed to fit the dataset for the model. In some cases the data preprocessing stage can vastly improve the end result of the ML model.
In the following sections, basic ML models and approaches which are relevant to this research work will be reviewed.

2.1.1 Artificial Neural Networks

Recent advances in the reinforcement learning field, which plays a main role in this research, often make use of artificial neural networks, and more specifically deep neural networks.
Artificial Neural Networks (ANNs) were inspired by the human neural mechanism as far as we know it. Their structure constitutes of multiple densely interconnected neurons in order to enable the system learn rules, recognize patterns and make complex decisions. ANNs learn using a weight modification rule, allowing the network to establish an internal structure applicable to a certain task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. When the input units are directly connected to the output units it is relatively easy to find connections that will reduce the difference between the actual and desired output vectors [57]. Learning becomes difficult, but more powerful, when we include hidden units whose actual or desired states are not specified by the task.

ANN Structure and Rules

Deep ANNs are made of several layers. The first layer of the network is the input layer. This layer is comprised of units that receive various forms of information from the outside world that the network attempts to learn, recognize or process. After the input layer, there are several intermediate layers, which we call *hidden* layers, consisting of up to millions of units that can interact with one another. the final layer of the network is the output layer. The rules of the networks are as follows: connections within a layer or from a higher layer to a lower one are forbidden, to void cir-

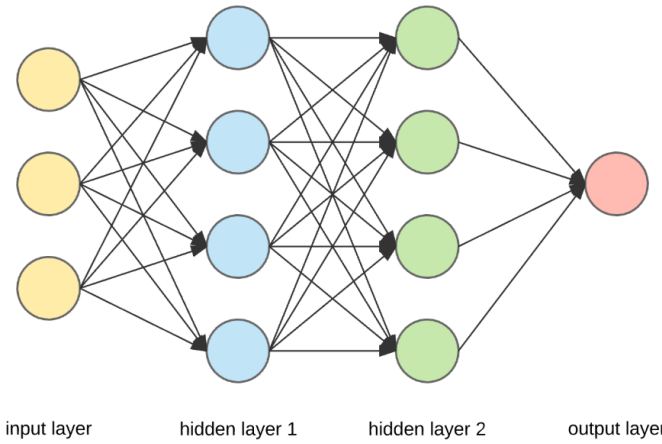

Figure 2.1: A simple neural network with an input layer, an output layer and two hidden layers

cles. However, connections can skip intermediate layer. The input vector sets the input unit's state. Then the state of every unit in each layer is determined by applying a non-linear function on the output of the previous layer. A graphic example of an ANN with two hidden layers is presented in Fig. 2.1.

Linear and Non-Linear Functions The input of layer i (x_i) is the output of layer j (y_j) multiplied by a set of weight between layer i and j (w_{ij})

$$x_i = \sum_j y_i w_{ij} \tag{2.1}$$

After applying a linear function, the output of unit j (y_j), is a non-linear function of its total input, such as

$$y_j = \frac{1}{1+e^{-x_j}} \qquad OR \qquad y_j = max(0, x_j) \tag{2.2}$$

Loss Function The aim of the learning procedure is to find a set of weights (W) ensuring that for each input vector, the output vector produced by the network is sufficiently close to the desired output vector. The total loss L, in a single classification experiment is defined as

$$L_{softmax}(W) = -\sum_{i=1}^{n} \log P(y_i|x_i, W) \tag{2.3}$$

Where x_i, y_i are included in the sample $S = \{x_i, y_i\}_{i=1}^{n}$. We minimize L over the weights (W) where for class $m = 1, ..., M$:

$$P(y = m|x) = \frac{exp(f_m)}{\sum_{c=1}^{M} exp(f_c)} \tag{2.4}$$

Where f_m is the score of the output units, given an example x. The loss function in this case is the softmax loss but other loss functions can be applied. In order to find this set of weights (W), a back propagation process is applied.

Minimizing Loss Function Using Back Propagation The back propagation (BP) learning algorithm is used to modify the weights (W) of an ANN. Weights are updated according to their contribution in reducing the error. In the BP process the partial derivatives of the error are being calculated, with respect to the weights. The weights can be updated in a batch; Instead of updating W after every single example (known as online stochastic gradient descent), the derivatives of an example is accumulated over a batch of B examples (known as stochastic gradient descent with batches). After calculating the gradient of B examples and summing the loss over time, the weights are being adjusted. This process is done iteratively where in every step the B examples are chosen randomly from all input examples.

Recurrent neural networks

Recurrent Neural Networks (RNNs) are models that allow to make use of sequential data [44]. In a traditional neural network we assume that all inputs (and outputs) are independent of each other. But for some tasks, which use longitudinal data, the variables (inputs and output) cannot be treated as independent; modeling the relations between the variables in a temporal approach is vital in those cases.
RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Another way to think about RNNs is that they have a *memory* which captures information about what has been calculated so far. Fig. 2.2 presents an RNN diagram, folded and unfolded. By unfolding the network we simply mean that we write out the network for the complete sequence. For example, if the sequence we care about is a sentence of 5 words, the network would be unrolled into a 5-layer neural network, one layer for each word. The formulas that govern the computation happening in a RNN are as follows:

- x_t is the input at time step t.
- The parameters of the networks, the weights U, V and W, are shared by all steps of the RNN. This reflects the fact that we are performing the same task at each step, just with different inputs. This greatly reduces the total number of parameters we need to learn.
- h_t is the hidden state at time step t. It's the *memory* of the network. h_t is calculated based on the previous hidden state and the input at the current step: $s_t = f(Ux_t + Ws_{t-1})$. The function f usually is a

nonlinearity such as tanh or ReLU. h_{-1}, which is required to calculate the first hidden state, is typically initialized to all zeros.

- h_t is also used for calculating the output at step t. For example, if we wanted to predict the next word in a sentence it would be a vector of probabilities across our vocabulary.

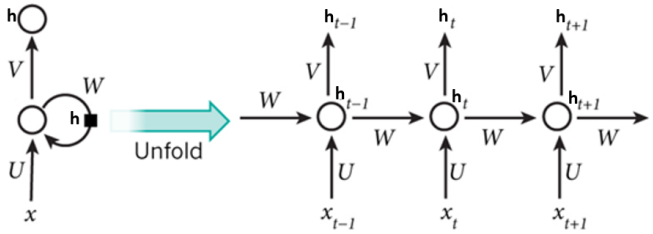

Figure 2.2: A recurrent neural network and the unfolding in time of the computation involved in its forward computation.

In theory RNNs can make use of information in arbitrarily long sequences, but in practice they are limited to looking back only a few steps. To overcome this problem, more advanced architectures of the recurrent cell were introduced and are commonly used, predominantly the gated recurrent unit (GRU) and the long short-term memory (LSTM) network. These are recurrent network units that excel at remembering values for either long or short durations of time. The key to this ability is that it uses no activation function within its recurrent components. Thus, the stored value is not iteratively squashed over time, and the gradient does not tend to vanish when back propagation through time is applied to train it [30] [14]. RNNs have been successfully applied to multiple domains such as: language modelling and prediction tasks including speech recognition [24] and machine translation [15], image recognition and characterization [74], anomaly detection in time series [41] and prediction in medical care path-

ways [16]. The use of RNN's in deep reinforcement learning is also a common practice since their memory ability can benefit the model in many RL domains. We will consider using an RNN as a part of our suggested system.

Long short term memory networks. One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous words might inform the understanding of the present context. Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in *"the clouds are in the sky"*, we do not need any further context - it is pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it is needed is small, RNNs can learn to use the past information. In other tasks, there is a large time gap between the relevant information and point when it is needed for prediction. Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.

In order to deal with this problem, Long Short Term Memory networks (LSTMs) were presented [30]. An LSTM unit is a recurrent network unit that excels at remembering values for either long or short durations of time. The key to this ability is that it uses no activation function within its recurrent components. Thus, the stored value is not iteratively squashed over time, and the gradient or blame term does not tend to vanish when back propagation through time is applied to train it.

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will

have a very simple structure, such as a single tanh layer. LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting between themselves.

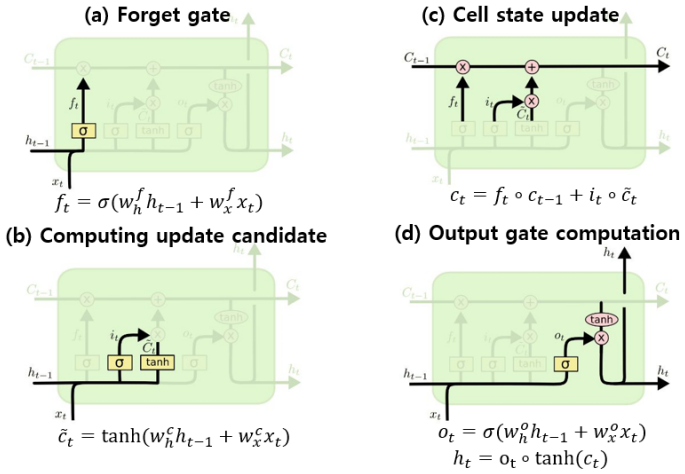

Figure 2.3: In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations.

The inner structure of LSTMs is demonstrated in Fig. 2.3. We will go through the structure's components:

1. Forget gate layer - Decides what information from the previous step the network keeps in its memory cell. This decision is made by a sigmoid layer. It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. 1 represents "completely keep the information" while 0 represents "completely forget the information".

2. Computing update candidate - i_t is the input gate layer which is a

sigmoid layer which decides what values in the memory cell will be updated using the current step's input and the hidden state from the last time point. Next, a tanh layer creates a vector of new 'candidate' values, $\tilde{C}_t$, that could be added to the cell state.

3. Cell state update - In this step, C_{t-1} is updated to C_t. We multiply C_{t-1} by f_t. Then we add $i_t * \tilde{C}_t$.

4. Output gate computation - The output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

2.2 Automatic Machine Learning (AutoML)

In recent years, the field of automated machine learning has gained much interest and had been the focus of many researches. While our suggested research is dealing with the entire end-to-end ML pipeline as defined in chapter 1, many of the previous works have focused on only parts of the pipeline, for example the automatic optimization of hyper-parameters or the automatic selection of features [32]. The automation of the entire process was researched in more recent works. We will detail these works in the this section.

2.2.1 Hyperparameters Optimization

Hyperparameters are considered to be parameters of machine learning algorithms that are not optimized in the learning process of the ML model but have to be tuned either manually or with some kind of a grid search. For example, the learning rate of gradient descent is considered as a hyperparameter. The chosen hyperparameters' values can have a tremendous effect on the performance in many machine learning algorithms, such as support vector machines, deep neural networks, and deep reinforcement learning hence the importance of finding the optimal configuration from the search space.

Grid search is the most commonly-used form of hyperparameter optimization that applies brute force search to explore a broad range of model parameters in order to discover the parameter configuration that allows for the best model fit. Unfortunately, this method can take a great deal of run-time and computational cost. Avoiding it may require expertise and intuition in the "art" of hyperparameters tuning. There is therefore great appeal for automatic approaches that can optimize the performance of any given learning algorithm to the problem at hand. Recent research has shown that randomly evaluating parameter sets within the grid search often discovers the ideal parameter set more efficiently than exhaustive search [7], which shows promise for intelligent search in the hyperparameter space.

The most prominent intelligent search method for hyperparameters is Bayesian Optimization (BO) based on Gaussian processes (GPs). In Bayesian optimization we are interested in finding the minimum of a function $f(x)$ on some bounded set x , which we will take to be a subset of the search space

[64]. Bayesian optimization constructs a probabilistic model for $f(x)$ and then exploits this model to make decisions about where in x to next evaluate the function, while integrating out uncertainty. The BO method was implemented, .e.g, in the Spearmint system [64]. In their work, Snoek et al. have shown that the method is very effective and it outperformed manual hyperparameter tuning by expert practitioners [64]. There are also several works on Bayesian Optimization which are designed specifically for large scale hyperparameters configuration problems like AutoML. For example, RoBO [65] includes multiple implementations of different Bayesian Optimization algorithms with the flexibility of changing the components of the optimization process. In their work, springenberg et al. proposed to use neural networks as a powerful and scalable parametric model, while staying as close to a truly Bayesian treatment as possible. Another example is the Hyperopt [8], which is a python library for optimizing the hyperparameters of machine learning algorithms. Hyperopt also uses Bayesian optimization methods. Its interface requires users to specify the configuration space as a probability distribution. Specifying a probability distribution rather than just bounds and hard constraints allows better understanding of which values are plausible for various hyperparameters. Another Bayesian approach for large scale hyperparameter search is presented in SMAC [31].

Finding the optimal hyperparameters configuration was also conducted with evolutionary algorithms (EA), such as in TPOT [49] [48] and in Autostacker [12]. Both models offer broader pipeline-based frameworks, but use EA for the part of hyperparameters optimization, as will be detailed later-on.

2.2.2 Feature Generation and Selection

Another line of works focuses on automatic feature generation and selection. Feature engineering is an important step in the data science workflow. Any machine learning algorithm relies on data that holds meaningful information, so it will be able to derive insights and predictive models out of it. When dealing with structured data, the data scientist must first form variables, otherwise known as features. The data scientist may start by using given static fields like gender, age, etc. from the tables as features, then form some specialized features by intuiting what might predict the outcome. Next, the scientist may develop new features that transform the raw fields into different measures. This process of transforming the raw data to a more meaningful state is traditionally done manually, relying heavily on the data scientist intuition and rule-of-thumbs and can consume most of the time dedicated in constructing the ML pipeline. Since standard ML algorithms, unlike deep learning algorithms, still rely on the input features for a better performance, there is still importance in developing feature engineering methods.

One work in this field resulted in the "Data Science Machine", which is a framework that automatically constructs features from relational databases via deep feature synthesis [34]. In essence, the algorithm follows relationships in the data to a base field, and then sequentially applies mathematical functions along that path to create the final feature. By stacking calculations sequentially, it is possible to define each new feature as having a certain depth, d. Hence, the algorithm is called Deep Feature Synthesis. features are calculated at different depths, d, by traversing relationships between entities. In their work, Kanter et al. demonstrated the crucial role

of automated feature construction in machine learning pipelines by entering their Data Science Machine in three machine learning competitions and achieving expert-level performance in all of them.

ExploreKit is another state-of-the-art framework for automatic feature generation. The framework automatically generates a large set of candidate features by applying multiple feature transformation operators, that were pre-defined based on the premise that highly informative features often result from manipulations of elementary ones, on the original features. It then uses a feature selection algorithm to select features out of the candidate feature space by predicting their usefulness. It first ranks the candidate features without evaluating the model with these features set, using a feature ranking classifier that takes as input meta-features representation of both the dataset and the candidate feature. Finally it evaluates the ranked features list and selects them using a greedy search. In their work, Katz et al. have been able to significantly improve the performance of multiple classifiers on a large variety of datasets using ExploreKit.

2.2.3 Model Selection

The selection of the actual ML prediction model is probably the most important component of the data science pipeline since it's the component that is used for training and prediction. A few works in the autoML field have focused their efforts in the direct selection of the prediction model, separately from the other parts of the pipeline. The most prominent approach for model selection and recommendation is based on meta- learning [10]. In this approach, different measures such as statistical measures, information-theoretic measures and performance results of the datasets

on different algorithms are extracted from datasets (and algorithms) to characterize them and to establish a connection between them and the algorithms. The purpose of this approach is to build a model that will be able to capture the relationship between the meta-data characterizing the datasets and the performance of algorithms on the datasets, for producing a ranked list of algorithms for a dataset [63].

A different approach, introduced with the Sommelier system [CITE] suggests the recommendation of algorithms to previously unseen datasets by domain-knowledge extraction from scholarly big data. The proposed system creates a word embedding representation of algorithms based on academic papers and matches algorithms to datasets based on keywords found in the datasets descriptions.

2.2.4 Pipeline-Based Models

In 2015, Fuerer et al. developed an autoML system called auto-sklearn [21], which uses Bayesian optimization to discover the ideal combination of feature preprocessors, models, and model hyperparameters to maximize classification accuracy. However, auto-sklearn explores a fixed set of pipelines that only include one data preprocessor, one feature preprocessor, and one model. Thus, auto-sklearn is incapable of producing arbitrarily large pipelines, which may be important for autoML. Following the same guidelines as auto-sklearn, Auto-Weka automatically selects a primitive for each one of a pipeline step with a pre-defined structure and then uses Bayesian Optimization (Sequential model-based optimization) to search for optimal hyperparameter settings of the pipeline [70] [38]. The pipeline fixed structure follows the traditional data science work-flow:

from data preprocessing, feature engineering to single model prediction. The main downside of fixing the structure of the pipeline to a very limited search space is its lack of flexibility to different kinds of problems. More specifically, these method has shown to be less suitable for complicated problems or small sample datasets [12]. In 2015, Zutty et al. similarly demonstrated an autoML system using genetic programming (GP) to optimize machine learning pipelines, and found that GP is capable of designing better pipelines than humans for one supervised classification task [81].

An attempt to extend the traditionally defined pipeline was first done by Olson et al. with TPOT, a tree-based pipeline optimization tool for automating ML [49]. TPOT allows for parallel feature engineering prior to model prediction. Subsequently, TPOT uses Evolutionary Algorithms and genetic programming to treat the parameter configuration problem as a search problem. Olson et al. define machine learning operators to be used as genetic programming primitives. These primitives are combined to a tree-based pipeline that forms a complete machine learning model. An example for a TPOT pipeline can be seen in Fig. 2.4 . Each primitive of the pipeline (i.e., ML operator) corresponds to a machine learning algorithm, e.g. a classifier, an algorithm that normalizes the features' values or any other ML algorithm. TPOT only uses algorithms that are modeled in the scikit-learn Python library [51]. The primitives are divided in the paper into three different kinds - supervised classification operators, feature preprocessing operators and feature selection operators. The combination of the primitives to a single pipeline tree is conducted with GP. TPOT enables the creation of several copies of the original dataset so that each copy will be modified by a different series of primitives and later unified again to

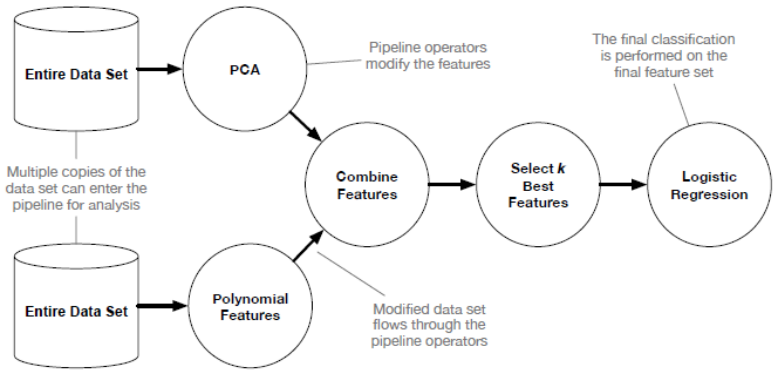

Figure 2.4: An example of a TPOT pipeline constructed out of machine learning operators, used as primitives. Image taken from [49].

form the input of the prediction primitive (e.g., classifier), as can be seen in Fig. 2.4. Each primitive receives as input a dataset and outputs a modified dataset. This enables the model to act simultaneously on different modified copies of the dataset and thus result in a flexible pipeline structure and representation of machine learning models. The paper also specify the exact form each dataset that TPOT operates on should adhere to. For the optimization of the pipeline, the GP algorithm that is used follows a standard process which includes the generation of 100 random tree-based pipelines, selection of the top 20 pipelines according to the cross-validation accuracy and the size of the pipeline (aspiring to minimize size) to the next step, and some further offspring crossovers in the next generation as detailed in the paper. The tool was evaluated on 150 different datasets with number of records ranging between 60 to 60,000 and with varying number of features and outperformed a basic machine learning analysis on several datasets. One of the main drawbacks of this tool, that we intend to address in this research, is the lack of use in possible meta-learning methods to enhance the ability of the model to match pipelines with datasets, as offered by [22].

Similarly to TPOT, Chen et al. developed a system called Autostacker based on a pipeline generation with evolutionary algorithms (EA) [12]. Unlike TPOT, Autostacker represents the pipeline as a stack of layers. The recently published paper combines a hierarchical stacking architecture with EA for the parameter search. Autostacker is inspired by the stacking method of ensemble learning, and automatically discovers pipelines made up of one or many models. The model enhances the flexibility of the pipeline structure by allowing the use of a combination of prediction primitives (such as SVM, regression, etc.) instead of only one such primitive as in TPOT. Autostacker uses EA to find solution in the search space that includes the different primitives, the structure of the pipeline (number of primitives in each stacking layer) and the hyperparameters in each primitive. The system achieved competitive or superior results on the 15 datasets that were evaluated compared to TPOT and auto-sklearn. The system should further be explored on greater number of datasets and with a larger primitives search space.

Recently, another optimization and search algorithm was suggested for autoML following a similar representation of the pipeline in a system called AlphaD3M [19]. The system is based on meta reinforcement learning using sequence models and self play and proclaim competitive performance to other pipeline-based models while presenting faster computation and explainability of the results. AlphaD3M is representing the search problem and pipeline discovery as a single-player game [43], where the player iteratively builds a pipeline by selecting among a set of actions which are insertion, deletion and replacement of pipeline primitives. Each action is essentially the transition from the current state to the next state, which is represented as the meta data and the entire pipeline chain, rather than in-

dividual primitives. A pipeline, together with the meta data and problem definition is analogous to an entire game board configuration. The reward of the action is defined as the performance of the pipeline. Drori et al. use deep reinforcement learning Inspired by AlphaZero [61] and expert iteration [3]: they use a recurrent neural network (LSTM) for predicting pipeline performance and action probabilities, along with a Monte-Carlo Tree Search (MCTS) which takes the network's actions probabilities and an estimation of the model's performance and searches a better pipeline sequence. Next an iterative self improvement with self-play takes place, as can be seen in Fig. 2.5. This iterative dual process results in an efficient solution in this high dimensional search space. One of the advantages of the system is its explainability, since it includes all the decisions made in the way to the full pipeline.

The AlphaD3M framework is integrated and based on Darpa's data driven discovery (D3M) program, which is an open source project involving dozens of universities from around the globe, aimed at the development of autoML systems. Our research will also integrate with the D3M project, as will be detailed in the following sections.

While effective, all the above methods perform all their learning on the currently-analyzed dataset. Several approaches do use meta-learning, but only for initialization purposes. Our approach, on the other hand, relies heavily on learning from previously-analyzed datasets and is therefore capable to produce high-quality pipelines for new datasets at a fraction of the time.

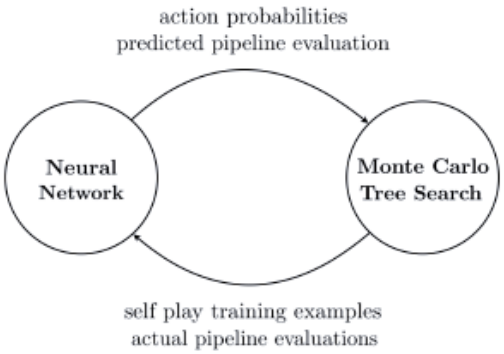

Figure 2.5: An illustration of the AlphaD3M process. Image taken from [19].

2.2.5 Neural Architecture Design with RL

The automatic design of neural networks architecture is a part of the autoML effort in recent years. Although it is not a part of the data science more traditional pipeline discussed in this research, the methods for automating the design of NNs are closely related to those of automating the design of the data science pipeline. While the field has seen a few different methods, such as evolutionary algorithms, to search and optimize the networks' architectures, we will focus on reinforcement learning (RL) methods since it is more related to our research.

Zoph et al. used REINFORCE, which is a policy gradient deep-RL algorithm detailed in subsection 2.3 to maximize the accuracy of generated NN architectures on given datasets and tasks. They used an RNN in their DRL model and formulated the problem as an RL environment where the hyperparameters space of the network are generated as a sequence of tokens, which form the actions of the environment, and the rewards are the accuracy results on the validation sets. each gradient update to the pol-

icy parameters corresponds to training one generated network to convergence. the proposed approach achieved competitive results for an image classification task with CIFAR-10 dataset and better results for a language modeling task with Penn Treebank. Baker et al. [4] proposed a meta reinforcement model called metaQNN for automatic CNN architecture generation with a Q-learning algorithm. They used the $\epsilon - greedy$ and experience replay methods in the Q-learning algorithm and defined a large finite space of finite architectures which the agent iteratively searched to find designs with better performance on the learning task. In this environment, the state was defined as a tuple of all relevant layer parameters and the action space was constrained to allow transitions ronly resulting in DAGs. In 2017, Zhong et al. [79] proposed to construct network blocks to reduce the search space of network design, trained by Q-learning and in 2018 [80] they introduced the BlockQNN model which also utilized a distributed asynchronous framework and an early stop strategy. In their model, Zhong et al. generate network blocks with the RL agent to later stack them together to form a larger and more complex networks. The state in their environment is represented using a matrix defining the blocks' layers graph.

2.3 Deep Reinforcement Learning

Deep reinforcement learning (DRL) has evolved out of the reinforcement learning (RL) field and denotes the incorporation of neural networks in RL algorithms.

2.3.1 Reinforcement Learning

Reinforcement learning (RL) is a sub-field of machine learning. It refers to the group of algorithms and methods in which an agent is interacting with an environment and learning the optimal policy to behave in this environment to achieve a goal or a maximum reward, and It is a powerful tool for sequential decision making problems [67]. RL is applied on a wide range of fields such as computer games and neural architecture design.

Problem definition In a reinforcement learning model an agent interacts with the environment over time. Under a specific policy π, the agent forms a trajectory of actions and states where in each time-step t, the agent has to choose an action a_t given the current state s_t that will bring it to the next state s_{t+1} (i.e. s'), with a reward r_t which is obtained from the reward function $R(s, a, s')$. This *transition* between states is governed by the environment's (i.e. model) dynamics that can be known or unknown. The transition function $P(s_{t+1}|s_t, a_t)$ defines the probability to end up in state s_{t+1} when taking action a_t in state s_t. Understandably, in deterministic environments this probability is always one. The state space of the environment is denoted by S and the action space is denoted by A. Both spaces can be either continuous or discrete. The underlying goal of the agent is to maximize the return, which is defined in eq. 2.5.

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{2.5}$$

The return is the discounted accumulated reward with a discount factor $\gamma \in (0, 1]$ which, roughly speaking, denotes how much the agent cares about current rewards in respect to future rewards. Some environments

are episodic, meaning that they have terminal states that define an end of an episode.

Markov Decision Process and RL. When an RL problem satisfies the Markov property, i.e., that a state is dependent only on the previous state and action, it is formulated as a Markov Decision Process (MDP). MDP is a formalization of sequential decision making where the actions taken in a certain step can influence the future rewards of a trajectory [68]. MDP is defined by the tuple $M(S, A, P, R, \gamma)$ [54] consisting of:

- S - a set of possible states of the environment
- A - a set of all possible actions
- P - the transition function defining the distribution over to which state we will randomly transition taking an action in a state.
- R - the reward function.
- γ - the discount factor, $\gamma \in (0, 1]$

In an MDP, we pick a random initial state with some probability and in each time-step we choose an action a_t, as a result of which the current state s_t transitions to some next state s_{t+1} with a probability defined by $P(s_{t+1}|s_t, a_t)$. This forms a trajectory of states with a sum of discounted rewards computed with eq. 2.5. The process is demonstrated in fig 2.6. In RL, the goal is to find the best possible action to take in each state to maximize the expected sum of discounted rewards. Because of the Markov property of MDPs, to attain the optimal expected sum of rewards it is enough to choose actions only as a function of the current state s_t [67][68].

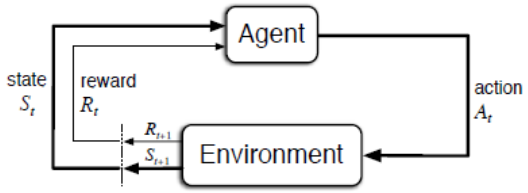

Figure 2.6: Interaction between an aget and the environment in a Markov Decision Process. Image taken from [68].

When this is given, the role of the reinforcement learning can be defined as finding a good *policy*. A policy is a mapping of all states and actions $\pi : S \rightarrow A$, so that if we take in each state s an action chosen by $\pi(s)$ we will be able to obtain the expected sum of discounted rewards:

$$E_\pi[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + ...] \tag{2.6}$$

Value Function A value function is the expected accumulated discounted future rewards used to measure how good a state is. Given a policy π we can define its value function $V^\pi : S \rightarrow \mathbb{R}$ for taking actions according to π starting from state s:

$$V^\pi(s) = E_\pi[R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, s_2) + ...|s_0 = s] \tag{2.7}$$

Similarly, we can compute the value of a state-action pair using $Q - function$. Given a policy π we can define its Q-function $Q^\pi : S \times A \rightarrow \mathbb{R}$ for taking actions according to π starting from state s, first taking a certain action a:

$$Q^\pi(s, a) = E_\pi[R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, s_2) + ...|s_0 = s, a_0 = a] \tag{2.8}$$

If policy π would have been optimal (i.e., π^*), the optimal value function could have been computed. A formal definition of the optimal value function is:

$$V^*(s) = max_\pi V^\pi(s) \tag{2.9}$$

And similarly for the optimal Q-function:

$$Q^*(s,a) = max_\pi Q^\pi(s,a) \tag{2.10}$$

Using the *Bellman equations* we can decompose equations 2.7 and 2.8 to a recursive form, while taking into consideration the stochastic nature of the transitions:

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)(R(s,a,s') + \gamma V_\pi(s')) \tag{2.11}$$

$$Q^\pi(s,a) = \sum_{s'} p(s'|s,a)(R(s,a,s') + \gamma \sum_{a'} \pi(a'|s') Q_\pi(s',a')) \tag{2.12}$$

Combining eq. 2.9 with eq. 2.11 and eq. 2.10 with eq. 2.12 gives the optimal value function as a bellman equation, which states for the sum of discounted rewards starting from state s and acting optimally from that step on-wards, and the bellman's optimal Q-function:

$$V^*(s) = max_a \sum_{s'} p(s'|s,a)(R(s,a,s') + \gamma V^*(s')) \tag{2.13}$$

$$Q^*(s,a) = \sum_{s'} p(s'|s,a)(R(s,a,s') + \gamma max_{a'} Q^*(s',a')) \tag{2.14}$$

Temporal Difference Learning. RL algorithms can be roughly divided into 2 types - algorithms that are used when the dynamics of the environment are known, and model-free algorithms, for environments in which

they are unknown or very complicated. When the dynamics are known (i.e., the transition function is given) we can use dynamic programming algorithms such as value iteration, using eq 2.13, and policy iteration [68]. When the dynamics are unknown we can use methods that first learn the dynamics and then use them in one of the algorithms used for known dynamics environments. Another option is to use sampling-based algorithms that sample the environment to learn its dynamics along side finding a good policy, rather than learning the dynamics as an intermediate step.

Temporal difference (TD) learning usually refers to model-free RL methods that learn the value function $V(s)$ directly from experience with TD error in an incremental way [68]. It's a prediction problem which follows an update rule to compute an estimation of the value function:

$$V(s) \leftarrow V(s) + \alpha[R(s,a,s') + \gamma V(s') - V(s)] \tag{2.15}$$

The value function is updated in an iterative manner with the TD error multiplied by the learning rate α.

For the model-free control problem, that is, for optimizing the value function in an unknown MDP, we can use the on-policy Monte-Carlo control approach, for example with GLIE algorithm [62] or the on-policy TD learning approach, with SARSA algorithm [68]. $On-policy$ methods use the same policy for computing the value function and for choosing the next action to take. In contrast, $off-policy$ methods use one policy to compute the value and a different one for taking actions. The most prominent off-policy algorithm, which can also be considered as a model-free TD learning algorithm, is Tabular Q-learning.

Algorithm 1: Q-Learning Algorithm

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$;
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$;

```
foreach episode do
    Initialize S;
    foreach step of episode do
        Choose A from S using policy derived from Q (e.g., ε-greedy);
        Take action A, observe R, S';
        Q(S, A) ← Q(S, A) + α[R + γ max_a Q(S', a) − Q(S, A)];
        S ← S';
    end
end
```

In Q-learning we store the Q-value of each state-action pair in a lookup table. The values are being updated in each iteration according to the update rule presented in algorithm 1. We use a policy to pick actions in each step, for example $\epsilon - greedy$ method which selects a random action with some probability or greedily, i.e. action with the maximum Q-value. The purpose of this method is to find a trade-off between *exploration* of new states for avoiding unknown situations and *explotation* of the learned policy for better performance in each step [37]. We use a different policy, *greedy*, for choosing an action for the computation of the TD error and thus for optimizing the Q-value.

[Consider adding λ and eligibility trace]

Function Approximation. The methods we mentioned so far use lookup tables to store values of states or q-values of state-action pairs. Understandably, these methods can be used only for small scale problems due to memory considerations. When the state or action space is very large or

continuous, a function approximation can be used to compute the value or the q-value and thus avoid using look-up tables. The approximation function aims to generalize across states so it would not be necessary to visit each state to iteratively update its value. Considering a value-function approximator $\hat{V}(s, w)$ that is using the set of weights w, the TD learning update rule would be:

$$w \leftarrow w + \alpha[R(s, a, s') + \gamma\hat{V}(s', w) - \hat{V}(s, w)]\nabla\hat{V}(s, w) \tag{2.16}$$

The update of the approximator weights would be performed by following the gradient of the function in respect to the weights multiplied by the TD error. Before the rise in popularity of neural networks, linear function approximation was a customary choice for representing state and action by a feature vector and the q-value function as a linear combination of the features [68]. In the next sub-section we will detail extensively the methods for integrating neural networks as function approximation.

Policy Gradients The methods described so far use $Q^*(s, a)$ to find the optimal policy by deriving it from the policy using $\pi^* = \arg\max_a Q^*(s, a)$ (same could be done with $V^*(s)$). Instead, some methods try to optimize the policy $\pi(a \mid s; \theta)$ directly, working on the policy space itself. They use a probability over actions parameterized by θ, expressing the probability to take action a in state s and defining the policy itself. An example for such a distribution is *softmax*, defined by $\pi(a_i|s; \theta) = \exp(\theta_i) \ / \ \sum_j \exp(\theta_j)$. The parameters are updated with gradient ascent to maximize the expected return $R_t = \mathrm{E}[\sum_t^\infty R(s_t)|\pi_\theta]$.

REINFORCE [77] is a policy gradient method that is updating θ in each

iteration taking a step in the direction of:

$$\mathrm{E}_t[\nabla \log \pi_\theta(a_t \mid s_t) R_t] \tag{2.17}$$

For reducing the variance of the gradient estimate it is customary to reduce a baseline $b_t(s_t)$ from the return R_t. We call the resulting expression the *advantage*:

$$A(a_t, s_t) = R_t - b_t(s_t) \tag{2.18}$$

A good estimator for the baseline would be $V(s_t)$ since subtracting it from the return will Increase log-probability of an action proportionally to how much its returns are better than the expected return under the current policy. To avoid having to finish an entire episode for summing all the rewards to compute the return, the Q-value $Q^\pi(s, a)$ can be used to estimate the return [68]. We result in:

$$A(a_t, s_t) = Q(a_t, s_t) - V(s_t) \tag{2.19}$$

2.3.2 Deep RL

Deep reinforcement learning (DRL) stands for the group of algorithms that use reinforcement learning incorporated with neural networks for various purposes. For example, a NN can be used as a function approximation of the value function $\hat{v}(s; \theta)$ or of the q-value function $\hat{q}(s; \theta)$ as been suggested by Mnih et. al in the Deep Q-Network (DQN) algorithm [46]. Other uses for NN in RL are approximations of the policy $\pi(a|s; \theta)$ or of the model itself, i.e. the transition function and the reward function.

Deep Q-Network (DQN)

The idea of using a NN as a function approximator is not new, and dates back to 1996 [9]. However, the performance of the algorithm suggested back then, that used a TD-based update rule on the NN weights (see eq. 2.16), suffered from instability and divergence of the NN, especially when using in off-policy methods [71]. The breakthrough came in 2015, when Mnih et. al introduced the DQN algorithm that managed to stabilize the learning process of the NN [46]. Recall the q-learning algorithm detailed in alg. 1, which is an off-policy, model-free RL method. In q-learning we use a behavioral policy to pick the next action to be taken for exploration, and a different policy, a target policy, for choosing the action to compute and optimize the q-value function. DQN follows the same guidelines of q-learning but instead of using a lookup table for the q-values it uses a deep NN as a function approximator. The NN, and more specifically a CNN when the environment is a computer game as in [46], takes the observation representing the state as an input and outputs the predicted q-value for each possible action in that state and thus generalize the algorithm across states. It uses the same network architecture to approximate $Q(a, s)$ for the *target policy* which is greedy w.r.t $Q(a, s)$ (eq. 2.20) and for the *behaviour polic*y which is $\epsilon - greedy$ w.r.t $Q(a, s)$.

$$target = R_t + \gamma \max_{a'} \hat{Q}(S_{t+1}, a'; \theta) \tag{2.20}$$

The parameters θ of the network are updated with a gradient descent rule w.r.t the loss function:

$$\mathrm{E}_{s,a,s'}(target - Q(s, a; \theta))^2 \tag{2.21}$$

The main contribution of DQN is its two network's stabilization methods:

- ***Experience Replay*** - first suggested in [40], the experience replay is a large set containing the agent's past experiences, i.e. state-action-reward triplets. The set has a predetermined size and it is used in the algorithm to sample minibatches of experiences in random order for the optimization process. Its purpose is sample-efficiency and stabilization of the network's dataset. Instead of updating the weights with each experience and throwing it away, with experience replay we keep past experiences for further use and the update is *stochastic* gradient descent with samples which are more *i.i.d*.

- ***Fixed Q-targets*** - The loss function is computed w.r.t the targets which are being approximated by the same network which is being optimized and updated. This results in a network which is 'chasing' a moving target, which in turn causes instability and divergence. To stabilize the network, Mnih et. al separate the network into two different networks - one for computing the targets using an older set of parameters θ^{-}, and one for predicting the q-value which is being updated every minibatch optimization using parameters θ. Instead of updating the 'old' parameters of the target network each step, they do it only once every C steps by performing $\theta^{-} \leftarrow \theta$.

The full algorithm Mnih et al. used is presented in algorithm 2. In their case, the observations were the pixels of a computer game. To represent the state of the environment they introduced a preprocessing stage that stacks the last few frames of the game. That way they captured the change in the frames over time and used it as a state rather than just obtaining a static state from a single frame that provides little information. DQN

Algorithm 2: Deep Q-Networks algorithm

Initialize replay memory $\mathcal{D}$;
Initialize action-value function Q with random weights θ;
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$;
for *episode in 1 to M* **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$;
  for *t in 1 to T* **do**
    With probability ϵ select a random action a_t ;
    otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$;
    Execute action a_t in emulator and observe reward r_t and image x_{t+1};
    Set $s_t + 1 = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$;
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D;
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$;
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$;
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ w.r.t. θ;
    Every C steps reset $\hat{Q} = Q$, i.e., set $\theta^- = \theta$;
  end
end
return Q function;

achieved human experts comparable results on 49 Atari games, with only minimal domain knowledge, getting as inputs only pixels and game scores [5].

Since DQN was first published, several improvements have been suggested.

Double DQN. DQN has an over-estimation problem caused due to an upward bias in $\max_a Q(s, a; \theta)$. The max operator is used both for selecting

the best action and both for evaluating that action with the same set of parameters. As a consequence, it is more likely to select over-estimated values, and result in over-optimistic value estimates. In 2016, Hasselt et al. proposed a small fix to that problem, changing the computation of the target y_j so it will use θ for selecting the action and θ^- for evaluating the action [73]:

$$y_j = r_j + \gamma Q(s_{j+1}, \arg\max_{a'} Q(s_{j+1}, a'; \theta); \theta^-)$$

Double DQN has shown an improvement in performance and stability of the NN.

Prioritized Experience Replay In DQN, experience transitions are uniformly sampled from the replay memory, regardless of the significance of the experiences. Schaul et al. proposed to prioritize experience replay, so that important experience transitions can be replayed more frequently, to learn more efficiently [58]. The importance of experience transitions are measured by TD errors. The authors designed a stochastic prioritization based on the TD errors, using importance sampling to avoid the bias in the update distribution. The authors used prioritized experience replay in DQN and D-DQN, and improved their performance on Atari games and vastly reduced the time to convergence of the models.

Dueling DQN Another improvement suggested in 2016 by Wang et al. for faster convergence is the decomposition of the Q-value into the value function of the state and an advantage function of the action:

$$Q^\pi(s, a) = V^\pi(s) + A^\pi(s, a)$$

In DQN, a CNN layer is followed by a fully connected (FC) layer. In dueling architecture, a CNN layer is followed by two streams of FC layers, to estimate value function and advantage function separately; then the two streams are combined to estimate the Q-value function. For better convergence, Wang et al. estimated also how much the advantage of the action was greater than the average advantage:

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a=1}^{|\mathcal{A}|} A(s,a)$$

By decoupling the estimation, intuitively the Dueling-DQN can learn which states are (or are not) valuable without having to learn the effect of each action at each state since it is calculating $V(s)$ separately.

Additional DQN Improvements Further extensions and improvements were published in recent years that might be useful for our research, such as the one of Anschel et al. that proposed to use an average of previous Q-values estimates to reduce variability and instability of the NN [2]. To propagate rewards faster and accelerate DQN, He et al. introduced a constrained optimization approach that also managed to improve the accuracy of the NN [27]. Some works tried to improve the exploration strategy of the agent in the DQN framework, such as the one of botstrapped-DQN by Osband et al. [50] and the "noisy nets" approach of Plappert [53].

Policy Gradients in DRL

In this subsection we discuss the methods for optimizing the policy of the agent directly, with deep NN as the estimators of the policy function which maps states to actions. The main advantages of policy methods over DQN

are its better convergence properties, its effectiveness in high-dimensional or continuous action space and its ability to learn stochastic policies.

As explained above, the most basic form of the policy gradients method was first introduced with the Monte-Carlo policy gradient called REINFORCE algorithm [77]. The approach aims at maximizing the objective function that is defined as:

$$J(\theta) = R_t = \mathrm{E}[\sum_{t}^{\infty} R(s_t)|\pi_\theta] = \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi_\theta(a_t|s_t) R_t$$

Where $d(s)$ is the distribution over states. The best approach for the optimization process of the parameters θ is gradient ascent with a policy gradient $\nabla_\theta J(\theta)$ taking a step in its direction controlled by a learning rate α . The policy gradient can be computed with the help of likelihood-ratios [67]:

$$\mathrm{E}[\nabla_\theta \log \pi_\theta(a_t|s_t) R_t]$$

Instead of computing R_t directly at the end of each episode we can estimate it by $Q^{\pi_\theta}(s,a)$ [69]. In REINFORCE they use V_t as an unbiased sample of $Q^{\pi_\theta}(s,a)$ and thus update the parameters in each episode by:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) V_t$$

The policy in this algorithm can be represented as a stochastic distribution such as softmax or gaussian distribution, and can also be represented by a deep NN with a stochastic or deterministic output layer.

Actor-Critic algorithms In Actor-Critic algorithms a *critic* is used to estimate and update the action-value function and an *actor* is used to update

the policy function for reducing the high variance of the Monte-Carlo policy gradient [68]. Two sets of parameters are maintained - w for the critic and θ for the *actor*. Intuitively, the *critic* is supposed to give the *actor* which determines the policy a feedback and a direction for improvement. The *critic* in Actor-critic algorithms follow an approximate gradient:

$$\mathrm{E}[\nabla_\theta \log \pi_\theta(a_t|s_t) Q_w(s_t, a_t)]$$

This approximation of the gradient has proven to be correct and unbiased with the compatible function approximation theorem [69]. Both w and θ are now updated in each step of the algorithm:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) Q_w(s_t, a_t)$$

$$w \leftarrow w + \beta(R(s_t, a_t) + \gamma Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, a_t)) \nabla_w Q_w(s_t, a_t)$$

A2C and A3C Both methods are actor-critic implementations which are also using an *advantage-function*. As we explained before, for reducing the variance of the model we subtract a baseline $b(s_t)$ function from the policy gradient to form an advantage function $A(s, t)$. It can be easily proven that using a baseline which is dependent only on the state does not introduce bias and does not change the gradient expectation [25]. The most common baseline used is the value function $V(s)$. So, considering an estimation of the action-state value $Q_w(s, a)$ using weights w and an estimation of the value function $V_v(s)$ using weights v we can define the advantage by:

$$A(s, a) = Q_w(s, a) - V_v(s)$$

Both Q and V can be updated with TD learning. Instead of updating both estimators' parameters in addition to the policy's parameters it is possible to update the advantage function estimator directly with the TD error of the value function so we will have only two sets of parameters in the model, since:

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) = r + \gamma V(s_{t+1}) - V(s_t)$$

It is also helpful to apply the TD-lambda (i.e., eligibility traces) method [67] for the value-function approximation.

Asynchronous Advantage Actor Critic (A3C) is an algorithm proposed in 2016 by Mnih et al. [45]. In A3C, we asynchronously execute different agents in parallel on multiple instances of the environment. Each worker (copy of the network) will update the global network asynchronously, so that experience replay is not utilized. Different from most deep learning algorithms, asynchronous methods can run on a single multi-core CPU. For Atari games, A3C ran much faster yet performed better than or comparably with DQN extensions and also succeeded on continuous motor control problems. Advantage Actor Critic (A2C) use a similar approach but it waits until all workers have finished their training and calculates their gradients to average them and then to update the global network synchronously.

In 2017, Wang et al. proposed a stable and sample efficient actor-critic deep RL model using experience replay, with truncated importance sampling, stochastic dueling network and trust region policy optimization (that will be introduced in the following paragraph) [75].

Trust Region Policy Optimization (TRPO) In 2015, Schulman et al. introduced the TRPO algorithm which use their method for optimizing control policies, with guaranteed monotonic improvement in optimization of a surrogate objective function [59]. The authors have made several approximations, including: introducing a trust region constraint defined by the KL divergence between the new policy and the old policy so that at every point in the state space, the KL divergence is bounded; approximating the trust region constraint by the average KL divergence constraint; replacing the expectations and Q-value in the optimization problem by sample estimate; and, solving the constrained optimization problem approximately to update the policy's parameter vector. A much simplified pseudo-code of TRPO is presented in Alg. 3. The authors have suggested

Algorithm 3: Trust Region Policy Optimization

for *iteration in 1 to M* **do**
  Run policy for T timesteps or N trajectories ;
  for *timestep t in 1 to T* **do**
    Estimate advantage function at all timesteps ;
    $\max_{\theta} \sum_{n=1}^{N} \frac{\pi_\theta(a_n|s_n)}{\pi_{\theta_{old}}(a_n|s_n)} \hat{A}_n$
    subject to $\overline{KL}_{\pi_{\theta_{old}}} \leq \delta$;
  end
end

using conjugate gradient for solving the constrained optimization problem efficiently. They also showed that policy iteration, policy gradient and natural policy gradients [33] are all special cases of TRPO. A small variation of TRPO is Proximal Policy Optimization (PPO) that uses a KL penalty instead of constraint [60]. PPO is a bit better than TRPO on continuous control and much better on Atari. It is also more compatible with multi-output networks and RNNs.

DRL in Large Actions Spaces.

The main challenge in applying DRL to pipeline generation is the large number of possible actions – a number that continuously grows as pipelines become complex. Few works proposed solutions to environments with large discrete action spaces. Dulac et al. suggested a generative model that performs embedding to create dense action representations in a countinuos space [20]. They used these embeddings in the settings of an Actor-Critic agent, where the actor outputs an approximate action in the continuous space. They then choose k most similar actions using k-nearest neighbors and then a final one out of them according to the values approximated by the critic. Another, more recent work uses actions embedding to learn latent representations that improve generalization over large, finite action sets by allowing the agent to infer the outcomes of actions similar to actions already taken [11].

Another work by Zahavy et al. used a solution of a dedicated sub-network for action-elimination [78]. There solution consists of a NN, in addition to a DQN agent's Q-network, that its purpose is to "learn what not to learn", by identifying and eliminating sub-optimal actions. This network is trained to predict invalid actions, supervised by an external elimination signal provided by the environment.

While effective, all the approaches presented above require the deployment and fine-tuning of additional deep-learning networks. Moreover, the proposed solutions are only applicable to specific types of DRL algorithms. We take inspiration from some of the described approaches, for example the embedding and clustering suggested by [20], however, our proposed approach is much easier to implement and integrate with ex-

isting algorithms, transparent to the DRL agent, and can be applied on almost any type of DRL algothm.

3 Research Goal

As machine learning tools become more popular, the demand for experts in the field increases accordingly. We argue that while some ML tasks are complex and require human expertise, other more classic tasks can become automated, a thing that might ease the demand for human experts. While previous works have tried to achieve automation in the context of ML, most approaches require great deal of computational resources and time (see Related Work). In our research, we try to advance the field of AutoML towards more accurate automatic methods that require much less computational cost and runtime. To that end we define the following objectives:

- **Establish a fully automated framework** for solving classic machine learning tasks such as classification and regression on tabular datasets of different types and complexities. The system should take as input a dataset, a task and a metric and produce an end-to-end machine learning model as a sequence of algorithms, starting from data preprocessing, feature extraction and feature engineering, and ending with a prediction model. The system should learn to match datasets with the algorithms that will produce the best score over the given metric.

- **Utilizing the recent advances in reinforcement learning** (RL), and

specifically deep reinforcement learning (DRL) algorithms, for the task of searching and discovering the optimal algorithms and ML pipeline architectures.

- **Formulating the task of AutoML as a sequential decision making problem** that could be naturally handled by different DRL algorithms. The formulation should adhere to the required assumptions of RL, e.g., to sustain the Markov property.

- **Designing and constructing an RL environment** that will best translate the sequential decision making formulation to a real-world RL system. The environment should allow for an easy search of the ML pipeline's best architectures and algorithms. Our main goal is to design an environment that implements a trade-off between an unlimited search space that enables the formulation of any ML pipeline architecture, and a simple constrained search space which is easy to implement but with the cost of sub-optimal solutions.

- **Dealing with a Large and Dynamic Search Space**. The AutoML task constitutes a formidable challenge to existing DRL methods, which have difficulties operating in large and complex search spaces and in real-world problems. Our goal is to establish a method that will overcome the current limitations of DRL algorithms in these settings.

- **Online use for unseen before datasets**. Implementing Meta-Learning approaches as a part of the DRL model, in the intention of enabling good generalization across multiple datasets and ML models. The integration of meta-learning approaches would allow the model to learn offline the dataset-model interaction so that it could be deployed online for unseen before datasets.

We currently limit our research to tabular datasets and classification tasks. While this is a limitation of the research' scope we argue that good results in this limited scope may imply the same behavior in other domains of ML.

4 Research Contributions

In the following chapters we present and evaluate DeepLine, a novel approach for automatic ML pipeline generation that achieves comparable performance to state-of-the-art methods with significantly less computational cost. Our approach introduces the following advances in the field of AutoML and Deep Reinforcement Learning.

DeepLine is able to learn offline across multiple datasets and ML pipelines using advanced DRL algorithms and meta-data representation of both the datasets and the ML pipelines. This ability means that unlike most previous AutoML models, DeepLine can be applied online on unseen before datasets, utilizing past experiences gathered in the offline phase to efficiently generate ML pipelines with no additional optimization.

We present a novel grid-world representation for the ML pipeline generation task. This representation implements the desired trade-off between highly complex and unconstrained search space of ML pipelines and a constrained space that allows for a single pipeline architecture. The grid-world environment enables an efficient and manageable pipeline generation process while accommodating the creation of relatively complex pipeline architectures.

We propose a novel action-modeling approach that we refer to as the *Hi-*

erarchical Step. The purpose of the Hierarchical Step is to solve the under-researched problem of large dynamic action spaces, i.e., discrete action spaces with many actions that are invalid in most states. As almost all DRL algorithms require a fixed-size actions space, our approach can integrate with any algorithm to enable it to operate in environments with dynamic action spaces by creating a fixed-size representation to model the dynamic action spaces. This solution also speeds up the training convergence of the DRL agent by enforcing a better and more inclusive exploration process.

Finally, we propose three different methods for integrating the Hierarchical Step with a DQN agent in a smooth and convenient implementation that does not require any significant effort and changes in the original DQN algorithm. We also present a unique online pipeline exploration process, using the characteristic of the DQN algorithm.

We conduct an extensive evaluation on 56 datasets and show that DeepLine achieves comparable or better results than state-of-the-art methods at a fraction of the computational cost.

5 Research Problem Formulation

Unlike some of the previous work on autoML that only focused on the automation of specific steps of the ML model, such as automatic feature generation and feature extraction, we define the problem as an end-to-end process integrating the entire model's steps, same as in TPOT [49], alphaD3M [19], auto-sklearn [21] and auto-weka [70]. To adhere the same notations and definitions from previous works, especially [49] and [19], we will formulate the general problem in a similar manner.

We define a *learning job* $\mathcal{L}(D, T, M)$ consisting of a tabular dataset D of m columns and k instances, a prediction task T and an evaluation metric M. Additionally, we define *primitives* as any type of machine learning algorithm (e.g., preprocessing, feature selection, classification) and denote the set of primitives as $\mathcal{P}_r = \{p_1, p_2, ..., p_{N_p}\}$. We define a directed acyclic graph (DAG) of primitives $G = \{V, E\}$, where $V \subseteq \mathcal{P}_r$ are the vertices of the graph, and E are the edges of the graph and determine the primitives' order of activation. We denote G as a ML pipeline, or *pipeline* in short. Given that G is able to produce predictions over the specified learning job, our goal is to generate a pipeline G as to achieve

$$\arg\min_{G} \mathcal{E}(\mathcal{L}(D, T, M), G)$$

where $\mathcal{E}$ is the error of pipeline G over the learning job $\mathcal{L}$.

To reduce the size of our search space, we consider the problem of generating G as a sequential decision making task, and generate the pipeline step-by-step. We further formalize the problem as a Markov Decision Process (MDP) defined by the tuple $\mathcal{M}(\mathrm{S}, \mathrm{A}, \mathrm{P}, \mathrm{R}, \gamma)$, where S is the set of all possible states (i.e., pipeline and learning job configurations), A is the set of all possible actions defining the transitions between states, P is the (deterministic) transition function between states, and R is the reward function which is directly derived from the pipeline score with metric M. We also use the rewards discount factor $\gamma \in (0, 1]$.

Given the expected sum of discounted rewards $R_t = E_\pi[R(S_0) + \gamma R(S_1) + \gamma^2 R(S_2) + ...]$, produced by a policy $\pi : \mathrm{S} \rightarrow \mathrm{A}$, our goal is to obtain:

$$\arg\max_{\pi} R_t$$

We now use this formalization to construct a reinforcement learning environment and an agent for the sequential decision making problem, assuming the Markov property is satisfied. In the following section we detail more specifically the exact methods for representing the MDP and explain how we solve this problem through the application of reinforcement learning.

6 Methods

Overview. Our framework is presented in Figure 6.1. As in all RL problems, it consists of an *environment* that models the problem and defines the state and action spaces, and an *agent* that interacts with it. However, we introduce a third component, the *Hierarchical Step Plugin* that mediates the interaction between the agent and the environment and helps overcome the large and complex actions space of the pipeline generation problem.

The environment models the pipeline generation process in a semi-constrained setting of a grid-world, where each cell acts as a placeholder for a primitive and each column of the grid is designated to a single type of primitives (e.g. feature selection). By placing these constraints, we are able to enforce a logical order of primitives in the pipeline, and reduce the size of the search space. Since the grid-world has multiple rows, it accommodates the generation of relatively complex and inter-connected pipelines with edges between primitives in different rows, such as the pipeline in Figure 6.2.

In the grid-world environment, the state space represents the current state of the pipeline and the actions space is defined as all the actions that can alter the pipeline in the settings of the grid. The problem with this actions space definition is the fact that the number of actions that can alter the

pipeline in each state is very large, with many of these actions being *invalid* and thus redundant in most states due to our enforced constraints. Our proposed solution consists of two parts: the creation of an actions open list, and the hierarchical-step.

The *actions open list* is produced by the environment in each state, and consists only of the *valid* (i.e., allowed) actions in the current state, thus significantly reducing the actions space. However, the changing number of actions between states prevents us from using most types of DRL algorithms, as they require a fixed actions space (i.e., fixed number of outputs in their output layer). We therefore propose a method to efficiently explore all possible actions in a fixed-size setting by partitioning the dynamic actions space into smaller sub-spaces.

The *hierarchical-step plugin* groups the actions of the open list into fixed-size clusters of n actions each, and iteratively queries the agent to select a single action (a_c) from each cluster ($\overline{A}_c$). The winning actions of all clusters are regrouped and clustered once more, and the process is repeated until a final set of n actions remains. Note that up to this point, the agent selected multiple actions but never executed them. The agent receives the final set of n actions and executes one chosen action (a_f). The executed action alters the environment (pipeline) and a new open list of actions is generated. The overall process is then repeated.

6.1 The Environment

The main challenge we faced in defining the environment was the need to restrict the size of our state and action spaces while maintaining the

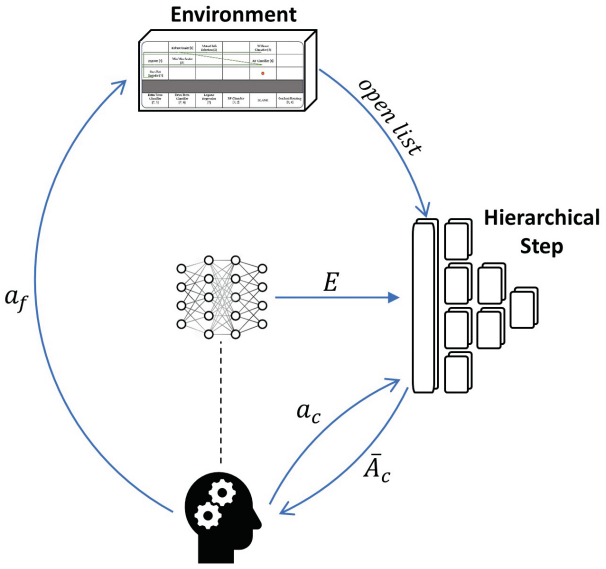

Figure 6.1: Schematic representation of our framework. The interaction between the environment (top) and the agent (bottom) is mediated by the hierarchical-step plugin (right).

ability to produce effective and complex pipelines. Our proposed solution consists of two parts: (1) the partition of the ML primitives into *families*, and; (2) a *grid-world* representation of the pipeline that restricts both the state space (i.e., the pipeline size) and action space.

6.1.1 Primitive Families

Human data scientists often apply a two-step decision process: they first determine whether a certain type of algorithm (e.g., feature selection) is required, and if the answer is positive they attempt to determine which specific algorithm. We opt for a similar approach by grouping our primitives into the following families:

- **Data Preprocessing**: data cleaning, feature encoding, etc.;

- **Feature Preprocessing**: feature discretization, scaling and normalization;
- **Feature Selection**: uni-variate selection, entropy-based selection etc.;
- **Feature Engineering**: data enrichment, dimensionality reduction etc.
- **Classification and Regression Models**: XGBoost, Random Forest, lasso regression, etc., and;
- **Combiners**: consisting of the same algorithms as family 5, but only a single member can exist in each pipeline. The combiner acts as a meta-learner: it receives as input all the outputs of members of family 5, thus ensuring that each pipeline has a single output.

As we later explain in detail, at any time-step our agent can only select from a single family of primitives. By doing so we reduce the sizes of both the state space and the action space of the agent.

6.1.2 Grid-world representation of the pipeline

We design our pipeline representation based on two common practices used by data scientists and academic studies. The first is the specific order that primitive families are applied in a pipeline (e.g., pre-processing $\rightarrow$ feature selection $\rightarrow$ classification) and the second is the union of inputs from two or more sub-pipelines into a single pipeline (e.g., in [28]).

As depicted in Figure 6.2, our state space is defined as a Nx6 grid, where N denotes the number of rows and 6 is the number of columns, one for each of the six primitive families. Each cell of the grid is a placeholder for a primitive. The cells of a given column can only contain members of that

column's assigned primitives family. Grid cells can be left empty, meaning that a pipeline doesn't have to contain all types of primitives.

The output of a grid cell is automatically passed to the subsequent non-empty cell in the same row, allowing each row to function as a sub-pipeline. For example, in Figure 6.2 we see that the Mutual Information primitive (step #2) is connected to the XGBoost classifier (step #3) since the Feature Engineering cell was left empty. Additionally, the output of a cell can be passed to additional cells in other rows, thus creating inter-connected pipelines. A cell can be connected to any other cell under two conditions: (i) that a cell's input is *valid*, with predefined rules (see the *Open List of Actions* sub-section), and (ii) the column index of the target cell is equal or larger than that of the source cell. Every primitive may receive multiple inputs, in which case all inputs are merged and duplicate columns are removed. An example of this is seen in Figure 6.2, where the RF classifier (step #6) receives inputs from two cells (steps #1 and #5).

We further reduce the size of the state space by constraining the transition between states. Only a single grid cell – marked by the dot cursor in Figure 6.2 – can be updated at any time step. Additionally, the cursor performs only a single pass on the grid in each episode, completing one row in the grid before moving to the next, making the process more manageable and efficient. In addition to making the pipeline generation challenge more manageable, this approach significantly improves running time.

The Combiner Column

Our representation of the environment may result in multiple prediction primitives (algorithms), each producing their own classification predic-

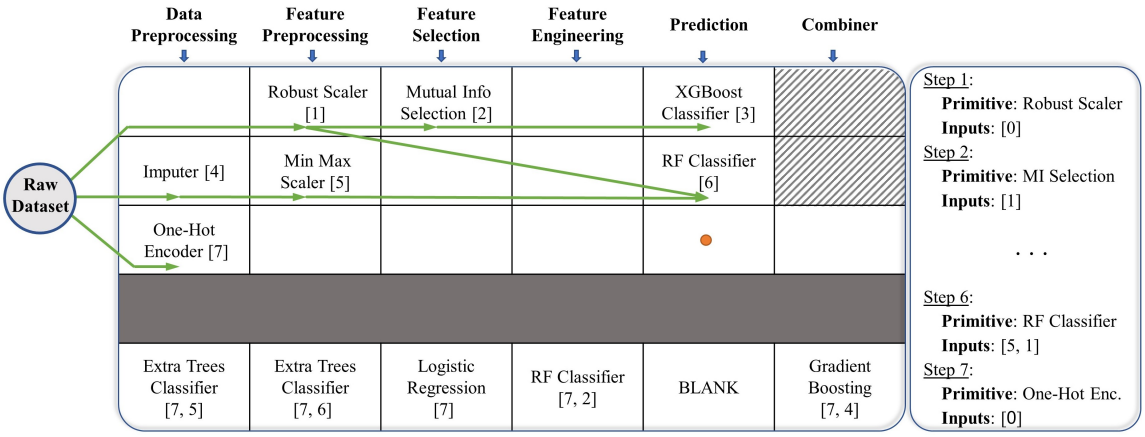

Figure 6.2: **(Left)** Example of the grid-world environment with N=3. The grid currently populates a pipeline with 7 steps which has 8 edges and 8 vertices (one for the raw dataset). Each column is assigned a different primitives family. In each episode, the dot cursor transitions from cell to cell, passing through all the cells of a row before moving to the next one. In each cell, the agent chooses either to leave it empty or to populate it with a *pipeline-step*. When the cursor leaves the last cell, the resulting pipeline is evaluated over the given dataset and the score is used as the reward. **(Left-Bottom)** The bottom row visualize the possible actions of the current cluster, with n=6. Each cell in this row represents a different candidate *pipeline-step* that the agent can choose to populate in the cell marked by the cursor. When the cursor is in either of the two darkened cells the agent can only choose to continue or to skip to the last cell of the grid. **(Right)** The translation of the grid into a sequence of pipeline-steps.

tion. Moreover, it is possible for our agent to produce a linear pipeline without a prediction primitive. To produce a single unified output for the entire pipeline, we added the Combiner column which receives the outputs of all linear pipelines and can be populated by primitives from the Combiner family.

The Combiner column consists of a single cell, and it has to be populated if two or more disjoint pipelines exist. Consider the example in Figure 6.2: since the presented grid contains two prediction primitives (#3 and #6) as well as a data preprocessing primitive (#7) without any connections,

the Combiner column will be populated by an primitive that receives the output of the three primitives as input.

6.1.3 State Representation

The environment's state-space is purposed to represent the current state of the pipeline and the current form of the dataset, after it has been processed by the pipeline's primitives. We model the pipeline using a simplified graph representation in the settings of the grid-world, and the dataset by dataset-based features that will be subsequently discussed.

Although our generated pipelines may vary in width and depth, we create a fixed-size representation that can be easily used as input for a neural network (NN). We now elaborate on all the components that comprise the state vector:

- $\overline{G}_p$ – represents the grid as a sequence of primitives. Concatenation of one-hot encoded vectors where each primitive is assigned an encoding value (blank cells have a separate encoding), so the length of this vector is $N_{cells}X|\mathcal{P}_r|$, where N_{cells} is the number of grid cells and $\mathcal{P}_r$ is the set of all primitives. This vector is sparse, and we create an embedding to compress it.

- $\overline{G}_{in}$ – represents the incoming edges (i.e., inputs) of each cell in the grid. The length of this vector is $N_{in}XN_{cells}$, where N_{in} is a configurable parameter defining the maximal number of inputs per cell. For example, in Figure 6.2, the cell containing the RF classifier (#6) has two inputs (#1 and #5). If $N_{in} = 3$, then the vector entry for this grid cell would be $\{1, 5, -1\}$.

- $\overline{P}_m$ – general meta-data describing the pipeline's topology: number of nodes and edges, graph centrality etc.
- $\overline{O}_m$ – dataset-based features, representing the *data being processed by the current grid cell*. This distinction is important because each cell "sees" a different dataset, based on its input(s). For example, step #6 in Figure 6.2 considers the dataset to be the concatenated outputs of steps #1 and #5. The values of this vector include the number of features, number of instances, % of numeric features etc.
- $\overline{L}_j$ – concatenation of: (1) a *task* vector, with one entry for each possible task (e.g., regression, classification, etc.), (2) a *metric* vector, with one entry for each possible metric (e.g., accuracy, AUC, etc.) and (3) dataset-based features vector of the raw (original) dataset, similarly to $\overline{O}_m$.
- $\overline{A}_c$ – a vector representing the available actions. Each action is represented by a vector that details both a candidate primitive to the current grid cell, in addition to its connection(s) to the cells that provide its input. Each time step, n actions are available for choice, as shown in the bottom row of figure 6.2. All n action vectors are concatenated. We elaborate on this further in the next section.

Dataset Meta-Features

Both $\overline{O}_m$ and $\overline{L}_j$ vector components rely on dataset-based features, which can also be referred to as *Meta-Features*. The purpose of these features is to capture the essence of the dataset in a way that will allow for meta-learning models to learn which ML algorithm suits the most each dataset, as we want to do in our work.

To create our dataset meta-features we build upon the previous work of [36] and [72] who successfully used dataset-based meta-features for AutoML related tasks. Our meta-features combine elements from both studies and can be divided into two groups:

- **Descriptive.** Used to describe various aspects of the dataset. This group of meta-features includes information such as the number of instances in the data, number of attributes, percentage of missing values and likewise.
- **Correlation-based.** Used to model the interdependence of features within the analyzed datasets. Meta-features of this group include correlation between different attributes and the target value, Pearson correlation between attributes and different aggregations such as average and standard deviation (among others).

6.1.4 Open List of Actions

In the previous section we described how we restrict our state space by allowing only a single cell to be updated at any given time step. We now describe how we generate the set of possible actions for each time step, reducing also the action space of our environment. We denote this set as the *actions open list*.

Let *pipeline-step* be defined as the combination of (1) a primitive and (2) its inputs, i.e., the cells that their output is fed into the primitive. In addition let *open list* be a list of all candidate *pipeline-steps* to populate the cell currently marked by the cursor. The action space is defined as $\mathcal{A} \in \{1, ..., |open\ list|\}$. Each action prompts the insertion of the corre-

Algorithm 4: Actions Open List

Input: N_{in} - max inputs allowed, (r, c) - cursor's coordinates where $r \in 1, ..., N$ and $c \in 1, ..., 6$, l-column index of last populated cell in row r which is not "blank", G-the grid world, $\mathcal{F}_{r,c}$ - set of family primitives associated with cell $G_{r,c}$

Output: *open list*

Function `OpenList(`G, N_{in}, r, c, l, $\mathcal{F}_{r,c}$`)`**:**

- $open\ list \longleftarrow \emptyset$
- `// Generate previous Outputs set`
- $\mathcal{O} \longleftarrow \{O_{i,j} | i \in [1, r), j \in [1, c], G_{i,j} \neq "blank"\}$
- $\mathcal{O} \longleftarrow \mathcal{O} \cup O_{r,l}$
- `// Generate input candidates set`
- $\mathcal{I} \longleftarrow \{s | s \subset \mathcal{O}, |s| \leq N_{in}, O_{r,l} \in s\}$
- **for** $primitive \in \mathcal{F}_{r,c}$ **do**
  - **for** $inputs \in \mathcal{I}$ **do**
    - **if** $CanAccept(primitive, inputs)$ **then**
      - $s \longleftarrow PipeStep(primitive, inputs)$
      - $open\ list \longleftarrow open\ list \cup s$
    - **end**
  - **end**
- **end**
- $open\ list \longleftarrow open\ list \cup "blank"$
- **return** *open list*

End Function

sponding *pipeline-step* from *open list* to the cell. We now describe how we generate the *open list* in each time-step, reducing the environment's action space.

The list of possible actions for a given cell is determined by two elements: (1) the primitive family assigned to the cell, and; (2) the set of possible candidate inputs, out of all the grid cells' outputs. The process for generating the *open list* is presented in Algorithm 4. We begin by creating the set of all possible combinations inputs, denoted by $\mathcal{I}$. While a cell C_{rc} in row r and column c always receives as input the output of the most recently

populated cell in row r, it may also receive up to $N_{in} - 1$ additional inputs from any other previously populated cell with an equal or smaller column index in previous rows. For example, step #6 in Figure 6.2 will always receive input from step #5, but may also receive inputs from step #1, #2 and #3. step #4 cannot be an input for #6 because it precedes #5, and #7 also cannot be used because it is populated at a later time step.

Once we created the set of all possible inputs, we match every item in this set to all the members of the cell's assigned family primitives (lines 6-10 in Algorithm 4). The validity of each combinations is then examined, with possible reasons for elimination including inability to process categorical features, missing entries, or negative values. All valid combinations are retained and the *blank* action which leaves a cell empty is also added to make up the *open list*.

6.2 Hierarchical-Step Plugin

While our actions open list narrows the search space, the fact that it is *dynamic in size* (i.e., the number of valid actions changes from cell to cell) makes it incompatible with most DRL algorithms that require fixed actions space. For example, policy gradient methods such as TRPO [59] output a vector of distribution over actions and DQN methods [46] output the q-values vector with one entry for each single action in the action space, given the current state. This means that the underlying action behind each action index should remain the same during all time steps.

While solutions to this problem exist in the literature, they are not without their shortcomings. One solution that is often taken in that situation is

padding. In this approach, all unique actions (in our case, all possible pipeline-steps) are mapped with constant action indices. Choosing invalid actions, i.e., actions that stand for pipeline-steps that are not in the *open list*, will be penalized with a negative reward. The main disadvantage of this solution is the large size of the actions vector, which will make training the agent slow and difficult. The agent would put most of its effort in differentiating valid actions from invalid actions, instead of focusing on learning the optimal pipeline-steps it should select given the current state.

Another option is to calculate the Q-value of every state-action pair in each iteration, but this approach is both computationally expensive and only applicable to some RL methods. This solution inflates the number of required forward passes (and back-propagations) of the NN in each of the agent's iterations, increasing significantly the computational cost.

Other, more advanced methods are detailed in the Related Work section, where we explain why most solutions are computationally expensive and only applicable to some RL methods. Therefore, we propose our hierarchical approach for dynamic actions modeling.

6.2.1 Hierarchical Representation of Actions

Our goal is to enable a DRL agent to evaluate a varying number of actions using a fixed-size representation. We partition the action space (i.e. *open list*) into smaller sub-spaces, each consisting of exactly n actions, and filter the actions in a hierarchical process which is presented in Algorithm 5.

First, we create a vector representation for each action in the *open list* and stack all the vectors in matrix A_{open}, so that each row holds a vector. In our case, each action vector represents a pipeline-step and comprised of *(a)* an

Algorithm 5: Hierarchical Step

Input: A - matrix containing dense vector of each action of the *open list*; $\overline{S}$ - current state vector (without actions vector); n - size of each cluster

Output: $\overline{S}_c, a_f$-final action , $\overline{S'}$-next state, reward, done

1 **Function** `HierarchicalStep(`$A, \overline{S}, n$`)`**:**
2 $A^h \longleftarrow \emptyset$
3 $Clusters \longleftarrow MakeClusters(n, A)$
4 **foreach** $C \in Clusters$ **do**
5 $\overline{A}_c \longleftarrow (A[C_0], A[C_1], ..., A[C_n])$
6 $\overline{S}_c \longleftarrow (\overline{S}, \overline{A}_c)$ `// Concatenation`
7 $a_c \longleftarrow AgentActionSelection(\overline{S}_c)$
8 $A^h \longleftarrow A^h \cup A[a_c]$
9 **end**
10 **if** $length(Clusters) = 1$ **then**
11 $a_f \longleftarrow a_c$
12 $\overline{S'}, reward, done \longleftarrow EnvStep(a_f)$
13 **else**
14 $HierarchicalStep(A^h, \overline{S}, n)$
15 **end**
16 **return** $\overline{S}_c, a_f, \overline{S'}, reward, done$
17 **End Function**

embedded representation of the primitive, as explained in the following section, and *(b)* the indices of its input cell(s).

We cluster A_{open} to $\lceil |A_{open}|/n \rceil$ clusters that have exactly n actions each (line 3). Our agent iterates over the clusters, selecting a single action (a_c) from each cluster (lines 4-8). The resulting set of chosen actions A^h is then clustered again and the process is repeated until a single cluster of n actions remains. The action chosen from this set (a_f) is the one actually carried out by the agent. Note that we pad clusters that do not have exactly n actions with *invalid* actions whose selection incurs a negative reward.

Figure 6.3 depicts an example of the process. In this example our open

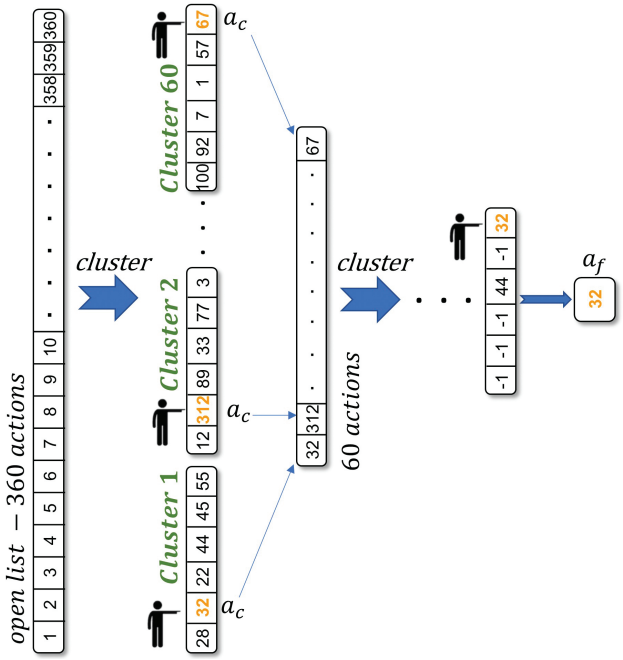

Figure 6.3: Example of the hierarchical step process, with n=6 actions in each cluster and an initial open list with 360 actions.

actions list consists of 360 possible actions. given that $n = 6$, the top level of the hierarchy is split into $360/6 = 60$ clusters, from which 60 actions are selected. These 60 actions are clustered again into ten clusters, then two (each padded with one invalid action) and then finally one from which the final action is selected.

From the agent's perspective, it only sees a single cluster at each time-step, represented by vector $\overline{A}_c$ which is added to the state vector in line 6 of the algorithm (see Figure 6.2 Left-Bottom). The weights of the agent's policy network are only updated with respect to the execution of action a_f but are fixed during the lower levels of the hierarchical process.

Clustering Method

The *MakeClusters* method (line 3) creates the clusters using a variation of

Algorithm 6: MakeClusters

Input: A - matrix containing dense vector of each action of the *open list*; n - size of each cluster
Output: Fixed-Size Clusters

```
1  Function MakeClusters(A, n):
2      Use K-Means to compute ⌈|A_open|/n⌉ centroids of the action
         vectors in A;
3      Assign n actions to each cluster using a the order of distance to
         their nearest cluster minus distance to the farthest cluster;
4      Assign points to their preferred cluster until this cluster is full,
         then resort to second best cluster, etc.;
5      pad clusters with -1s if needed;
6      Compute current cluster means using the action vectors in A;
7      Sort elements based on the delta of the current assignment and
         the best possible alternate assignment;
8      foreach Element By the computed Order do
9          foreach cluster which the element does not belong to do
10             If there is an element wanting to leave the other cluster
                 and this swap yields and improvement, swap the two
                 elements;
11             If the element can be moved without violating size
                 constraints, move it;
12         end
13         If the element was not changed, add to outgoing transfer list;
14     end
15     If no more transfers were done (or max iteration threshold was
         reached), terminate;
16 End Function
```

the popular K-Means clustering algorithm [26]. The algorithm is applied at each level of the hierarchy, with the number of clusters determined by A_{open} and n. Our approach has one significant difference from the standard algorithm, since we require not only a fixed number of clusters but

also a fixed number of samples in each cluster.

The fixed-size clustering approach first uses K-means to derive centroids of actions and then assigns the same amount of actions to each centroid (i.e., cluster). In the proposed approach, which is presented in Algorithm 6, we first initialize the clusters in the following way: the actions are ordered by their distance to the closest centroid, subtracted by the distance to the farthest centroid. Each point is assigned to the best cluster in this order, in a best-over-worst assignment technique. If the best cluster is fully occupied, the second best is chosen, etc.

After the clusters initialization we perform an elements swapping technique that its objective is to minimize the variance in the clusters. Note that a swap is carried out only if it yields gain, i.e. reduces the variance which is computed by the average distance of all vectors of the cluster to its centroid. We use the Euclidean distance metric throughout the algorithm.

Hierarchical Step Contributions

An important strength of the proposed approach is the fact that the hierarchical process is transparent to the agent. In addition to being compatible with a variety of popular DRL approaches, including actor-critic methods such as A3C [45] and the different variations of DQN, DeepLine places no limitations on the use of exploration methods such as ϵ-greedy or techniques such as prioritized experience replay [58], which we use in our model. Furthermore, we implement the hierarchical step as a part of our environment in compliance with OpenAI-Gym's settings, suitable for use with any DRL agent.

In addition to the fact that the *hierarchical step* solves the problem of dynamic action spaces, it also introduces a significant improvement to the agent's exploration properties. When using the plugin, the agent is coerced to visit and evaluate sub-spaces of the actions space that otherwise it would be prone to overlook. As we show in the evaluation section, the hierarchical approach outperforms a model with no hierarchical plugin, where the agent has to learn the environment with all possible unique actions at every time-step.

Visualization

We created a simple visualization to render the environment when combined with the hierarchical step. The render plots the pipeline generation process step by step by visualizing the current pipeline, the possible n actions in the current cluster, the selected action and the obtained reward of the previous step. The purpose of this visualization is to enable an easy track of the agent's decisions along the way and to establish an understanding of the resulting pipelines. An example of the render under a random agent (agent which selects random actions) can be seen in a video provided in **this link**.

6.3 DRL Agent

We implement our agent using the DQN algorithm, which is an off-policy algorithm. While on-policy algorithms such as policy gradients are generally more stable, they are also less sample-efficient and prone to converge to a local optimum. Moreover, while on-policy approaches generally out-

perform off-policy approaches in large action spaces, our hierarchical representation of actions makes this point irrelevant.

6.3.1 DQN with Hierarchical Step

We have developed three different versions of our DRL agent, all heavily rely on the basic DQN algorithm but with some minor changes. The difference between the versions lies on the method of backpropagation of the NN through the Hierarchical Step.

Under a DQN agent, the actions selected from each cluster of the Hierarchical Step are chosen by the *AgentActionSelection* method (see Algorithm 5 - line 7), which implements the ϵ-greedy exploration algorithm. In this algorithm a random action is selected under probability ϵ or otherwise the action is selected by $\arg\max_a Q(s_t, a|\theta)$.

Let the tuple (s_t, a_t, r_t, s_{t+1}) be defined as a *transition* which is obtained in a standard step of the environment. When using the Hierarchical Step, only the action a_f that is selected in the final hierarchy (from its single cluster of actions) transitions the environment from the current state to the next with a reward. We refer to this transition as the *final transition*. However, we consider the actions selected from lower hierarchies and clusters of the Hierarchical Step as *inner-transitions*. An inner-transition is defined by the action selected from the current cluster a_c along with the current state s_t and the next state s_{t+1} which is reached by the final transition of the Hierarchical Step, along with the reward r_t. We now use these notations to describe the three versions of the agent.

The first and most basic algorithm for training the agent is identical to the DQN algorithm that was published in [46] and presented in Algorithm 2.

The one main difference: our use of the hierarchical-step plugin, which replaces the application of a conventional environment step, as seen in Algorithm 7, line 4. This method's main strength is the fact that the application of the Hierarchical Step is virtually transparent to the architecture of the DQN agent and can be integrated with almost no effort.

Algorithm 7: DQN with Hierarchical Step

Initialize replay memory $\mathcal{D}$;
Initialize action-value function Q with random weights θ;
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$;
1 **for** *episode in 1 to M* **do**
2 Initialize sequence with s_1 ;
3 **for** *t in 1 to T* **do**
4 s_c, a_f, s_{t+1}, r_t, done = $HierarchicalStep(A, s_t, n)$;
5 Store transition (s_t, a_t, r_t, s_{t+1}) in D;
6 Set $s_t = s_{t+1}$;
7 Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$;
8 Set $y_j = \begin{cases} r_j & \text{if done} \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$;
9 Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ w.r.t. θ;
10 Every C steps reset $\hat{Q} = Q$, i.e., set $\theta^- = \theta$;
11 **end**
12 **end**
13 **return** Q function;

In this method, the Hierarchical Step only returns the final transition, so only this transition is added to the experience replay in line 5. The idea is that this transition is the most important one since it is the one that actually transitions the environment from the current state s_t to the next state s_{t+1} (or s'), as can be seen in line 12 of Algorithm 5.

The fact that only the final transition is added to the experience replay means that it is the only one that the DQN's NN parameters θ would be

updated by. The inner-transitions are only generated with a feedforward pass of the NN but no backpropagation and parameters update is performed by them. This way we insure that the NN will only learn according to the most important (final) transitions, and the inner-transitions would be generated with the current "wisdom" of the NN.

The second version of the agent is presented in Algorithm 8. In this version we want parameters θ to be updated not only by the final-transition, but also by the inner-transitions, with a reduced importance weight. We assume that updating the NN parameters for the inner-transitions would enhance the convergence properties of the agent due to the mere fact that they significantly enlarge and diversify the observations used for the NN's learning.

We create a weighing method that assigns an importance for each transition (inner or final) and defines the size of the NN's parameters update. We slightly modify Algorithm 5 so it would not only return the final action a_f but the set of all actions of inner-transitions $\{a_c^l\}_{c=1,\dots,C}^{l=1,\dots,L}$, where l is the level of the hierarchy of each action, L is the final hierarchy, c is the index of the transition and C is the total number of transitions in the current hierarchical step. The weight assigned for each action $\{a_c^l\}$ is calculated by:

$$w_c^l = \frac{l}{L * \sum_{c=1}^{C} w_c^l} \tag{6.1}$$

We then add each inner-transition to the experience replay with its weight (lines 6-7). We sample a minibatch of transitions with their weights in line 10 and use these weights to control the size of the update by multiplying them with the loss in line 12.

A possible problem of the second version is getting a *flooded* experience

Algorithm 8: DQN with Hierarchical Step - Weighted Transitions

Initialize replay memory $\mathcal{D}$;
Initialize action-value function Q with random weights θ;
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$;
for *episode in 1 to M* **do**
  Initialize sequence with s_1 ;
  for *t in 1 to T* **do**
    s_t,$\{a_c^l\}$, s_{t+1}, r_t, done = $HierarchicalStep(A, s_t, n)$;
    Create weights set $\{w_t\}$;
    for *i in length*($\{a_c^l\}$) **do**
      Store transition $(s_t, \{a_c^l\}_i, s_{t+1}, r_t, \{w_c^l\}_{c=i})$ in D ;
    end
    Set $s_t = s_{t+1}$;
    Sample random minibatch of transitions $(s_j, a_j, r_j, s_{j+1}, w_j)$ from $\mathcal{D}$;
    Set $y_j = \begin{cases} r_j & \text{if done} \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$;
    Perform a gradient descent step on $w_j * (y_j - Q(s_j, a_j; \theta))^2$ w.r.t. θ;
    Every C steps reset $\hat{Q} = Q$, i.e., set $\theta^- = \theta$;
  end
end
return Q function;

replay, with too many inner-transition which have little influence on the agent's trajectories compared to final-transitions. When sampling a mini-batch of actions, the probability of sampling final-transitions is reduced significantly, so many updates of the NN would be less significant than in the first version. The third version we developed aims to integrate the two previous versions by taking the strong aspects from each one.

In the third version which is presented in Algorithm 9, we keep the weighting method of the second version, but instead of adding each inner-transition to the experience replay with its weight, we add only one transition of a

Algorithm 9: DQN with Hierarchical Step - Weighted Q

Initialize replay memory $\mathcal{D}$;
Initialize action-value function Q with random weights θ;
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$;
for *episode in 1 to M* **do**
  Initialize sequence with s_1 ;
  for *t in 1 to T* **do**
    s_t,$\{a_c^l\}$, s_{t+1}, r_t, done = $HierarchicalStep(A, s_t, n)$;
    Create weights set $\{w_c^l\}$;
    Store transition (s_t,$\{a_c^l\}$, s_{t+1}, r_t, $\{w_c^l\}$) in D ;
    Set $s_t = s_{t+1}$;
    Sample random minibatch of transitions
     $(s_j, \{a_c^l\}_j, r_j, s_{j+1}, \{w_c^l\}_j)$ from $\mathcal{D}$;
    Set $y_j = \begin{cases} r_j & \text{if done} \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$;
    $Q_b = \sum_{c=1}^{C} Q(s_j, \{a_c^l\}_j; \theta) * \{w_c^l\}_j$;
    Perform a gradient descent step on $(y_j - Q_b)^2$ w.r.t. θ;
    Every C steps reset $\hat{Q} = Q$, i.e., set $\theta^- = \theta$;
  end
end
return Q function;

new kind – the *hierarchical-transition*. For each single hierarchical step, this transition is defined by the tuple (s_t,$\{a_c^l\}$, s_{t+1}, r_t, $\{w_c^l\}$). The *hierarchical-transition* enables to save only one transition per hierarchical step in the experience replay, but with the set of all actions selected in the inner-transitions and with all the weights assigned for the inner-transitions. When we sample the minibatch of the hierarchical-transition we can use these actions and their weights to compute a weighted estimate of the Q-value by summing the Q-value estimates of all the actions selected in the inner-

transitions, multiplied by their weights:

$$Q_b = \sum_{c=1}^{C} Q(s_j, \{a_c^l\}_j; \theta) * \{w_c^l\}_j$$

We then compute the loss as usual, but the backpropagation induced by this loss would prompt the update of the output layer's neurons of each inner-transition's action. This method enables a single parameters update for a single hierarchical step but at the same time to consider the inner-transitions in the update.

We

6.3.2 Neural Network Architecture

A more recent improvement to the DQN algorithm is dueling-DQN (D-DQN) [76]. D-DQN achieves faster convergence by decoupling the Q-function to the value function of the state and the advantage function of the actions, thus enabling the DQN agent to learn the value function $V(s)$, separately from the actions:

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a=1}^{|\mathcal{A}|} A(s,a)$$

The D-DQN architecture consists of two separate sub-architectures – one for the value function and one for the advantage of each action over the average – each with its own output layer. Both sub-architectures are fed to a global output layer which computes the combined loss.

Our implementation is a variation of the D-DQN, which makes use of the fact that our state representation consists of multiple components. Our du-

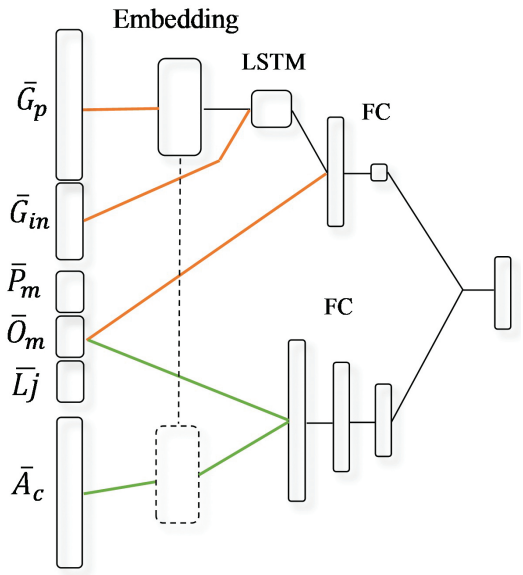

Figure 6.4: The D-DQN agent's NN architecture. The top stream is the *value function component* and the bottom stream is the *actions advantage function component*.

eling architecture is presented in Figure 6.4. We partition the state vector as follows: the vectors that model the state of the grid form the input to the value-function sub-architecture. The vectors that model the task and the possible actions form the input to the action advantage sub-architecture. We define the architecture's objective function as follows:

$$Q(s,a) = V(s^{state}) + A(s^{act},a) - \frac{1}{|\mathcal{A}|}\sum_{a=1}^{|\mathcal{A}|} A(s^{act},a)$$

where the state and action vectors are

$$S^{state} = \{\overline{G}_p, \overline{G}_{in}, \overline{P}_m, \overline{O}_m, \overline{L}_j\}$$

and

$$S^{act} = \{\overline{P}_m, \overline{O}_m, \overline{L}_j, \overline{A}_c\}$$

Due to the unique characteristics of our problem domain, our D-DQN implementation differs from the one proposed in [76] in several important aspects. Unlike the original D-DQN implementation, where the lower layers of both sub-architectures are shared, we separate the two streams completely. The action-advantage sub architecture, which consists only of fully-connected layers, is completely separate from the value-function sub-architecture because of the different characteristics of the inputs for each stream. In the original D-DQN paper, the input for both sub architectures was identical – the pixels of a video game frames. However, in our case we actively distinguish the part of the input that represents the current pipeline in the grid as the *state* from the part of the input that represents the *actions*. Moreover, the types of input are different, with a sequential nature of the *state* input and a spatial nature of the *actions* input.

LSTM and Primitives Embeddings

As depicted in Figure 6.4, we use long short-term memory (LSTM) layers in the value-function sub-architecture [30], due to nature of this sub-architecture's input. The input – concatenation of vectors $\overline{G}_p, \overline{G}_{in}$ – represents our ML-pipeline design as a sequence of *pipeline-steps*, i.e. combination of the primitives and the indices of their inputs. As a preprocessing step, since $\overline{G}_p$, the vector representation of the primitives that populate the current pipeline, is sparse, we add an embedding layer denoted by E.

The value-function sub-architecture is depicted more closely in Figure 6.5. The sequence of *pipeline-steps* is processed in the following way: each entry of vector $\overline{G}_p$ (which represents a primitive) is fed to the shared primitives

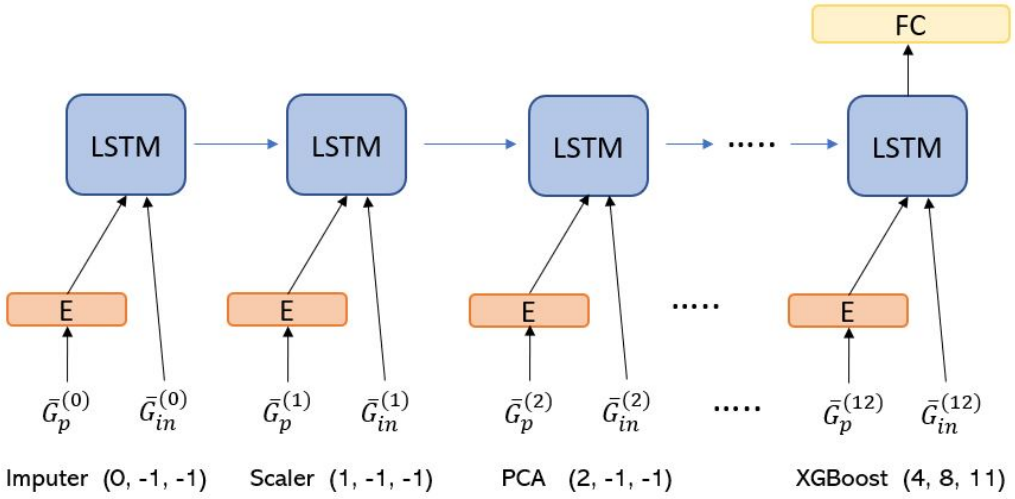

Figure 6.5: A closer look at the *value function component*. We use a language model-inspired architecture to process the sequential input of *pipeline-steps*. An embedding layer outputs a dense representation of each entry in the sparse primitives vector $\overline{G}_p$ which is concatenated with the corresponding entry in the inputs vector $\overline{G}_{in}$ to feed the LSTM cell.

embedding layer (matrix) E. The output of the embedding is a dense primitive representation, which is concatenated with the corresponding entry of the inputs vector $\overline{G}_{in}$. The resulting vector is processed by the LSTM cell. The sequence starts with the first pipeline-step of the pipeline and ends with the final pipline-step. The final output of the LSTM architecture is fed to a Fully-Connected layer.

We note that the described architecture is inspired by models that successfully use LSTMs for language modeling and text generation [66]. We treat the *pipeline steps* in our model in the same way that words of a sentence are processed by an embedding layer and LSTM cells in language models.

Embedding Matrix in other Parts of DeepLine's Framework

As previously mentioned and depicted in Figure 6.1, matrix E is also utilized to represent the primitives in the actions vectors $\overline{A}_c$. The *hierarchical step* relies on this embedding to perform the clustering of the actions in the *open list*. The action vectors are also used as inputs to the actions advantage sub-architecture of the Q-function NN, as seen in Figure 6.4. Applying the same embedding for the hierarchical step means that in the early stages of the training, the actions representations are random but as time progresses the representation becomes meaningful and the clusters more concise.

6.4 Pipeline Exploration

After training the agent offline, we use the trained model online for unseen before datasets. The exploration of a pipeline for a given learning job can be conducted with two different methods:
(1) Returning the best pipeline. For a given number of pipelines to explore, k, defined by the user, the agent generates and evaluates the pipelines on the train set, with k-fold evaluation. Denoting the k-fold score as $KScore$, we calculate the following score for each pipeline:

$$Score = \beta Q(S^{final}, a^{final}) + (1 - \beta)KScore \tag{6.2}$$

Where S^{final}, a^{final} are the final state and action of the episode (i.e., the final product of the pipeline) and β is a tunable parameter. We then return the pipeline with the highest score to be evaluated on the test set by the

user. The use of the q-function ability to predict the performance of the pipeline is a powerful tool for comparing pipelines, giving our framework an additional advantage.

(2) Ensemble of pipelines. We gather the probability predictions of the k explored pipelines and perform a weighed average according to the pipelines' obtained scores, computed as detailed in method (1).

7 Evaluation

In our evaluation we want to examine two main things. First, the performance of our framework in the supervised classification problem in regard to its ability to produce ML pipelines compared to other methods. Here we also consider the number of pipelines evaluated during the exploration process. Secondly, we want to evaluate the effect of the hierarchical-step plugin on the convergence properties of the agent.

7.1 Experimental Setup

In this section we present the database which we designed and implemented and the data that we gathered to populate the database. We also detail the complete settings configuration that was used throughout the experimental process and describe the different experiments and their purpose.

7.1.1 Database Setup

DeepLine's framework requires integration with a platform that would enable the generation of pipelines, step by step, based on the graph rep-

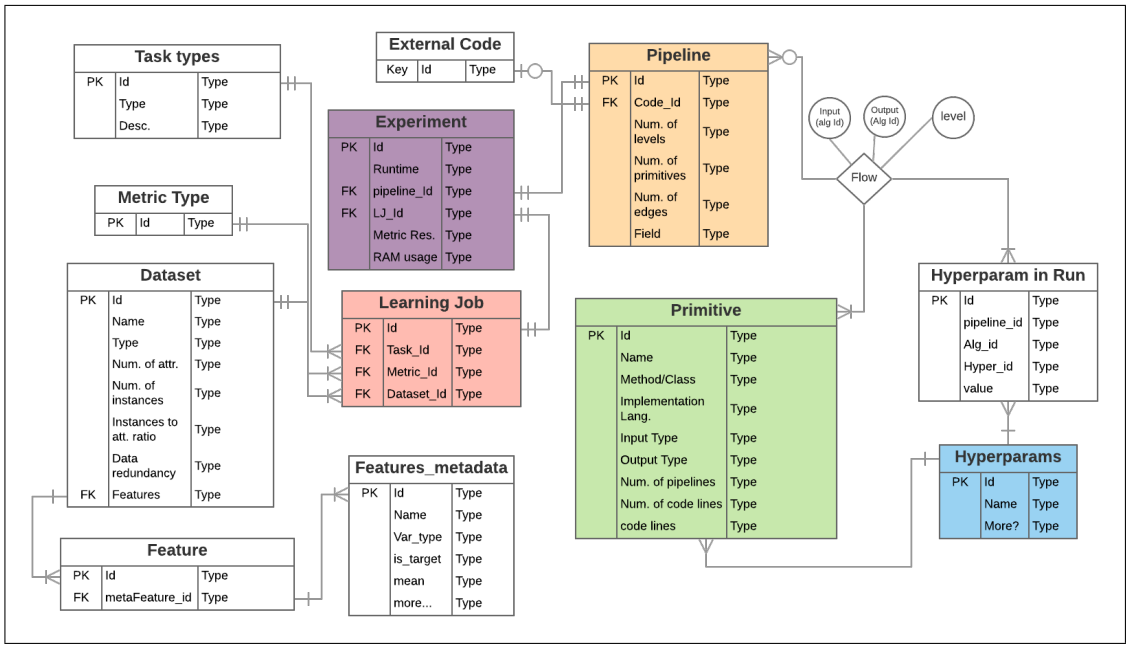

Figure 7.1: The ERD of the database used for our experiments

resentation that was defined in the Problem Formulation. The platform should also accommodate the evaluation of pipelines over specific learning jobs. Due to the complexity of the desired platform and the need to save the different components (i.e., primitives, pipelines and datasets) and the pipelines' products, we designed the platform as a database with a schema which is depicted in Figure 7.1.

The database purpose is to allow for any operation required during the learning or online phases of our model, including the following: insert primitives to an existing pipeline; extract a dataset and its meta-features; fit pipelines over a dataset; produce predictions over a dataset; allow for easy evaluation of the generated pipelines; storing the pipelines that were generated during the offline phase; and storing the online experiments results. The database schema was built as an ER diagram but was implemented in a No-SQL environment to enable easy modifications and flexi-

bility during our research. It contains the following main entities:

- **Primitive** - entity for storing the entire collection of primitives. As detailed in chapters 2 and 5, the primitives are the building blocks of the pipeline and can be any algorithm used as a part of an ML model. Each primitive has a collection of hyperparameters associated with it with a range of possible values. The primitive can have either a single input or multiple inputs, and one output. An input to a primitive is either the raw tabular dataset or a *dataframe* – a transformation of the original dataset after being processed by other primitive(s). In case of multiple inputs, the dataframes are merged horizontally and duplicate columns (features that repeat over more than one input) are removed. A primitive has code lines for executing the algorithm with two main methods: *fit* for fitting the primitive over the merged input and *produce* for producing the output of that primitive over the input. Note that after fitting the primitive for a certain dataframe, it can be used for other dataframes for producing the results (e.g. fitting over a train set and producing over a test set). Each primitive also has a method called *CanAccept* which returns a Boolean noting whether the primitive can be applied for a given input. This method is important for the creation of the actions **open list** of DeepLine's environment.

- **Pipeline** - entity for storing and executing the pipelines that are generated by our model and for storing pipelines that were either manually or automatically generated beforehand by experts and/or other autoML systems. As depicted in Figure 7.1, the pipeline is defined as a *flow* of pipeline-steps, where each step has a single primitive that

receives input(s) and produces an output which can be transferred to other primitives. This definition enables the creation of pipelines in the form of a DAG. This pipeline entity too has methods *fit* and *produce* for fitting the pipeline by fitting each of its primitives and for producing predictions over a dataset. The pipeline entity also holds metadata information of the pipeline (graph-based features).

- **Hyperparams** - Each primitive is associated with a collection of hyperparameters entities. The values of the hyperparameters are determined only when associated with a pipeline, by the **Hyperparams in Run** entity.

- **Learning Job** - As mentioned before, the learning job is defined as the combination of a dataset, a task and a metric and can be treated as the input of a pipeline.

- **Dataset** - The dataset is the most prominent piece of the learning job and is associated with the metadata of its features.

- **Experiment** - entity for storing the products of a pipeline for a given learning job. The results are the scores of the pipeline on the defined learning job's metric/s, the run-time and the RAM usage. The collection of experiments can be used for pre-training our model's network on predefined pipelines.

We now detail the data that we acquired for specific entities of the database, namely the datasets and primitives which form the main building blocks of the database.

Datasets

We populate our database with 56 tabular datasets that are all available in the following online repositories: UCI[1], Kaggle[2] and OpenML[3]. All datasets are specified for the classification task with either binary or multi-class targets. We use these datasets for the extensive evaluation of DeepLine, as will be detailed in the following sections.

We perform analysis of the 56 datasets, that is presented in Figure 7.2. As we can see, the datasets are very much diverse, with large variety in size, number of attributes, feature type composition (i.e., categorical and numeric features), percentage of missing data and number of classes. We assume that using a diverse set of datasets for DeepLine's evaluation may strengthen the validity of the results and better indicate real-world behavior of our model. The complete list of datasets is given in Table A.1, Appendix A.

Primitives

We populate our database with a set of 36 primitives. Almost all primitives are implementation of ML algorithms based on modules from the Scikit-Learn Python library [52]. We use the same set of primitives for all experiments, both for DeepLine and for the two AutoML baselines. We list all primitives by their associated family in Table 7.1.

[1]https://archive.ics.uci.edu/ml/datasets.php
[2]https://www.kaggle.com/datasets
[3]https://www.openml.org/search?type=data

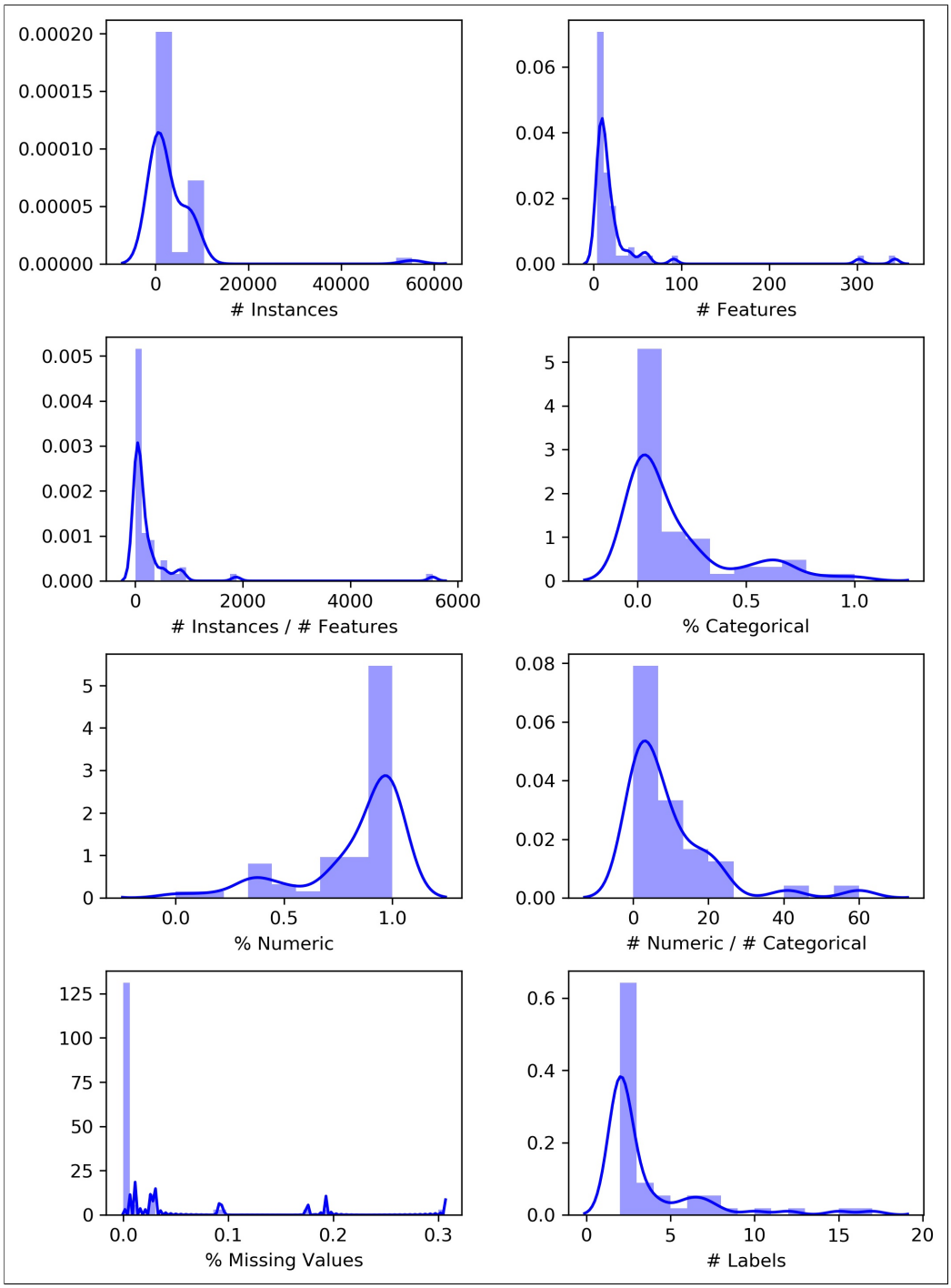

Figure 7.2: Datasets Analysis

Table 7.1: The list of Primitives used by DeepLine and the baselines for the evaluation

Data Preprocessing	**Feature Preprocessing**	**Feature Engineering**	**Feature Selection**	**Classifiers**
One Hot Encoder Imputer - Mean Imputer - Median Imputer-Encoder Numeric Extractor	Normalizer Quantile Transformer Robust Scaler Max Absolute Scaler Min Max Scaler Standard Scaler	Polynomial Features Interaction Features Kernel PCA Randomized PCA Fast ICA RandomTrees Embedding	Variance Threshold Univariate Select: (Chi Kbest) (Chi Percentile) (Chi FWE) f_classif: (percentile) (FWE) Mutual Info: (Kbest) RFE-RandomForest	Random Forest XGBoost Logistic Regression Bernoulli NB Decision Tree Extra Trees Gaussian NB Gradient Boosting KNeighbors Linear SVC Multinomial NB

7.1.2 Pipeline Generation Experiment

Baselines

We evaluate five different baselines to compare with DeepLine's performance in the task of classification. We evaluate two types of baselines, in compliance with other AutoML works in the past:

- **Basic popular pipelines**: used to evaluate whether our approach is better than popular pipelines that are often used by non-experts or as a first-line solution for many tasks. We use three basic pipelines as baselines, each consisting of two pre-processing primitives – missing values imputation and one-hot encoding for categorical features – and one of the following classifiers: **Random Forest** [39], **XGBoost** [13] and **Extra-Trees** [23]. All three models have shown great success over a wide range of domains.
- **Pipeline generation frameworks**: we chose two of the most popular open source AutoML pipeline generation platforms: **TPOT** and **Auto-Sklearn**. Both achieve current state-of-the-art results [55]. We detail extensively about each of these frameworks in Chapter 2.

To ensure fair comparison, DeepLine uses the same set of primitives used both in TPOT and Auto-Sklearn. These primitives are listed in Table 7.1.

Pipeline Generation Settings

For both TPOT and Auto-Sklearn, we used the default parameter settings, as detailed in [49] and [21]. In the case of TPOT, this results in the generation and evaluation of approximately 10,000 pipelines for each dataset.

We ran Auto-Sklearn for 30 minutes on each dataset, resulting in approximately 700 pipelines on average per dataset.

By default, both TPOT and Auto-Sklearn return at the end of the search process a single "optimal" pipeline. This pipeline is chosen based on its average performance on the folds of the training set. However, both DeepLine and Auto-SKlearn can also perform ensemble of pipelines. We therefor evaluate two versions of DeepLine- the first one, denoted by v (for vanilla) in Table 7.2 returns the top-ranked pipeline out of k generated pipelines. The second one, denoted by e (for ensemble) creates a weighted average of the k pipelines predictions relative to their scores. The same notations are used for the vanilla and ensemble versions of Auto-SKlearn.

Experimental Process

We perform the following steps to evaluate DeepLine's in the classification task over the 56 datasets:

- We split the 56 datasets that are listed in Table A.1 to four folds, each containing 14 datasets.
- For each fold, do:
  - Train the DQN agent on the three remaining folds for a few thousand episodes
  - Obtain the best model of the training phase using the average rewards as the metric to evaluate the model in each iteration.
  - Test the best model (of the trained agent) over the current fold in the following way:
  - For each dataset in the current fold:

    - Split the dataset to train-set in test-set in a ratio of 0.8-0.2.
    - Perform pipeline exploration as detailed in the previous chapter - the agent takes the train-set and generates k pipelines for it.
    - Obtain the best pipeline or the ensemble of generated pipelines using Equation 6.2.
    - Do the same with the two pipeline generation baselines and obtain their best pipelines.
    - Evaluate the obtained pipelines over the test-set using the metric of choice and register the scores as the results of the models (ours and the baselines) for the current dataset
- Perform extensive analysis of all the registered results.

We use the *Accuracy* metric to evaluate the classification performance and we also consider the number of generated and evaluated pipelines (k) to compare DeepLine's computational cost with that of the baselines.

Hyperparameters Configuration

We used the following settings of DeepLine throughout the evaluation:

- We set the parameters N and n to 3 and 6 respectively, meaning that we use a 3X6 grid (see Figure 6.2) and action-clusters of size 6.
- Parameter β is set to 0.3.

The DQN implementation is based on the Stable-Baselines library, but with many additional changes made by us [29]. The DQN hyperparameters configuration is as follows:

- Discount factor $\gamma = 0.99$
- Initial ϵ of the $\epsilon - greedy$ algorithm is set to 1.0
- Final ϵ of the $\epsilon - greedy$ algorithm is set to 0.02
- $\epsilon - greedy$ decay rate is set to 0.99
- Batch size is 32 transitions
- Q-targets update frequency (parameter c in Algorithm 7) is 500 steps.
- PER memory size is 50,000 transitions
- PER-α is 0.6
- PER-β is 0.4

We used the Tensorflow library to construct the agent's NN. The network's architecture is constructed as follows:

- the value-function sub-architecture consists of embedding vectors of size 15 and an LSTM of size 80, followed by three fully connected (FC) layers with lengths of 256, 128 and 32.
- The action-advantage sub-architecture consists of four FC layers with lengths of 256, 128, 64 and 32.
- The NN's learning rate is set to $\alpha = 0.0005$.

The agent was trained offline for 30 hours on average on 1 NVIDIA Tesla P100 GPU.

7.1.3 DQN with Hierarchical Step Experiment

The purpose of the second experiment we conduct is to analyse and determine the contribution of the Hierarchical Step Plugin to the performance of DeepLine, and more specifically the convergence speed of the agent that integrates with the plugin.

We analyse four different DQN agents. The first three agents are the three *DQN with hierarchical step* variations that we detailed in section 6.3.1: (1) Version 1.0 - the basic and most simple variation that use the hierarchical step instead of the simple environment step with no additional changes to the DQN algorithm (Algorithm 7). We remind that in this version we only use the final transition of the hierarchical step and add only this transition to the experience replay; (2) Version 2.0 - DQN with weighted transitions - using the inner-transitions of the hierarchical step in addition to the final transition (Algorithm 8). We add these transitions to the experience replay and assign weight to each transition relative to its hierarchy level. We denote this variation as *weighed obs*, and; (3) Version 3.0 - DQN with weighed Q-values. In this variation we assign weights to the inner transitions relative to their hierarchy levels and use the weights to compute a weighted average of the estimated Q-value of the step (Algorithm 9). We denote this variation as *weighed Q*.

The fourth agent is a DQN without the hierarchical plugin at all (Version 4.0). Not using the plugin required us to create a fixed-length actions space, resulting in a set of 7,800 unique actions, each action being a primitive-inputs combination. We also included one action that leaves the cell empty - the *blank* action. All other settings, including the negative reward for invalid actions, remained the same.

Table 7.2: Pipeline generation results. The result column is the average *accuracy* score over 56 datasets. We also present the p-value obtained in a paired t-test between the accuracy results of our model (vanilla (v) or ensemble (e)) over the specified method. k - number of evaluated pipelines online.

Method	**Result**	**P-Value^v**	**P-Value^e**
Random Forest	0.785	0.041	0.002
XGBoost	0.791	0.151	0.011
Extra Trees	0.778	0.006	0.0001
TPOT, $k \cong 10,000$	0.793	0.122	0.004
Auto-Sklearn^v, $k \cong 700$	0.784	0.125	-
Auto-Sklearn^e , $k \cong 700$	0.794	-	0.062
DeepLine^v, $k = 25$	0.799	-	-
DeepLine^e, $k = 25$	**0.811**	-	-

7.2 Evaluation Results and Analysis

7.2.1 Pipeline Generation Results

The results of the evaluation are presented in Table 7.2 and Figure 7.3. Figure 7.3 depicts the average accuracy results of both versions of DeepLine (vanilla and ensemble) by the number of generated (i.e., evaluated) pipelines. It is clear that DeepLine outperforms all the baselines: vanilla DeepLine needs to evaluate only 10 pipelines to outperform the three basic pipelines - XGBoost, Extra Trees and Random Forest. Moreover, only 15 pipelines need to be evaluated for DeepLine to outperform the two state-of-the-art methods, TPOT and vanilla Auto-Sklearn. The same number of pipelines (15) is also sufficient to match the performance of ensemble Auto-Sklearn.

The exact results of all models are presented in Table 7.2. We also note in Table 7.2 the parameter k, which is the number of pipelines per dataset that were evaluated in each of the AutoML frameworks to achieve its best results. When used in its ensemble version, DeepLine outperforms

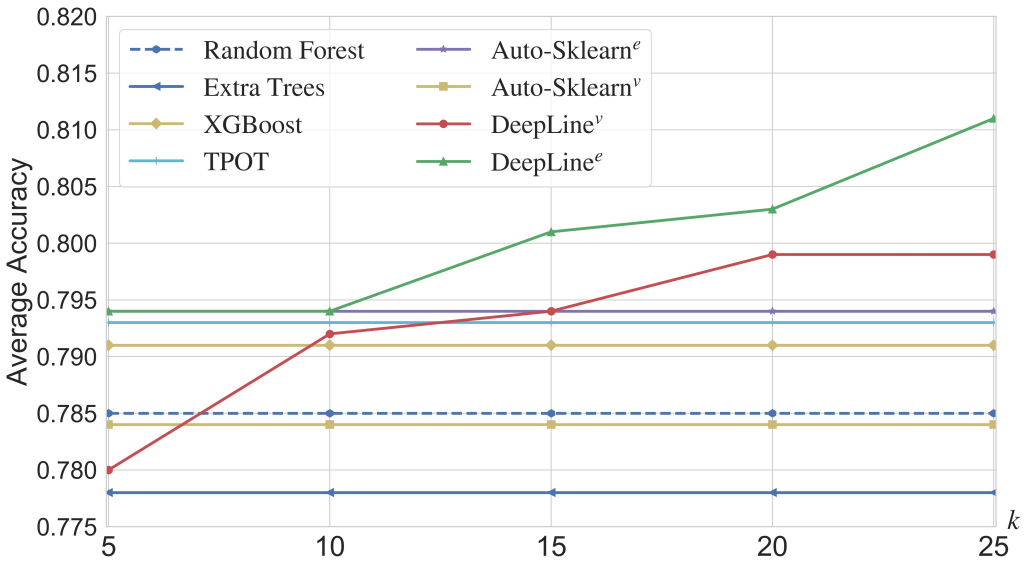

Figure 7.3: Average accuracy over 56 datasets with increasing number of generated pipelines (k)

all other methods, even beating the top-performing baseline (ensemble Auto-Sklearn) by 2.1%. These results are particularly significant given that DeepLine evaluates 25 pipelines at most for each dataset, 0.25% and 3.5% of the number of pipelines evaluated by TPOT and Auto-Sklearn during their optimization, respectively.

We analyze the per-dataset performance of DeepLine compared to those of the various baselines. Figure 7.4 plots the relative performance of our framework to each of the baselines. It is clear that our approach achieves the top performance in a large majority of cases, with the percentage of improvement ranging from 58.9% to 75% of all datasets. Moreover, when DeepLine achieves better result on a dataset it usually does so with a larger margin than the other way around. The positive margins are particularly large when we compare DeepLine to the three basic pipelines. We also see that the improvements of DeepLine over the baselines are not limited

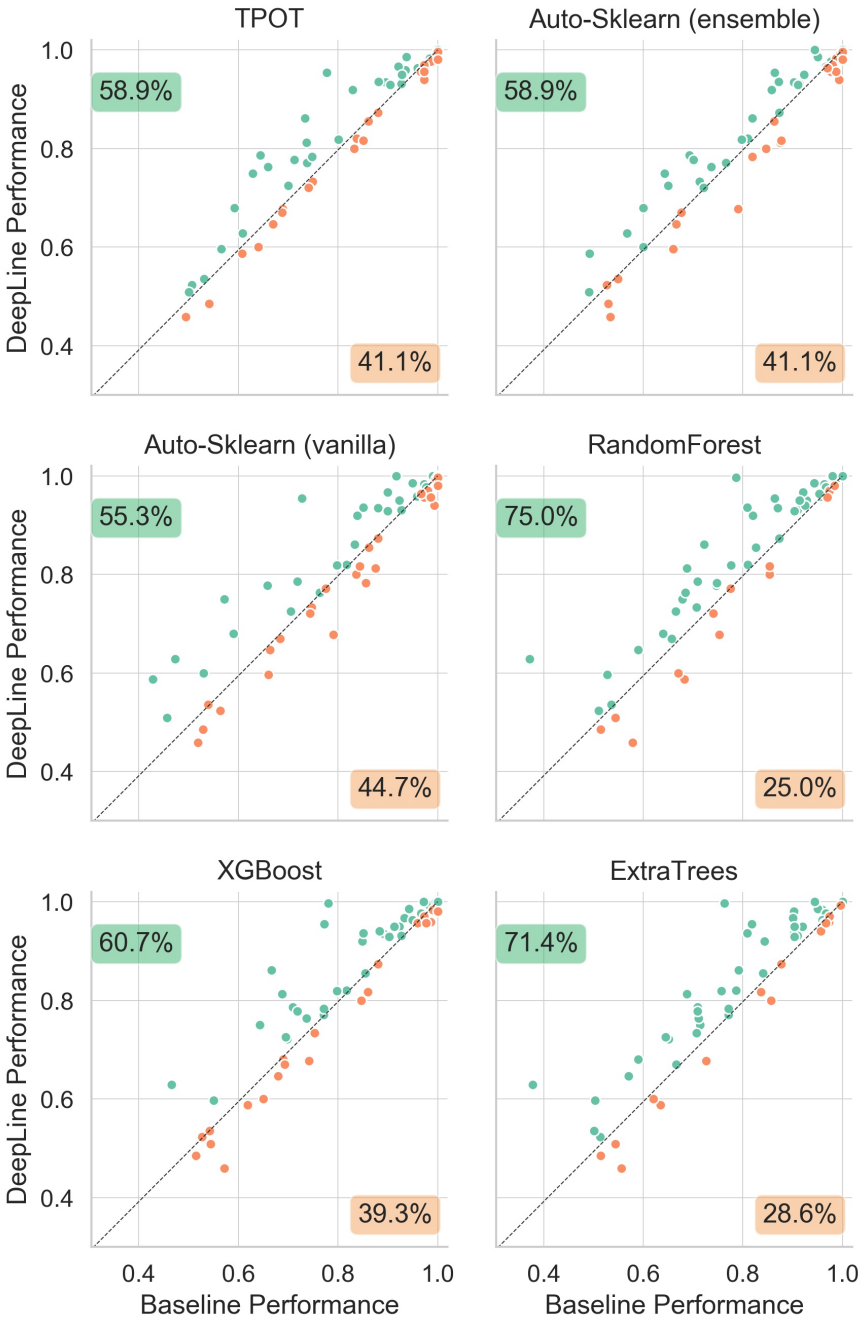

Figure 7.4: DeepLine Vs. baselines performance in 56 datasets

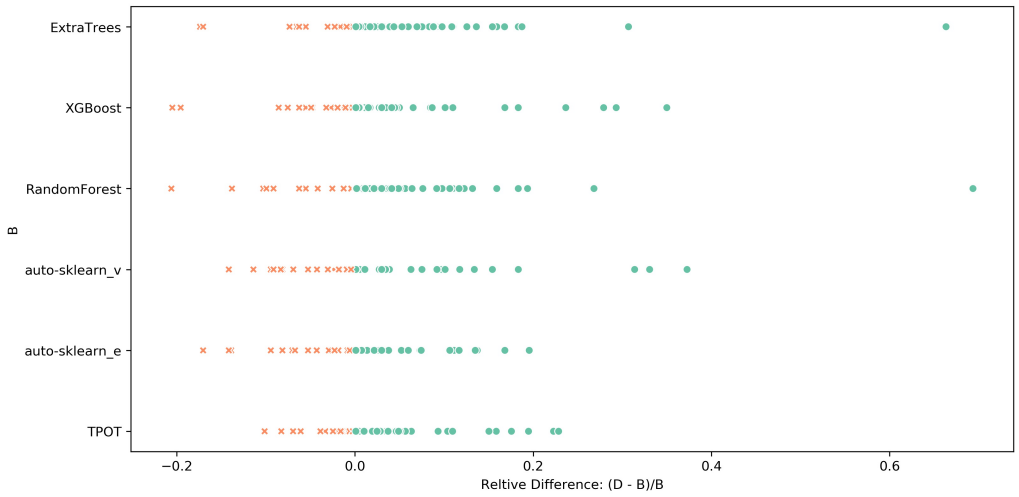

Figure 7.5: Relative difference in the scores of DeepLine (D) compared to each of the baselines (B) per dataset, for 56 datasets.

to any specific scope of datasets or results, for example difficult datasets with lower accuracy results in general or easy datasets with high accuracy results, but are spread throughout the scores range.

The same tendency is observed when looking at the relative difference between the scores of DeepLine and each of the baselines (per dataset). In Figure 7.5 we present the relative accuracy scores on the 56 datasets, calculated by: $D - B/B$, where D represent the score achieved by DeepLine and B the score obtained by a baseline for the same dataset. We see that the relative difference leans firmly towards DeepLine in a large portion of the datasets and that the range of the negative difference is smaller significantly than the positive one.

Next, we analyze DeepLine's relative performance on datasets with varying #records/#features ratio. We hypothesized that lower ratios (i.e., smaller datasets with a large number of features) would be more difficult for the baselines to perform well on, because of the dangers of overfitting. Fur-

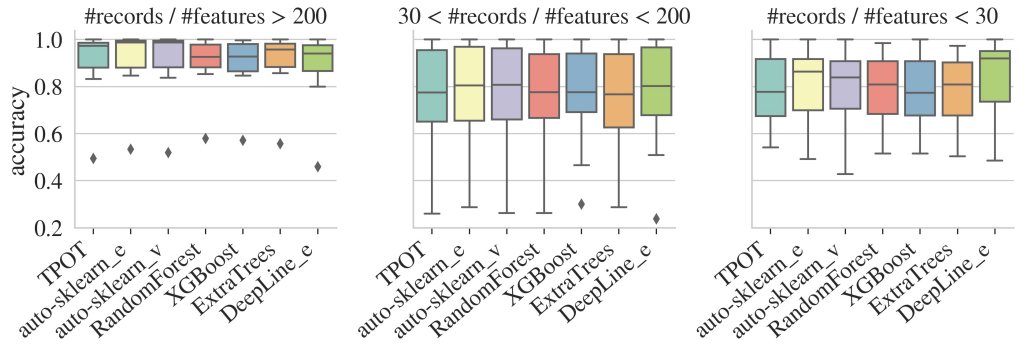

Figure 7.6: Box plots depicting the distribution of accuracy results on the test sets across multiple datasets, comparing DeepLine with its baselines. The three plots show the results over datasets with a decreasing records-to-features ratio range.

thermore, we hypothesized that DeepLine would be less affected by this setting, as the majority of its learning is performed *offline*, on other datasets. The results in Figure 7.6 confirm our hypotheses, as they show that DeepLine's lead increases as the said ratio declines.

Finally, we use the paired t-test to verify the statistical significance of our results. The results, presented in Table 7.2, clearly indicate that the ensemble version of DeepLine outperforms all baselines with a $p - value < 0.01$. The only exception is the Auto-Sklearn ensemble, for which the p-value is slightly higher but still significant, with $p = 0.062$.

Analysing the Generated Pipelines

We now look at the pipelines that were generated by DeepLine during the exploration process and analyse them.

First, we consider the architecture of the generated pipelines by plotting histograms of the number of nodes and number of edges of the pipelines. In figure 7.7 we can see that DeepLine generates pipelines of all sizes, rang-

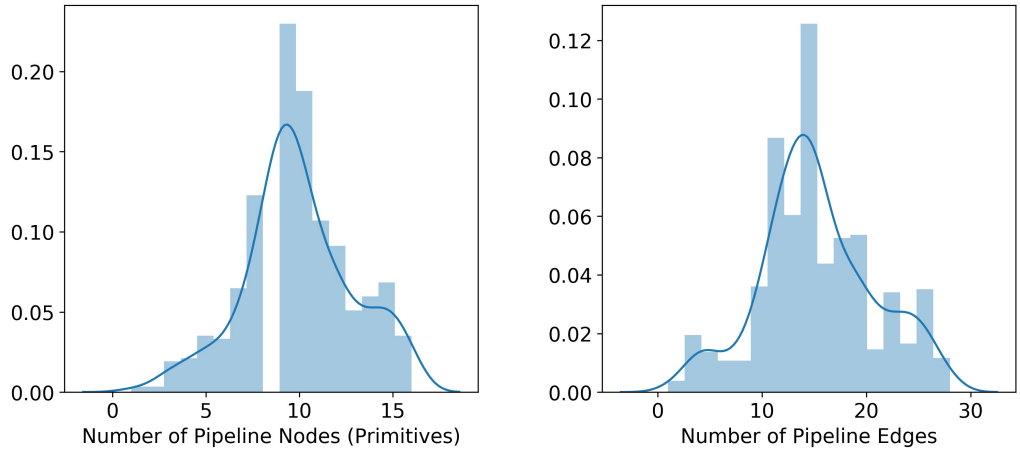

Figure 7.7: Generated Pipelines: number of nodes (left) and number of edges (right) histograms.

ing from pipelines with only two or three primitives to pipelines with the maximal number of primitives allowed by our environment. We can also learn from the plots that the number of edges in the pipeline usually exceeds the number of nodes, meaning that more complex, inter-connected pipelines are generated. Both distributions resemble normal distribution, with a mean of approximately 9 primitives and 15 edges in a pipeline.

The fact that the pipelines vary in size and intricacy indicates that our initial intention of creating a RL environment that will allow for a relatively large range of pipeline's complexity levels in a semi-constrained environment has succeeded. Furthermore, it is clear that DeepLine's DQN agent takes full advantage of the flexibility that it is given by the environment for generating different types of pipeline architectures.

The fact that many of the pipelines are large in size means that the agent learned to postpone immediate rewards and decided not to jump to the last cell of the grid when it had the option in many cases (remember that in the the right-most top cells the agent can choose to skip to the last cell

and receive the score of the current pipeline as the reward).

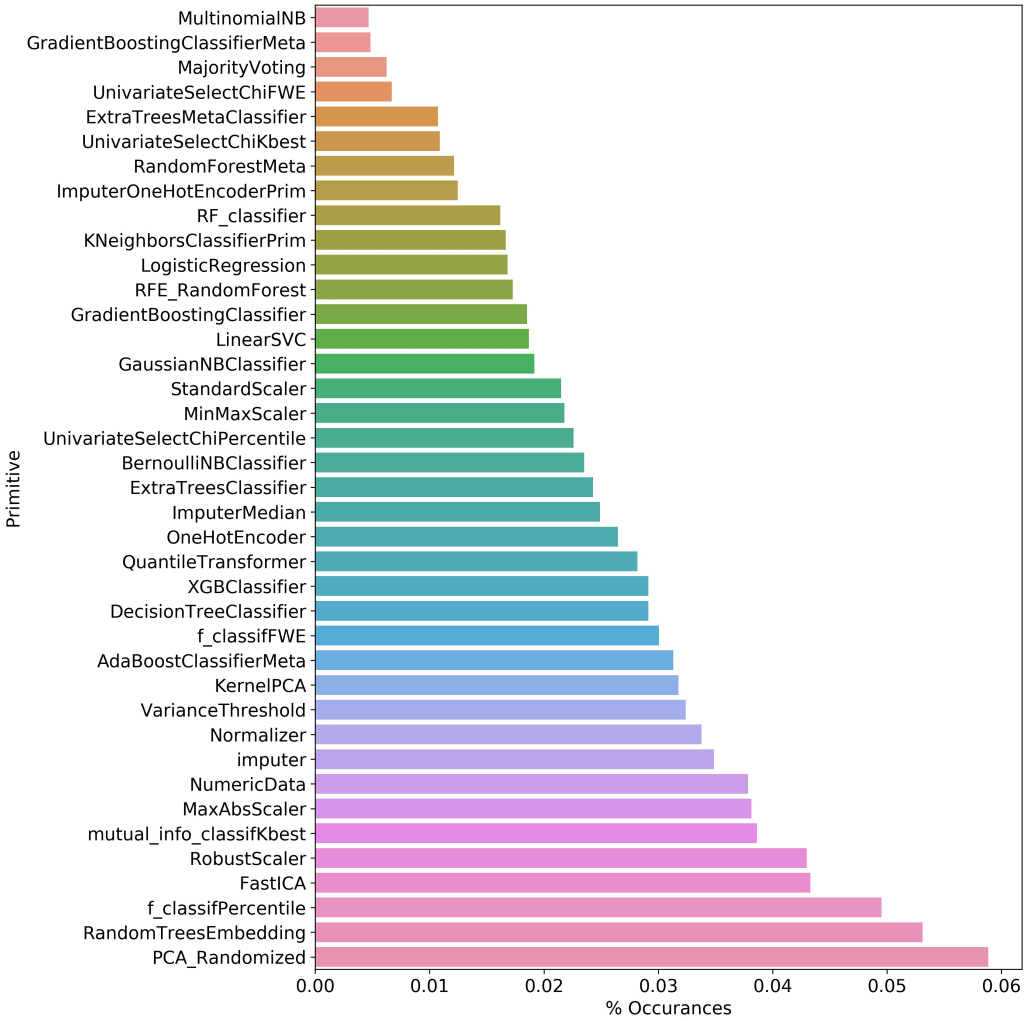

Figure 7.8: Percentage of each primitive occurrences in all generated pipelines.

When looking at the percentage of occurrences of each primitive in the generated pipelines (Figure 7.8), we see that the agent used each and every primitive available at some point of the pipeline exploration process. This means that it does not ”lock” to any specific primitive or primitives and that there is a chance it would pick any primitive at some point.

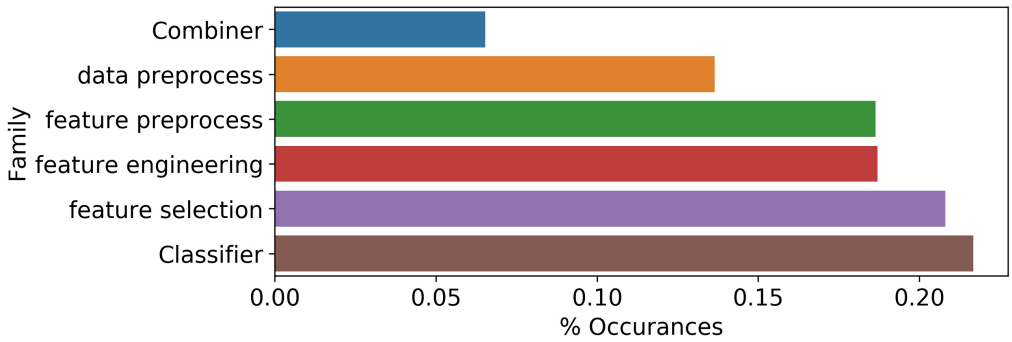

Figure 7.9: Percentage of each primitives' family occurrences in all generated pipelines.

We also see that some primitives are much more frequent than others and that the primitives at the top of the list are primitives that transform the data dimensionality and primitives that perform feature selection. However, when we look at the percentage of occurrences by primitive' families (Figure 7.9) we see that the most frequently used family is the Classifiers family, but other families are also not far behind. The only exception is the Combiner family, but the reason for its lower frequency is the fact that only one Combiner primitive is allowed in each pipeline.

When we focus only on the primitives that belong to the Classifiers family in Figure 7.10, we see that the most frequently used primitives are XGBoost Classifier and surprisingly, Decision Tree Classifier. The fact that a weak classifier such as the Decision Tree was chosen many times by the agent may be explained by the architecture of the generated pipelines – as ensemble models. The generated pipelines often include multiple classifiers that are combined by the Combiner primitive, so each classifier is not a single player. Even more complex ensemble models such as Random Forest use the decision tree as the weak classifier of choice.

We also see in Figure 7.10 that almost all classifiers are used quite fre-

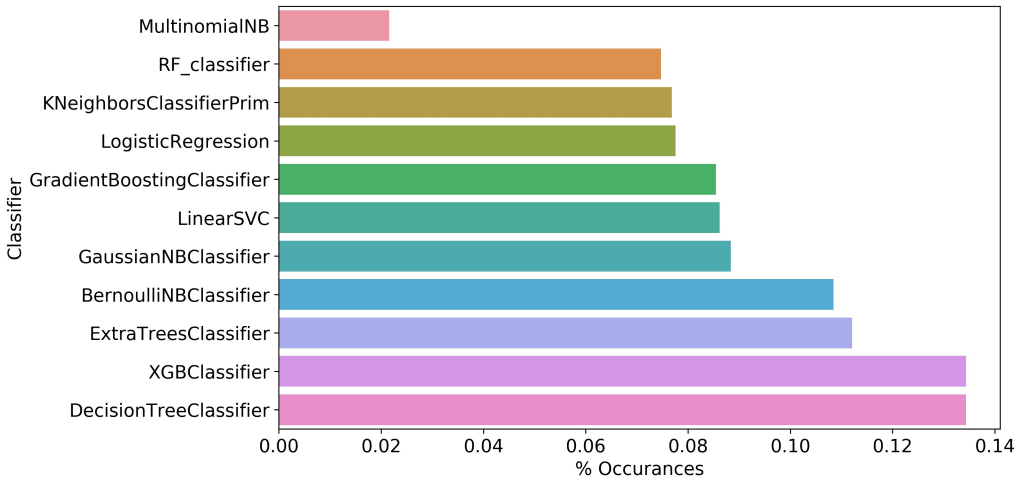

Figure 7.10: Percentage of each classifier-primitive occurrences in all generated pipelines.

quently. This is another example of an ensemble technique, known as the Stacking method, that might have been learned and adopted by DeepLine's agent[56]. In the Stacking method, several classifiers are trained over the dataset and their predictions are stacked and used as input to a meta-classifier, similar to the Combiner primitive. To achieve best results, it is recommended to use classifiers of different types (e.g. Trees-based, Naive Bayes algorithms, SVM, etc.) since each type classifies the samples a bit differently, where each classifier might perform better than others in some cases. The fact that all classifiers are almost equally frequent in the plot of Figure 7.10 might indicate the the agent selects multiple types of classifiers for each pipeline.

7.2.2 Hierarchical Step Results

We first compare our basic form of DQN with hierarchical step (Version 1.0) with a DQN agent with no hierarchical plugin at all and a standard

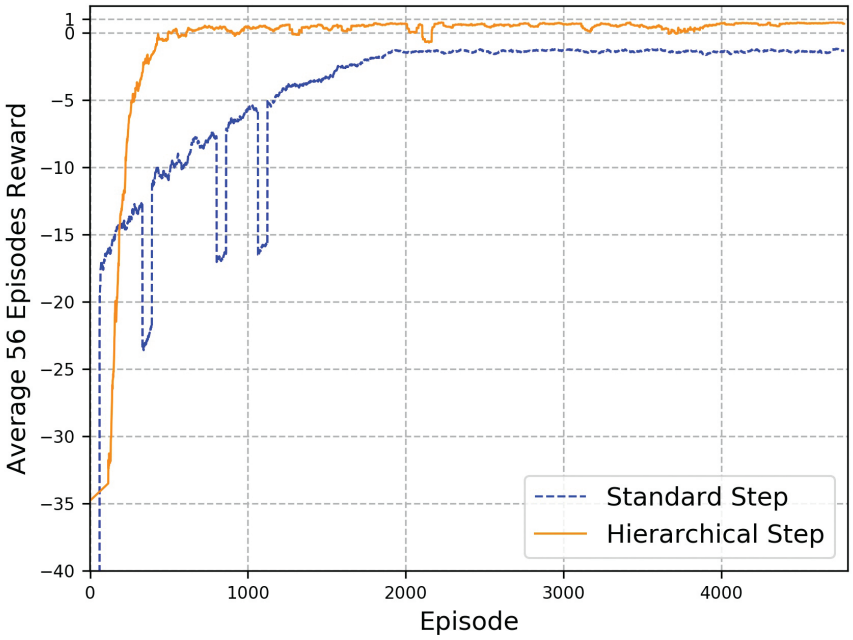

Figure 7.11: Our agent's convergence with a large fixed-size set of unique actions and a standard environment step Vs. the agent with the use of the *open list* and the hierarchical step.

environment step (Version 4.0). Figure 7.11 plots the average rewards obtained by the two agents over all 56 datasets for the first 5,000 training episodes. It is clear that the hierarchical plugin not only enables faster convergence, but also produces better rewards in the long term.

Additional analysis of the agent's behavior during training with and without the plugin shows significant differences in action selection. The hierarchical step agent quickly learns which actions incur a negative reward and avoids them. Furthermore, the fast convergence of the agent (only about 500 episodes needed to achieve only positive rewards) indicates good exploration properties of the agent. The hierarchical plugin not only enables the agent to explore various primitive-inputs combinations, but also forces it to explore and evaluate all the actions space in each step by iterating over

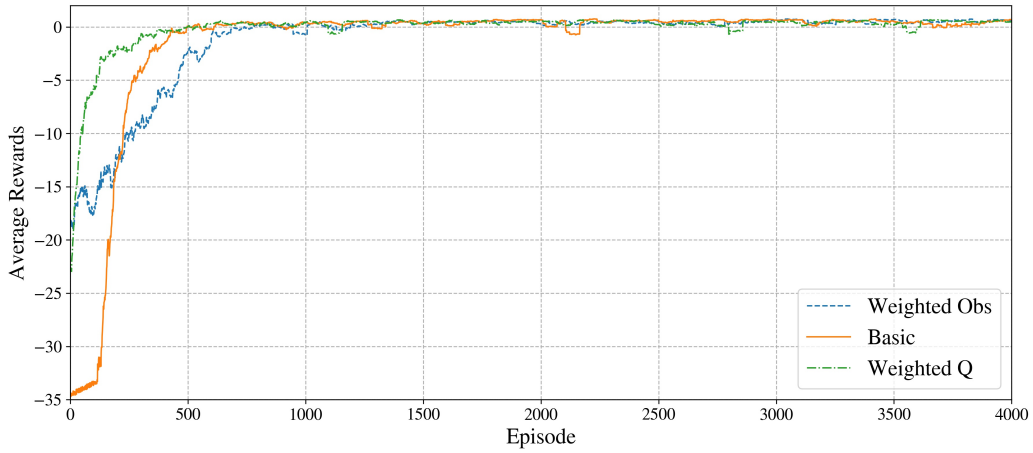

Figure 7.12: Comparing the convergence properties of the three DQN with Hierarchical Step variations. The graph shows the running average rewards with a window of 56 episodes, of each of the agents.

all the action clusters of the *open list*.

Without the plugin, the large and complex actions space made it extremely difficult for the agent to carry out effective exploration and to converge in a reasonable time. In most states, a large number of the available 7,800 actions were *invalid*, reaching even a portion of 95% of the actions. As a result, the agent learned to select almost exclusively the *blank* action which is always valid. However, this policy is of course idle, since it results in a grid with only blank cells, meaning that no pipelines are generated at all. As seen in Figure 7.11, whenever the agent tries to venture new actions it experiences large drops in the rewards so it immediately returns to the *blank* action policy. To conclude, the hierarchical plugin forces the DQN agent to explore multiple actions and obtain useful feedback while the non-hierarchical representation delays useful exploration.

We also present a comparison between versions 1.0, 2.0 and 3.0 regarding the convergence properties of the different agents, in Figure 7.12. We

see that all three agents reach convergence and present similar performance in the long term, with relatively stable average rewards over time. However, before reaching convergence, each agent performs slightly differently. We see that the worst performing agent is the *weighed obs* agent (version 2.0), that achieves only linear progress in the accumulated rewards before reaching convergence. The two other agents (Versions 1.0 and 3.0) achieve exponential growth in the average rewards. The fact that the *weighed obs* agent performs worse than the other two can be explained by the fact that the experience replay gets "flooded" with inner-transitions which has less explanatory power. These transitions have much higher probability to be selected to the mini-batch in each iteration than the final-transitions. Furthermore, due to the lesser weights that the inner-transitions are assigned, each mini-batch update becomes much less significant.

As seen in Figure 7.12, the best way to integrate the hierarchical step with a DQN agent is by version 3.0 - the *weighted Q* agent. In this version, the weighted update of all the neurons that participated in the hierarchical step significantly advances the learning in its early stages. Moreover, this method enables a trade-off between the two other methods, as it considers the final-transition in each of the mini-batch sampled observations while still considering the inner-transitions that are also important for the learning. The only limitation of this method is the need to modify the DQN agent, while the basic form of version 1.0 does not require any changes and still achieves exponential, but a bit slower, growth of the rewards. The choice between the two versions is not straight-forward, and should be decided by weighing the implementation implications and the convergence speed of each version. In cases of "light" environments, one would probably pick version 1.0 since it its easy to implement and the time cost

of applying it is not significant. When dealing with more computationally expensive problems, version 3.0 might be preferred.

8 Summary and Conclusions

8.1 Summary

In this Thesis we researched the application of advanced deep reinforcement learning (DRL) methods for the task of automatic machine learning pipeline generation. By formulating the problem as a Markov Decision Process (MDP) we were able to treat the task at hand as a sequential decision making problem, and to design an end-to-end DRL-based framework we call DeepLine to solve it.

We first defined the ML pipeline architecture as a DAG, with ML primitives as its vertices. Using this representation we presented a novel approach to model the pipeline generation process in a semi-constrained setting of a grid-world environment. This environment significantly limits the huge search-space of primitives and pipeline architectures while accommodating the creation of relatively complex and inter-connected pipelines, including pipelines with several preprocessing primitives and multiple classifiers that are all stacked together to produce a single output.

While efficient, the aforementioned environment came with the challenge of large and *dynamic* actions space in which the number of *valid* actions changes in each state and the number of invalid actions could constitute

up to 95% of the actions, making the agent's learning process impractical. This problem is not unique to our environment and can be found in other domains as well. To tackle this challenge we proposed the Hierarchical Step Plugin, a novel and efficient hierarchical approach that enables the agent to iteratively query small, fixed-size subsets of the dynamic actions space. The plugin is implemented as a component that can be easily integrated to mediate the interaction between an agent and an environment, and can be used with any DRL algorithm. As part of the plugin we also implemented a clustering method that creates constant-length clusters of actions, where each cluster represents a different subset of the actions space. The use of the plugin not only solves the problem of the changing actions space but also improves the agent's exploration properties and increases the agent's convergence speed.

We also presented a DQN agent with a unique dueling architecture that utilized the fact that our state representation consists of multiple components and distinguished the inputs that relate to the actions from inputs that represent the state. We used each input in a different stream of a bi-streamed neural network with a single Q-value output. We also processed an efficient, sequential representation of the pipeline with a LSTM architecture in a way that resembles sentences processing in language models.

Finally, we proposed three different variations for integrating the Hierarchical Step with the DQN algorithm: (1) a basic integration method that simply places the hierarchical plugin instead of a standard step; (2) A method that assigns weights to the transitions of the lower hierarchies of the plugin and adds them to the experience replay, and; (3) computing a weighted average of the Q-values of all the transitions in all hierarchies. We also detailed the advantages and limitations of each of the three varia-

tions.

Extensive evaluation on 56 datasets demonstrated the merits of our framework. In the task of classification, we have shown that DeepLine outperforms current state-of-the-art AutoML systems and popular ML models such as Random Forest and XGBoost. Furthermore, we achieved these results with a significant reduction in the number of evaluated pipelines per dataset compared to other methods. This improvement was possible due to the fact that DeepLine can learn offline the pipeline-dataset interaction over multiple datasets, using meta-learning representation of the datasets and pipelines, while other method make all their optimization online. We have also shown that DeepLine is particularly efficient in datasets that other models are more prone to overfit. Analysis of the Hierarchical Step demonstrated a significant enhancement to the DQN convergence rate when using the hierarchical plugin and that all three methods for integrating the plugin with a DQN agent achieve exponential convergence rate.

8.2 Conclusions

The field of Deep Reinforcement Learning has seen an up-rise in the number of works and research in recent years, with many innovative and state-of-the-art algorithms proposed. However, this rise was not matched by successful use cases of DRL-based applications for real-world problems, as most of the research was directed and tested on simulated environments, such as computer games. In our research, we have shown that DRL methods can be used for a real-world sequential decision making problem that until recently was only performed by human experts.

We proposed DeepLine, a DRL-based framework for the automatic generation of machine learning pipelines. We saw that by using stat-of-the-art DRL algorithms along with methods from other ML fields, such as clustering methods and deep temporal models (with LSTM) we can successfully create a framework to automatically generate ML pipelines. Moreover, we demonstrated the advantage of using deep neural networks in RL by showing how its generalization power can be used for offline learning when resources are abundant, and efficiently deployed for online use with no additional optimization. This advantage was especially useful in the task of AutoML, where current state-of-the-art methods require great deal of computation for the online optimization process.

We demonstrated how the problem of large and dynamic action spaces, which is encountered in some of the real-world RL tasks, can be solved with our novel Hierarchical Step approach. We also saw how our approach can be applied successfully with very little effort on any DRL agent and with an added benefit of enhanced convergence rate and improved exploration process during the agent's learning phase.

For future work, we suggest the following improvements and extensions: (1) exploring the possibility of including Hyper-parameters search as part of our framework, either by adding hyper-parameters optimization models as primitives, or by extending our definition of the *pipeline-step* to also include hyperparameters; (2) extend our framework to other ML tasks such as regression, which will require the development of a more advanced reward function; (3) Applying our framework for other data types, e.g. temporal data. This requires the extension of our primitives set to include temporal models, and; (4) exploring other DRL algorithms and methods for solving our environment, for example with meta-learning.

Appendices

A Datasets

Table A.1: Details of the 56 datasets used for the evaluation, including the number of instances in each dataset, number of features, percentage of categorical features, percentage of missing values and number of labels in the target (number of classes).

dataset	# Inst.	# Feat.	% Categ.	% Missing	# Labels
Accident_Casualties	7500	14	0	0	4
adult	7500	15	60	0	2
analcatdata_broadwaymult	285	8	50	1.2	7
analcatdata_germangss	400	6	66.7	0	17
ar4	107	30	0	0	2
bank-full	7500	15	53.3	0	2
baseball	1340	17	5.9	0.1	3
biodeg	1055	42	2.4	0	2
blood-transfusion	748	5	0	0	2
bodyfat	252	15	6.7	0	2
braziltourism	412	9	0	2.6	7

Table A.1 continued from previous page

bureau	7500	302	5.3	30.6	2
car	1728	7	100	0	4
chatfield_4	235	13	7.7	0	2
cmc	1473	10	0	0	3
crx	690	16	62.5	0.6	2
data_banknote_authentication	1372	5	0	0	2
dermatology	365	35	0	0.1	6
diggle_table_a2	310	9	11.1	0	2
disclosure_z	662	4	25	0	2
Frogs_MFCCs_family	7195	23	4.3	0	4
Frogs_MFCCs_gen	7195	23	4.3	0	8
Frogs_MFCCs_spec	7195	23	4.3	0	10
glass	214	10	0	0	6
haberman	306	4	0	0	2
home_credit	7500	343	0	19.3	2
HTRU_2	7500	9	0	0	2
image segmentation	2310	20	5	0	7
Indian Liver Patient	583	11	0	0.1	2
Iris	150	5	20	0	3
kc3	458	40	0	0	2
kidney	76	7	42.9	0	2
LendingClub Issued Loans	7500	53	22.6	3	12
magic04	7500	11	9.1	0	2
mammographic_masses	961	6	0	2.8	2

Table A.1 continued from previous page

movement_libras	360	91	0	0	15
no2	500	8	12.5	0	2
plasma_retinol	315	14	28.6	0	2
pm10	500	8	12.5	0	2
schizo	340	14	21.4	17.5	2
Skin_NonSkin	7500	4	0	0	2
socmob	1156	6	83.3	0	2
solar-flare	1066	13	23.1	0	6
bng_cmc	55296	10	0	0	3
breast	699	10	10	0.2	2
credit	1000	21	66.7	0	2
eucalyptus	736	20	30	3	5
ilpd	583	11	9.1	0	2
irish	500	6	66.7	1.1	2
phoneme	5404	6	0	0	2
The_broken_machine	7500	59	0	9.4	2
triazines	186	61	1.6	0	2
veteran	137	8	12.5	0	2
weatherAUS	7500	25	24	9.1	2
wilt	4839	6	16.7	0	2
Wine_classification	178	14	0	0	3

Bibliography

[1] Alpayadin, E. (2004). *Introduction to machine learning*. MIT Press.

[2] Anschel, O., Baram, N., and Shimkin, N. (2016). Averaged-dqn: Variance reduction and stabilization for deep reinforcement learning. *arXiv preprint arXiv:1611.01929*.

[3] Anthony, T., Tian, Z., and Barber, D. (2017). Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, pages 5360–5370.

[4] Baker, B., Gupta, O., Naik, N., and Raskar, R. (2016). Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*.

[5] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

[6] Bello, I., Zoph, B., Vasudevan, V., and Le, Q. V. (2017). Neural optimizer search with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 459–468. JMLR. org.

[7] Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305.

[8] Bergstra, J., Yamins, D., and Cox, D. D. (2013). Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in Science Conference*, pages 13–20. Citeseer.

[9] Bertsekas, D. P. et al. (1996). *Dynamic programming and optimal control*, volume 1. Athena Scientific Belmont, Massachusetts.

[10] Brazdil, P., Carrier, C. G., Soares, C., and Vilalta, R. (2008). *Metalearning: Applications to data mining*. Springer Science & Business Media.

[11] Chandak, Y., Theocharous, G., Kostas, J., Jordan, S., and Thomas, P. S. (2019). Learning action representations for reinforcement learning. *arXiv preprint arXiv:1902.00183*.

[12] Chen, B., Wu, H., Mo, W., Chattopadhyay, I., and Lipson, H. (2018). Autostacker: A compositional evolutionary learning system. *arXiv preprint arXiv:1803.00684*.

[13] Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM.

[14] Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014a). On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

[15] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014b). Learning phrase represen-

tations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

[16] Choi, E., Bahadori, M. T., Schuetz, A., Stewart, W. F., and Sun, J. (2016). Doctor ai: Predicting clinical events via recurrent neural networks. In *Machine Learning for Healthcare Conference*, pages 301–318.

[17] Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM*, 55:78–87.

[18] Drori, I., Krishnamurthy, Y., Lourenco, R., Rampin, R., Cho, K., Silva, C., and Freire, J. (2019). Automatic machine learning by pipeline synthesis using model-based reinforcement learning and a grammar. *arXiv preprint arXiv:1905.10345*.

[19] Drori, I., Krishnamurthy, Y., Rampin, R., de Paula Lourenco, R., Ono, J. P., Cho, K., Silva, C., and Freire, J. (2018). Alphad3m: Machine learning pipeline synthesis. In *AutoML Workshop at ICML*.

[20] Dulac-Arnold, G., Evans, R., van Hasselt, H., Sunehag, P., Lillicrap, T., Hunt, J., Mann, T., Weber, T., Degris, T., and Coppin, B. (2015). Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*.

[21] Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., and Hutter, F. (2015a). Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970.

[22] Feurer, M., Springenberg, J. T., and Hutter, F. (2015b). Initializing bayesian hyperparameter optimization via meta-learning. In *AAAI*, pages 1128–1135.

[23] Geurts, P., Ernst, D., and Wehenkel, L. (2006). Extremely randomized trees. *Machine learning*, 63(1):3–42.

[24] Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE.

[25] Greensmith, E., Bartlett, P. L., and Baxter, J. (2004). Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5(Nov):1471–1530.

[26] Hartigan, J. A. and Wong, M. A. (1979). Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108.

[27] He, F. S., Liu, Y., Schwing, A. G., and Peng, J. (2016). Learning to play in a day: Faster deep reinforcement learning by optimality tightening. *arXiv preprint arXiv:1611.01606*.

[28] He, X., Liao, L., Zhang, H., Nie, L., Hu, X., and Chua, T.-S. (2017). Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182. International World Wide Web Conferences Steering Committee.

[29] Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. (2018). Stable baselines. `https://github.com/hill-a/stable-baselines`.

[30] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

[31] Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer.

[32] Hutter, F., Lücke, J., and Schmidt-Thieme, L. (2015). Beyond manual tuning of hyperparameters. *KI-Künstliche Intelligenz*, 29(4):329–337.

[33] Kakade, S. M. (2002). A natural policy gradient. In *Advances in neural information processing systems*, pages 1531–1538.

[34] Kanter, J. M. and Veeramachaneni, K. (2015). Deep feature synthesis: Towards automating data science endeavors. In *Data Science and Advanced Analytics (DSAA), 2015. 36678 2015. IEEE International Conference on*, pages 1–10. IEEE.

[35] Katz, G., Shin, E. C. R., and Song, D. (2016). Explorekit: Automatic feature generation and selection. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 979–984. IEEE.

[36] Katz, G., Shin, E. C. R., and Song, D. (2017). ExploreKit: Automatic feature generation and selection. *Proc. - IEEE Int. Conf. Data Mining, ICDM*, pages 979–984.

[37] Kearns, M. and Singh, S. (2002). Near-optimal reinforcement learning in polynomial time. *Machine learning*, 49(2-3):209–232.

[38] Kotthoff, L., Thornton, C., Hoos, H. H., Hutter, F., and Leyton-Brown, K. (2017). Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *The Journal of Machine Learning Research*, 18(1):826–830.

[39] Liaw, A., Wiener, M., et al. (2002). Classification and regression by randomforest. *R news*, 2(3):18–22.

[40] Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321.

[41] Malhotra, P., Vig, L., Shroff, G., and Agarwal, P. (2015). Long short term memory networks for anomaly detection in time series. In *Proceedings*, page 89. Presses universitaires de Louvain.

[42] Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., and Byers, A. H. (2011). Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute.

[43] McAleer, S., Agostinelli, F., Shmakov, A., and Baldi, P. (2018). Solving the rubik's cube without human knowledge. *arXiv preprint arXiv:1805.07470*.

[44] Medsker, L. and Jain, L. (2001). Recurrent neural networks. *Design and Applications*, 5.

[45] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937.

[46] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529.

[47] Mohammed, M. and Pathan, A.-S. K. (2013). *Automatic Defense Against Zero-day Polymorphic Worms in Communication Networks*. Auerbach Publications, Boston, MA, USA.

[48] Olson, R. S., Bartley, N., Urbanowicz, R. J., and Moore, J. H. (2016). Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 485–492. ACM.

[49] Olson, R. S. and Moore, J. H. (2016). Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on Automatic Machine Learning*, pages 66–74.

[50] Osband, I., Blundell, C., Pritzel, A., and Van Roy, B. (2016). Deep exploration via bootstrapped dqn. In *Advances in neural information processing systems*, pages 4026–4034.

[51] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011a). Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830.

[52] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011b). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[53] Plappert, M. (2017). *Parameter Space Noise for Exploration in Deep Reinforcement Learning*. PhD thesis, Karlsruhe Institute of Technology.

[54] Puterman, M. (1994). Markov decision processes. 1994. *Jhon Wiley & Sons, New Jersey*.

[55] Rakotoarison, H., Schoenauer, M., and Sebag, M. (2019). Automated machine learning with monte-carlo tree search (extended version). *arXiv preprint arXiv:1906.00170*.

[56] Raschka, S. (2016). Mlxtend.

[57] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386.

[58] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

[59] Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015). Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897.

[60] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

[61] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

[62] Singh, S., Jaakkola, T., Littman, M. L., and Szepesvári, C. (2000). Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine learning*, 38(3):287–308.

[63] Smith-Miles, K. A. (2009). Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys (CSUR)*, 41(1):6.

[64] Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959.

[65] Springenberg, J. T., Klein, A., Falkner, S., and Hutter, F. (2016). Bayesian optimization with robust bayesian neural networks. In *Advances in Neural Information Processing Systems*, pages 4134–4142.

[66] Sundermeyer, M., Schlüter, R., and Ney, H. (2012). Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*.

[67] Sutton, R. S. and Barto, A. G. (1998). *Introduction to reinforcement learning*, volume 135. MIT press Cambridge.

[68] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

[69] Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063.

[70] Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013). Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855. ACM.

[71] Tsitsiklis, J. N. and Van Roy, B. (1997). Analysis of temporal-diffference learning with function approximation. In *Advances in neural information processing systems*, pages 1075–1081.

[72] Vainshtein, R., Greenstein-Messica, A., Katz, G., Shapira, B., and Rokach, L. (2018). A Hybrid Approach for Automatic Model Recommendation. pages 1623–1626.

[73] Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *AAAI*, volume 2, page 5. Phoenix, AZ.

[74] Wang, J., Yang, Y., Mao, J., Huang, Z., Huang, C., and Xu, W. (2016a). Cnn-rnn: A unified framework for multi-label image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2285–2294.

[75] Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. (2016b). Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*.

[76] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and De Freitas, N. (2015). Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*.

[77] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.

[78] Zahavy, T., Haroush, M., Merlis, N., Mankowitz, D. J., and Mannor, S. (2018). Learn what not to learn: Action elimination with deep rein-

forcement learning. In *Advances in Neural Information Processing Systems*, pages 3562–3573.

[79] Zhong, Z., Yan, J., and Liu, C.-L. (2017). Practical network blocks design with q-learning. *arXiv preprint arXiv:1708.05552*.

[80] Zhong, Z., Yang, Z., Deng, B., Yan, J., Wu, W., Shao, J., and Liu, C.-L. (2018). Blockqnn: Efficient block-wise neural network architecture generation. *arXiv preprint arXiv:1808.05584*.

[81] Zutty, J., Long, D., Adams, H., Bennett, G., and Baxter, C. (2015). Multiple objective vector-based genetic programming using human-derived primitives. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1127–1134. ACM.