Karlsruhe Institute of Technology

AutoGFE - Automatic Feature Engineering with Graph Neural Networks

Master's Thesis
by

Tobias Meisel

Chair of Pervasive Computing Systems/TECO
Institute of Telematics
Department of Informatics

First Reviewer:	Prof. Dr. Michael Beigl
Second Reviewer:	Yexu Zhou
Supervisor:	Yiran Huang

Project Period: 12/07/2021 – 12/02/2022

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, den 10. Februar 2022

Zusammenfassung

Die Transformation eines Datensatzes zur Verbesserung der Leistung eines Klassifikators oder Regressors ist ein Kernbestandteil vieler Data Science-Projekte. Dieser Prozess ist gemeinhin als Feature Engineering bekannt. Mehrere Algorithmen wurden mit dem Anspruch veröffentlicht, Feature Engineering automatisieren zu können. Die meisten dieser Algorithmen berücksichtigen dabei nur den gerade aktiven Datensatz sowie Informationen die implizit während des Trainingsprozesses gesammelt wurden. Sie berücksichtigen meist keine Information die während des aktuellen Feature Engineering-Prozesses gesammelt wurden. Typischerweise berücksichtigen solche Algorithmen auch keine Domänen-spezifische Informationen. Mit dieser Arbeit präsentieren wir einen neuartigen Algorithmus für das Feature Engineering. Wir repräsentieren Datensätze und Transformationen in einer Baumstrukturen und nutzen Graph Neuronale Netzwerke um Informationen die während der Durchquerung dieser Baumstruktur gesammelt werden um die Effizienz des Algorithmus zu steigern. Wir nutzen Reinforcement Learning um unseren Algorithmus dynamisch zu trainineren ohne große Mengen von Trainingsdaten vorbereiten zu müssen. Wir stellen Kombonenten vor, um zusätzliche Informationen einzubinden. Diese Informationen können von unserem Modell benutzt werden um dessen Leistung weiter zu verbessern. Unser Modell ist leistungsstark und flexibel im Aufbau.
Unsere Evaluation ergab eine gute Leistung auch ohne die zusätzlichen Komponenten. Beide zusätzliche Komponenten steigern die Leistung des Algorithmus zusätzlich. Die beste Leistung wird erzielt, wenn alle Komponenten im Einsatz sind. Die Leistung unseres Algorithmus ist auf Augenhöhe mit anderen State-of-the-Art-Algorithmen und übertrifft deren Leistung auf einigen Datensätzen.
Mit diesem neuen Ansatz präsentieren wir ein Model das die Leistung und notwendige Komplexität in vielen Data Science Projekten verbessert, besonders wenn domänenspezifisches Wissen verfügbar ist. Unser Modell ist flexibel und zusätzliche Komponenten können nach Bedarf hinzugefügt und entfernt werden.

Abstract

The task of transforming base datasets in a way that increases some measure of performance in a classification or regression task is a core component in most Data Science projects. This process is commonly known as Feature Engineering. Several algorithms have been proposed to automate this process, most of which only make decisions based on the currently active dataset enhanced with information implicitly obtained during training but do not explicitly use any information previously obtained during the current process of Feature Engineering. Typically such algorithms also do not capture known domain-specific knowledge.

In this paper we propose a novel algorithm for Feature Engineering. We represent datasets and transformations in a tree structure and use Graph Neural Networks to efficiently traverse it. We use Reinforcement Learning to dynamically train our algorithm without the need for large amounts of training data. We introduce additional components to capture additional information. These additional components help to improve the performance of Feature Engineering. We found that our baseline algorithm performs well on many datasets. The addition of domain-specific knowledge and local context further improves the performance. The best performance is obtained when both local context and domain-specific knowledge is added. Our algorithm is on-par with many state-of-the-art algorithms.

AutoGFE proved to be a successful way to increase performance and decrease necessary complexity in various Data Science pipelines. It is especially useful when domain-specific experience is already available. AutoGFE's architecture is highly flexible and several components may be added or removed depending on circumstances.

Contents

1. Introduction

Feature Engineering (FE) is an essential and often overlooked task in most Data Science pipelines and fundamental to the successful completion of any project. It encompasses the task of transforming dataset to obtain a suitable representation that is useful when the data is used in subsequent Machine Learning (ML) algorithms. The typical Data Science pipeline goes as following: Data is obtained from surveys, sensors or similar sources. After it has been cleaned from nonsensical values, outliers and incomplete datapoints, it is transformed using Feature Engineering. The newly transformed data is then used as input for *Machine Learning* (ML) algorithms such as regressors and classifiers. This process is illustrated in figure 1.1 on page 1. Traditionally, the steps of data cleaning and Feature Engineering are often performed manually while the training of ML models is automated or at least semi-automated. In recent years however, steps have been taken to automated the data cleaning[26] an FE [24] steps. The difficulty in automating FE is the need for knowledge both of the theoretical background of FE as well as the specific domain from which the data stems. The sets of applicable and useful transformations differ vastly between different domains or even within the same domain when the set of given features varies. A transformation that is useful and provides insightful information in one

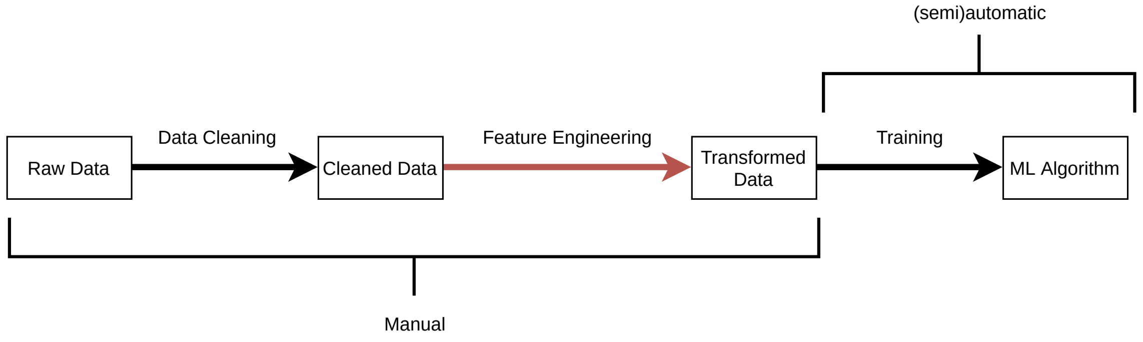

Figure 1.1: A traditional Data Science Pipeline; Raw data is cleaned, transformed and then used to train an ML algorithm. The process of preparing the data necessitates manual labor, while the training process is often automatic or at least semi-automatic.

domain might be completely useless in another. An example: Assume we want to predict whether some dairy product is expired or not. We do not know the expiry date but we do know the date of production. In such a case, we can reasonably assume that a transformation that takes as input a date and produces as output the difference of the input date and the current date in days might prove useful. On the other hand this transformation is not at all useful when predicting the size of some object given its weight. It is also useless when no date of production is given. This example demonstrates some of the difficulties of Feature Engineering.

Despite these problems, FE is worthwhile. The performance of subsequent ML algorithm depends heavily on the quality of the FE step to increase performance as previous works on FE have shown [24][33][23]. On the other hand, bad Feature Engineering, either by automatic models or by data scientists unfamiliar with the domain in question, can cripple ML models in terms of performance and might even prove to make classification or regression impossible when important information is unwittingly discarded.
In practical applications Feature Engineering is done by professionals that do not only have a good overview over the differing FE techniques that can be applied on different features but also have the domain-specific knowledge to be able to evaluate which features might provide useful information and in which way they should be transformed. This however leads to a considerable expenditure of man-hours and might even require the employment of external help and the expenses associated with that. As such, the need for competent automatisation becomes apparent.

Several approaches to facilitate automatic FE have previously been proposed. Such approaches typically model FE as a search in a data structure designed to accommodate the various properties of FE. The search in this data structure is often guided either by fixed heuristics, by training a Machine Learning algorithm to make favorable predictions or are randomized. We will survey those previously proposed methods in greater detail in chapter 2, starting on page 5.
One of those specialized data structures mentioned before is a modified tree structure known as the transformation tree [24]. Transformation trees are a flexible and powerful datastructure, that captures both individual datasets as well as the relations between them. Many modern state-of-the-art approaches explicitly represent the task of Feature Engineering as a search in the space of features ([24],[23]) or can be reformulated as such ([30],[9]).
Agents that traverse the transformation tree to find suitable sequences of transformations are varied in scope and complexity. They utilize concepts such as heuristic or randomized searches[9], rule-based application of transformations[20] or learned heuristics[23]. To the best of our knowledge all approaches however fail to use two important types of information, I) domain-specific knowledge and II) local context information. By I) we mean that they fail to take into account any type of information that has been gained by previous applications of Feature Engineering on datasets in the same domain. One could argue that this is generally not a problem for learned algorithms as their training data could be limited to only include in-domain datasets. However this would severely limit both the amount of training data and the generality of the learned model. By II) we mean that all previously proposed algorithms fail to include information that has already been obtained during the current task of Feature Engineering. Algorithms that have already applied a

function to little effect are not deterred from trying another but similar function. An example for what we mean is the following: Consider a feature for which, for some specific application, it is only relevant whether the absolute value of said feature is larger than one. In this case, both the transformation that maps a feature on its absolute value and the transformation that takes the square of a feature might perform similarly. By discovering and exploiting correlations such as the one just outlined we strive to decrease the complexity of the search space. We hope that utilizing this we can improve the overall performance of the resulting dataset.
Our hypothesis is, that by using Graph Neural Networks and Reinforcement Learning techniques we can improve the performance of Feature Engineering compared to existing approaches.
The importance of our work therefore is to introduce an algorithm, called AutoGFE, that can traverse the transformation tree to find good sequences of dataset representations. AutoGFe that can additionally leverage existing domain-specific knowledge as well as runtime experience to better explore the search space. By this, AutoGFE can hopefully find a good solution which enables good performance on Machine Learning algorithms further down the pipeline. We design AutoGFE to be highly flexible and composable, with multiple components that can be added or removed as need be.

This paper is organized in seven chapters. We start with an *Introduction*. We follow this up with a chapter on *Related Work* in which we go into detail on similar algorithms and the evolution of methods we are using. In the *Background* chapter we introduce notations and methods we need to establish the algorithm later on. After this, in the *Design* section, we discuss the general architecture of the algorithm, the training environment as well as give some notes on implementational details. In the *Experiment* section we discuss how we evaluated our algorithm and how the experiments were set up. In the *Evaluation* section we discuss the results of the experiments as well as the performance of our algorithm and compare it to other State-of-the-Art algorithms. Finally, in the *Conclusion and Future Work* section we give conclusions and offer some words on further development.

We propose an extensible algorithm using Reinforcement Learning and Graph Neural Networks called AutoGFE. AutoGFE is based on the concept of transformation trees and uses Neural Networks to learn to make advantageous decisions based on the current dataset. We evaluate several sets of parameters. Our approach is agnostic to many parameters such as choice of possible transformations, number of transformations and complexity of datasets.
We compare performance both with and without the additional components.

The code associated with this thesis can be found on GitHub:
https://github.com/meisto/AutoGFE.

2. Related Work

2.1 Automatic Feature Engineering

FICUS (Markovitch and Rosenstein, 2002, [30]) utilizes searching in the space of possible features. Its selection of candidate features is based on an evolutionary approach. FICUS uses a set of predefined transformations to generate new features and a performance measure based on the information gain in a decision tree to select which features to keep. *FCTREE* (Fan et al., 2010, [9]) randomly generates features and selects whether to keep the original feature or the newly generated feature based on one of several decision-tree-based metrics.
FEADIS (*FEAture DIScovery*, Dor and Reich, 2012, [5]) performs feature engineering by iteratively and randomly constructing new features by applying transformations. To ensure a feasible number of features, feature selection is additionally applied to remove redundant or confusing features. The *Data Science Machine* (Kanter and Veeramachaneni, 2015, [20]) encompasses a component algorithm, called *Deep Feature Synthesis*, to automatically generate new features. Transformations are applied to all applicable features at once and the result is pruned using feature selection. Such approaches require a considerable amount of evaluation steps during feature generation and selection. Our approach uses experience to circumvent this problem.
ExploreKIT (Katz et al., 2016, [21]) uses an iterative approach with Machine Learning techniques. It generates candidate features which are ranked based on performance. The best performing candidate feature is added to the dataset. This process is repeated until a certain number of steps has been reached.
Cognito (Khurana et al., 2016, [24]) first introduced the concept of representing datasets and features as transformation trees. Khurana et al. proposed several non-heuristic search algorithms, such as depth-first search, global search and constrained search to traverse the transformation tree. This dependence on non-heuristic searches does not allow prioritization of transformations based on experience during the search. This increases the potential search space and therefore decreases the likelihood of finding a well-performing set of transformations.
LFE (*Learning Feature Engineering*, Nargesian et al., 2017, [33]) uses one Multilayer Perceptrons per transformations to predict scores for transformation-feature pairs.

They generate new features by evaluating the pair with the highest score. This approach fails to take sequential decisions into account and suffers from the large amount of pre-generated training data that is necessary for training.
Kuhrana et al. (2018, [23]) propose a Feature Engineering algorithm based on Reinforcement Learning in the transformation tree. Their approach utilizes Q-learning with function approximattors to learn heuristics for a targeted exploration of the transformation tree. This fails to take into account domain-specific knowledge as well as experience obtained during Feature Engineering.

Certain Machine Learning classification algorithms implicitly perform Feature Engineering, such as Artificial Neural Network, Random Forests and Kernel Methods. Especially Artificial Neural Networks have proven to be very powerful in many applications and represent many of the State-Of-the-Art algorithms in fields such as Image Classification, Automatic Speech Recognition and time series analysis. The downside to these powerful algorithms is the need for large quantities of data for training as well as the associated expenses in time and hardware. Such models with large numbers of parameters often require extended periods of training. In contrast, our algorithm does not require as much resources for training and also performs well for use cases with a limited number of training data. Another downside of implicitly performing Feature Engineering is the lack of possibilities for the transformation process to be analyzed and evaluated by a human expert. AutoGFE transforms features on the basis of atomar and well-known mathematical functions and as such can be easily analyzed and understood by human observers.

2.2 Deep Reinforcement Learning

A core component of Reinforcement Learning is the concept of *Markov Decision Processes* (Howard, 1960, [17]). It offers a framework to model and evaluate agents in a precisely specified environment.
The REINFORCE algorithm (Williams, 1992, [53]) approaches the learning of a Reinforcement Learning agent by optimizing an agent using the gradient descent method. The algorithm is formulated in such a way that a gradient can be estimated using the empiric mean over a number simulation rollouts.
TRPO (*Trust Region Policy Optimization*, Schulman et al., 2015, [44]) also optimize their agent using gradient ascent but limit the possible step size by adding additional constraints. This protects the agent from varying too much between gradient updates. *PPO* (*Proximal Policy Optimization*, Schulman et al., 2017, [43]) extends TRPO by replacing the hard constraints on gradient updates with optimization penalties to improve performance.

2.3 Graph Neural Networks

Graph Neural Networks are a class of Artificial Neural Network architectures specifically designed to be able to operate on graph data [41]. First introduced by Scarselli et al. (2008, [42]), Graph Neural Networks exist as a generalization of pre-existing Neural Networks. Each node in a graph is represented by a vector. Graph Neural Networks update node values based on their own value and the values of their

neighbours in the graph.
Interaction Network (Battaglia et al., 2016, [1]) is a specialized GNN architecture to model complex physical systems over time. Battaglia et al. represent individual entities as nodes in a graph structure and relations between those entities as edges. Updates over time are computed as functions on entities and their neighbourhood. Interaction Networks can also be applied to non-physical use-cases.
GIN (*Graph Isomorphism Networks*, Xu et al., 2019, [55]) is both a Graph Neural Network architecture as well as a framework to evaluate the discriminative capabilities of Graph Neural Network Architectures. GIN convolution layers aggregate node representation and their environment and use Multilayer Perceptrons to transform the aggregated representation. Xu et al. prove that, given a sufficient number of convolutional layers, the GIN architecture is as powerful as the Weisfeiler-Lehman graph isomorphism test[6].

3. Background

3.1 Features, Datasets and Transformations

For this work we will only cover relational datasets, meaning only datasets that can be stored in table-like structures. We will not cover weakly structured or unstructured data, such as text, graphs or trees.
In the context of this paper, a **Feature** is a series of data points as well as associated descriptors. It can be thought of as a column in a table together with its associated metadata such as the specific data type and a number of tags. A tag is a string that denotes some characteristic of the feature. Tags are used to limit the application of transformations to features. A list of tags commonly used in this work can be found in table 3.1. We denote the tags of a feature r as $v_{\text{tags}}(r)$ and the values of the feature as $v(r)$.
Data types describe the type of values that can be found in a feature such as numbers or strings. Some of the different data types can be seen in table 3.2. The datatype of a feature r is denoted as $v_{\text{type}}(r)$.
Tags and datatypes can be used to restrict the transformations that can be applied to the individual features. For example, the tag *ZERO* is used to indicate the presence of zeroes and can prevent functions that are undefined on certain inputs, such as the logarithmic function, from being used.

A **Dataset** is a set of related features. Features are typically related by a common source. We will denote the features of a dataset D as $r \in D$. All feature values in a dataset D have the same dimensionality $\forall r_1, r_2 \in D : |v(r_1)| = |v(r_2)|$. A feature

Tag	Meaning
NONZERO	No value in this column is zero.
POSITIVE	All values are positive.
BOUND	Possible values are bound to a fixed scale.

Table 3.1: Some exemplary tags that might be used to control transformation applicability.

Datatype	Description
Numeric	A general number.
String	A general string of characters.
Date	An object representing a date, can be seen as a structured set of numbers or strings.
Categorical	One class instance of a number of classes.

Table 3.2: Exemplary datatypes a feature might take.

representing the target vector is given for each dataset. We denote the target vector of a dataset D as $v_{\text{target}}(D)$. The target vector has the same dimensionality as all feature vectors, $\forall r \in D : |v_{\text{target}}(D)| = |r|$. The target vector might contain of class labels for a classification task or continuous values for regression tasks.

A **Transformation** is an operation that can be applied to one or more features to produce a new feature when certain criteria are met. Each transformation generates exactly one new feature for each given input. A transformation t has a list of input datatypes $v^{\text{tf}}_{\text{indt}}$, an output datatype $v^{\text{tf}}_{\text{outdt}}$ and a set of (blacklist) tags $v^{\text{tf}}_{\text{tags}}$.
A transformation t is said to be *applicable* to a feature or list of features $(r_i)_1^N$ if the input data types match and the set of tags of the individual input features does not contain tags that occur in the transformations blacklist set of tags.

$$\forall i \in \{1 \dots N\} : \Big(\big(v_{\text{tags}}(f_i) \cap v^{\text{tf}}_{\text{tags}}(t) = \emptyset \big) \wedge \big(v_{\text{indt}}(f_i) = v^{\text{tf}}_{\text{indt}}(t)_i \big) \Big) \tag{3.1}$$

Note that we are using a blacklist-type of approach to limit applicability of functions, where the applicability of a transformation depends on the nonexistence of tags. We can however emulate a whitelist-type approach by introducing negated tags.

3.2 Artificial Neural Networks

Artificial Neural Networks (ANNs)[11] are a powerful class of Machine Learning algorithms. ANNs are notable among Machine Learning algorithms in that they are, given a suitable architecture, both general function approximattors [4][29] and Turing complete[46]. These properties make them especially useful for replacing whole sub-models or components of models with single ANNs. ANNs can be trained end-to-end but often require an extensive amount of both hyperparameter tuning and training data. ANNs are typically trained using the back-propagation algorithm[8]. We denote functions modeled with an ANNs as

$$f_{\boldsymbol{\theta}} : \mathcal{X} \to \mathcal{Y}. \tag{3.2}$$

Here the function f is parameterized by a parameter vector $\boldsymbol{\theta}$. ANN models are typically defined as a number of distinct sub-functions, called *layers*, which are traversed in sequence when an input is supplied. Various layer architectures exits, mostly sharing common elements, but differing in details and functionality. Notable ANN architectures include *Recurrent Neural Networks* (RNNs), *Convolutional Neural Networks* (CNNs) and *Multilayer Perceptrons* (MLPs). The sequential nature of ANNs allows the learning of hierarchical features which has proven to be highly useful in many applications.

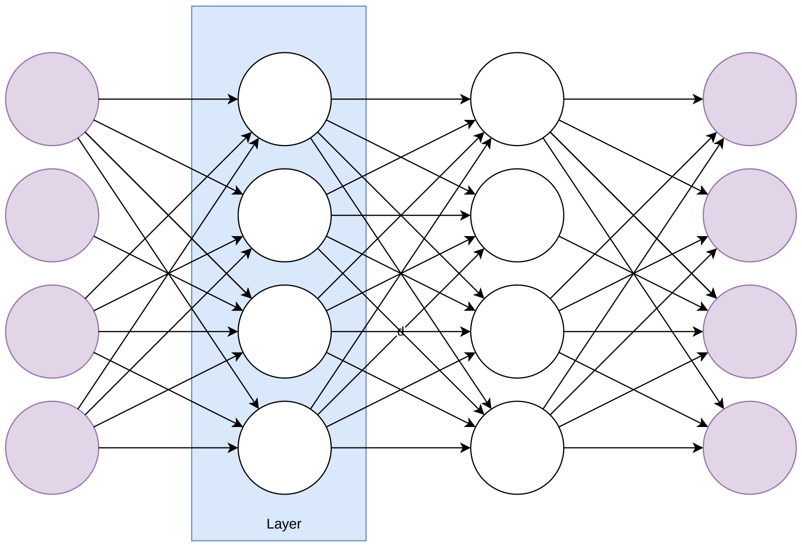

Figure 3.1: Illustration of the layer-based architecture of a type of basic ANN (Multilayer Perceptron). Each nodes performs a linear aggregation and applies a nonlinear function.; Legend: Circles: Neurons, dark grey circles: Input and Output layers respectively.

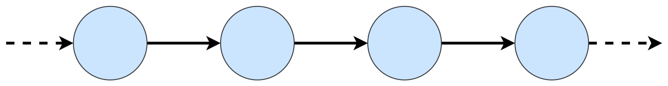

Figure 3.2: Illustration of sequential data as graph. Black arrows indicate dependencies, such as time, trial number, et cetera. Light blue circles indicate singular datapoints, such as word encodings, numerical sensor reading et cetera.

3.3 Graph Neural Networks

While Artificial Neural Network Algorithms have proven to be highly capable in terms of learning a diverse set of functions, historically they have been limited by their narrowly defined set of possible input structures. Traditional Multilayer Perceptrons only accept fixed-sized vectors as input and are not invariant to input permutation (meaning $f_{\text{MLP}}(a, b, c) \neq f_{\text{MLP}}(a, c, b)$). Recurrent Neural Networks, such as *Long short-term memory* (LSTM) networks [16] or *Gated Recurrent Units* (GRUs) [3] have been successfully applied to process sequential data. Convolutional Neural Networks are used widely to process image data (e.g. Alex Net[27]) and represent State-of-the-Art models in many tasks of computer vision.
Both the sequential input of RNNs, as well as the image input of CNNs can be seen as types of strongly regularized graphs. In the case of sequential data, that graph is a ordered list of nodes as seen in illustration 3.2. In the case of image input, each pixel represents a node with the edges lying between neighboring nodes as seen in figure 3.3.

The class of Deep Neural Networks known as *Graph Neural Networks* is a generalization of these more specialized Artificial Neural Network architectures, which is able to process less strictly structured graphs. They generally operate similar to CNNs, alternating between convolutions of node environments and transforming node representations. Wu et al.[54] propose a taxonomy, which divides the class of Graph Neural Networks in four sub-classes: Recurrent Graph Neural Networks, Convolutional Graph Neural Networks, Graph Autoencoders and spatial-temporal

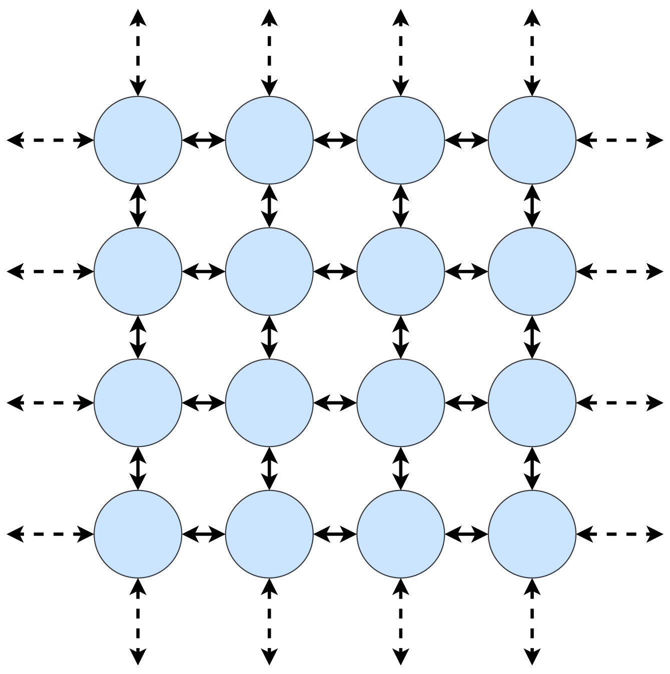

Figure 3.3: Illustration of an image interpreted as a graph. Each node in the graph represents a pixel in the image such as a scalar grey value or an equivalently representation of a pixel, such as a vector containing red, green and blue scalar values.

Graph Neural Networks. We focus on spatial-based convolutional Graph Neural Networks (ConvGNNs) for this paper as they are amongst the most widely known types of GNNs.
Spatial-based ConvGNNs are a direct generalization of the convolutional layers found in CNNs. Convolutions are performed over the spatial neighbourhood of a node using its own representation and the representations of all adjacent nodes. ConvGNNs are typically stacked similarly to most popular CNNs.

For AutoGFE we use the *Graph Isomorphism Network* (GIN)[55] as our choice of ConvGNN architecture. The GIN architecture is both conceptually easy and shown to be sufficiently powerful in comparison to other GNNs[54] making them a obvious choice.
GIN convolutional layers aggregate the neighborhood of a given node as seen in equation 3.3.

$$h_v^{(k)} = \mathrm{MLP}^{(k)} \left(\left(1 + \epsilon^{(k)}\right) h_v^{(k-1)} + \sum_{u \in \mathcal{N}(v)} \left(h_u^{(k-1)}\right) \right) \tag{3.3}$$

Here $\epsilon^{(k)}$ is a tunable hyperparameter, $h_v^{(k)}$ is the representation of node v after the kth convolutional step and $\mathcal{N}$ is the set of neighbours of node v. ϵ may be set as a fixed value before training or learned during training.
In a model with k GIN convolution layers, pairs of nodes that have k or less edges between them are able to influence each other. This enables the propagation of information through graphs. In our case the specific value of k influences the number of explored transformations in the transformation tree that can influence our decision-making process at each time step. Setting a large value k may enable better

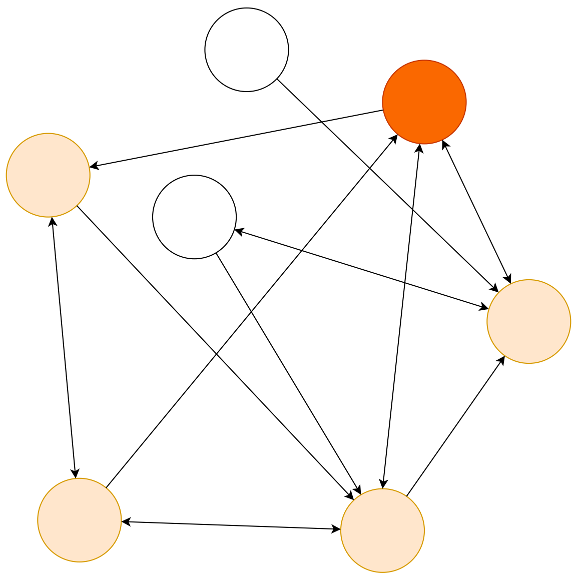

Figure 3.4: Illustration of the neighbourhood of an active node. Note that the neighbourhood of a node must not necessarily be defined as any edge direction but might be restricted to include only certain types of nodes such as nodes with outgoing or incoming edges. Legend: dark orange node: active node, light orange nodes: neighbourhood of active node, white nodes: other nodes.

long-time performance but increases the computational overhead at each timestep. We define the neighborhood of a node u as only those nodes v that have edges going from node u to node v, meaning only nodes that haven been generated from the current node or one of its children by applying transformations. As such the GIN aggregation scheme is limited to only use information contained in the subtree rooted at node v. We further discuss this in chapter 4.

We denote the application of a GIN convolution on a Graph G as $\text{GIN}(N, A)$, where N is the set of all node representations in G and A is the adjacency matrix of the Graph, containing all structural information.

3.4 Reinforcement Learning

Along with *Supervised Learning* and *Unsupervised Learning*, *Reinforcement Learning* is one of the three main training modes in Machine Learning. Reinforcement Learning is defined by the usage of agents in an initially unknown environment that learn to take certain actions in order to maximize some measure of reward. Well-known examples of such agents and their respective environments are software agents learning to play video games[2] and robots learning to perform some task in a physical setting[19][45]. In the case of a software agent learning to play a game the performance measure that is to be maximized is the score of the video game. In the case of a robot the performance measure is usually defined in a goal-oriented way in which each successful sub-step earns a small reward, with a large reward being given upon completion of the whole task[45].
Agents are able to interact with their environment by using a set of well known actions such as button presses (Left, Right, Up, Down, A, B, ...) or by modifying continuous parameters such as the torque applied to engines.

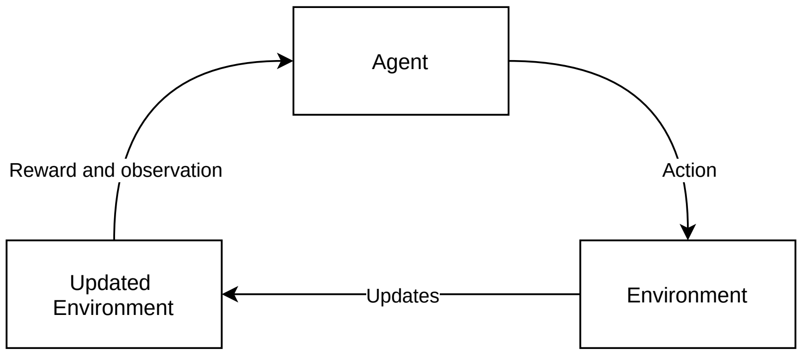

Figure 3.5: The basic Reinforcement Learning loop. An agent takes some action, the environment receives that action and changes states, the agent receives some measure of reward and an observation of the changed environment.

Reinforcement Learning problems are often stated as Markov Decision Processes. A Markov Decision Process (MDP) M is defined as a quintuple of values[47][10]. Let $\mathcal{S} \subseteq \mathbb{R}^N$, $\mathcal{A} \subseteq \mathbb{R}^M$, t be a transition function, r be a reward function and $\gamma \in \mathbb{R}$. Then a MDP M is defined as:

$$M = \big(\mathcal{S}, \mathcal{A}, t, r, \gamma\big) \tag{3.4}$$

$\mathcal{S}$ and $\mathcal{A}$ are the sets containing all possible states or actions of the MDP respectively. The transition function

$$t : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1] \subset \mathbb{R} \tag{3.5}$$

maps two states $s_1 \in \mathcal{S}$, $s_2 \in \mathcal{S}$ and an action $a \in \mathcal{A}$ onto a real value p. p is the probability that given a state s_1 and action a being taken, the environment will transition to state s_2, $t(s_{t+1}, a_t, s_t) = p = \mathbb{P}(s_{t+1}|s_t, a_t)$. The reward function

$$r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}^+ \tag{3.6}$$

describes the award that is obtained when action a is taken in state s_1 and the environment transitions to state s_2. The discount factor $\gamma \in \{0 \leq x \leq 1 | x \in \mathbb{R}\}$ determines how much rewards in later timesteps influence the current reward. Setting $\gamma = 0$ will prioritize instant reward. Setting $\gamma = 1$ will prioritize rewards at a later point in time.
MDPs make some assumptions that are overly simplistic for many real-world application. An example for this is the assumption that the agent has perfect knowledge of the state of the environment. This is in many cases not true, such as in robotics where the environment is perceived via a set of imperfect sensors. For our usecase this assumptions holds.

An Reinforcement Learning agent then, is a function, usually denoted as π, that learns to maximize the total expected reward $\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t, s_{t+1})$ over time. An agent π might either be deterministic $\pi : \mathcal{S} \rightarrow \mathcal{A}$ or stochastic $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$. A stochastic agent can easily be transformed in an deterministic one by $\pi^{\text{det}}(s) = \arg\max_a(\pi^{\text{stoch}}(s, a))$.
The expected reward in a state s is denoted as $V^\pi : \mathcal{S} \rightarrow \mathbb{R}$.

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t, s_{t+1}) | a_t \sim \pi(s_t), s_{t+1} \sim t(s_t, a_t, s_{t+1}), s_0 = s\right] \tag{3.7}$$

The process of learning for a policy π can be reformulated as the search for an optimal set of parameters $\boldsymbol{\theta}$ given a starting state s_0.

$$\pi^* = \arg\max_{\pi \in \Pi} V^\pi(s_0) \tag{3.8}$$

Related to the V^π function is the Q^π function which gives the expected reward when an action a is taken while in state s.

$$Q^\pi(s,a) = \mathbb{E}\left[\sum_{t=0}^{\infty}\left(\gamma^t r(s_t, a_t, s_{t+1})\right) | s_0 = s, a_0 = a\right] \tag{3.9}$$

The *Q-learning* algorithm [52] uses a recursive formulation to define an iterative algorithm that calculates an optimal Q-value function Q^*. Equation 3.9 is restated in a recursive form by using Bellman's equation[10]:

$$Q^\pi(s,a) = \sum_{s' \in \mathcal{S}} \left(t(s,a,s')\left(r(s,a,s') + \gamma Q^\pi(s',a')\right)\right), a' \sim \pi(s') \tag{3.10}$$

Q-learning stores the Q-value for each state-action-pair and is therefore unfeasible for problems with large state and/or action spaces. Several derivative algorithms exist to alleviate this problem such as *Neural Fitted Q Iteration*[38] and *Deep Q Networks*[31]. There exist a plethora of other Reinforcement Learning Algorithms that try to tackle this problem. Some algorithm model the reward function V directly to evaluate states and make decision based on this[47]. Some algorithms learn an approximate model of the environment that is then used to make local searches for optima[10]. Another class of models directly approximates policies π [47].

3.5 Deep Reinforcement Learning

Deep Reinforcement Learning are Reinforcement Learning algorithms that use Artificial Neural Networks to replace or supplement some or all components of more traditional RL algorithms. Many different approaches with various degrees of ANN inclusion exist.
Most of of these approaches require specialized problem statements, such as discrete actions, continuous state spaces et cetera. In the following section we will provide a list of Deep Reinforcement Learning algorithms that can be applied to our problem statement.

Deep Q Network (DQN)[31] is an extension of the standard Q-learning algorithm that uses ANNs to approximate the Q-value function. Naively modeling ANNs to model Q-value functions proved problematic, suffering from instability issues[31]. Mnih et al.[31] introduced replay buffers, delayed updating and other measures to stabilize and improve the algorithm. A replay buffer is the a set of the last (s, a, r, s') tuples observed, where s is the initial state, a is the action taken, s' is the new state and r is the received reward. During training the Q-value approximator is trained on random batches sampled from the replay buffer. A rough outline can be seen in figure 3.6.

A class of RL algorithms that directly learn a policy π is commonly known as **Policy Gradient Methods**[10]. Algorithms of this class optimize the total expected reward

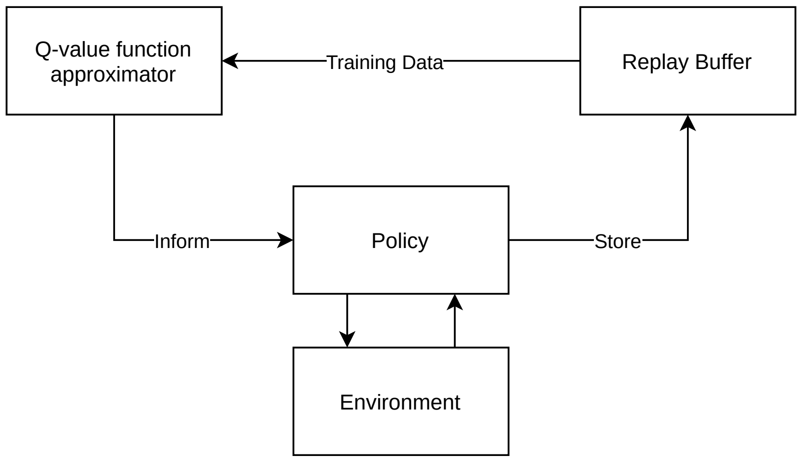

Figure 3.6: Basic overview of the Deep Q Network architecture.

by directly optimizing a policy using the gradient ascent/descent algorithm[39].
To be able to analytically derive a gradient for the policy, equation 3.7 can be reformulated by replacing the expectation value with integrals over all states and actions[10].

$$V^{\pi}(s_0) = \int_{\mathcal{S}} \underbrace{\left(\sum_{t=0}^{\infty} \gamma^t \mathbb{P}(s_t = s | s_0, \pi)\right)}_{\rho^{\pi}(s_0, s)} \left(\int_{\mathcal{A}} \pi(s,a) \left(\int_{\mathcal{S}} t(s,a,s') r(s,a,s') ds'\right) da\right) ds \tag{3.11}$$

Here $\rho^{\pi}(s_0, s)$ is the probability that, given a state s_0 the model ever reaches the state s, with the probability being discounted using some scalar $\gamma \in \mathbb{R}$.
For our cases we assume that the environment in a state s, given an action a, will deterministically transition to another state s', meaning

$$t(s,a,x) = \begin{cases} t(s,a,x) = 1 & \text{if } x = s' \\ t(s,a,x) = 0 & \text{if } x \neq s' \end{cases}$$

. We can therefore simplify equation 3.11 to:

$$V^{\pi}(s_0) = \int_{\mathcal{S}} \rho^{\pi}(s) \left(\int_{\mathcal{A}} \pi(s,a) r(s,a,s') da\right) ds \tag{3.12}$$

If the policy π is parameterized by parameters $\boldsymbol{\theta}$, Sutton et al.[48] prove that the gradient of the value function regarding the parameter vector $\boldsymbol{\theta}$, $\nabla_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}$, can be given as:

$$\nabla_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}} = \int_{\mathcal{S}} \rho^{\pi_{\boldsymbol{\theta}}}(s) \left(\int_{\mathcal{A}} \left(\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(s,a)\right) Q^{\pi_{\boldsymbol{\theta}}}(s,a) da\right) ds \tag{3.13}$$

The gradient of the value function therefore depends only on the gradient of the policy. This mean the expected reward can be improved by optimizing the policy directly. Note that, while we henceforth assume the policy to be implemented as neural networks, the policy can be implemented as any model trainable with gradient ascent/descent.

Williams[53] use equation 3.13 in their **REINFORCE** algorithm. They reformulate the policy gradient as seen in equation 3.14 by exploiting the chain rule and the properties of the logarithmic function.

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s,a) &= \frac{\pi_{\boldsymbol{\theta}}(s,a)}{\pi_{\boldsymbol{\theta}}(s,a)}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s,a) \\ &= \pi_{\boldsymbol{\theta}}(s,a)\frac{\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s,a)}{\pi_{\boldsymbol{\theta}}(s,a)} \\ &= \pi_{\boldsymbol{\theta}}(s,a)\nabla_{\boldsymbol{\theta}}\log\big(\pi_{\boldsymbol{\theta}}(s,a)\big)\end{aligned} \tag{3.14}$$

By inserting equation 3.14 in equation 3.11 they obtain a function that can be reformulated as an expectation value.

$$\nabla_{\boldsymbol{\theta}}V^{\pi_{\boldsymbol{\theta}}} = \mathbb{E}\Big[\nabla_{\boldsymbol{\theta}}\left(\log\pi_{\boldsymbol{\theta}}(s,a)\right)Q^{\pi_{\boldsymbol{\theta}}}(s,a)\Big] \tag{3.15}$$

Equation 3.15 can be effectively implemented by using Monte-Carlo estimators for both the expectation value and the Q-function. In practice this is done by evaluating the policy multiple times in a given environment. In the process a number of trajectories are generated. Each trajectory consists of sequences of states, rewards and decisions. The trajectories are used to estimate the Q-function and to update the policy using gradient ascent/descent. This process is iteratively repeated until convergence.

Proximal Policy Optimization (PPO)[43] is a Deep Reinforcement Learning algorithm related to the REINFORCE algorithm. PPO utilizes Policy Gradient and Trust Region methods. Similar to the REINFORCE algorithm, PPO estimates gradients for policies by simulating the policy several times in an environment. The trajectories produced in the process are used to estimate a gradient and update the agent. PPO replaces the empirical estimate of the Q-value function in equation 3.15 with a function $\hat{A}^{\pi}(s,a)$.

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{3.16}$$

$$\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}} = \mathbb{E}\left[\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(a_t|s_t)\hat{A}_t\right] \tag{3.17}$$

PPO utilizes an additional ANN to estimate the value function V which is trained together with the policy network. Deep RL models that use this two-component architecture are commonly known as Actor-Critic architectures or as Actor-Critic Models[10]. The actor represents a policy function and the critic represents an approximation of the value function. Both models are trained in tandem to increase the overall performance.
In practice, many simple Reinforcement Learning algorithms tend to be sensitive to hyperparameter tuning. If the step size in the gradient-descent algorithm is chosen deficiently, even a small gradient update may lead to a large change in the policy distribution. More advanced models limit the range of possible updates to alleviate this problem. PPO uses so-called Trust Region Methods to facilitate this. Trust Region Methods are a class of methods widely used in mathematical optimization, where a simple model function is introduced to represent some unknown or complex function. This model function is assumed to represent the original function well in

a limited region close to a known evaluation point. PPO's trust region function is based on an earlier model known as *Trust Region Policy Optimization*(TRPO)[44].

$$\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}} = \boldsymbol{\theta}' - \boldsymbol{\theta} = \max_{\boldsymbol{\theta}'} \left[\mathbb{E} \left[\frac{\pi_{\boldsymbol{\theta}'}(a_t|s_t)}{\pi_{\boldsymbol{\theta}}(a_t|s_t)} \hat{A} \right] \right] \tag{3.18}$$

Intuitively this may be interpreted as finding the set of parameters $\boldsymbol{\theta}'$ that maximize the expected advantage value, when that advantage function is weighted by their new probability of occurrence relative to the old probability. To limited the range of the trust region to a region close to a known point $\boldsymbol{\theta}$, TRPO introduces a constraint based on the Kullback-Leibler divergence.

$$\mathbb{E} \left[\mathrm{KL} \Big(\pi_{\boldsymbol{\theta}'}(\cdot|s_t) || \pi_{\boldsymbol{\theta}}(\cdot|s_t) \Big) \right] \leq \delta \tag{3.19}$$

Where KL is the Kullback-Leibler divergence and δ is a fixed scalar value. This constraint limits the updates to a region around the original point based on the Kullback-Leibler divergence of the policy distribution. The addition of this constraint to the optimization problem is however computationally expensive to solve since the Kullback-Leibler divergence has to be evaluated multiple times at each step in the algorithm. As an improvement to this, PPO replaces the hard constraint of TRPO by clipping the gradient and adding penalty terms to the optimization problem.

Let $r_t(\boldsymbol{\theta})$ be the ratio, given a state s_t and an action a_t, between an updated policy and the policy before the update, $r_t(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}'}(a_t|s_t)}{\pi_{\boldsymbol{\theta}}(a_t|s_t)}$. Then the clipped loss $L_{\text{clip},t}$ is an bounded estimation of the gradient at time t.

$$L_{\text{clip},t}(\boldsymbol{\theta}) = \mathbb{E} \left[\min \left\{ r_t(\boldsymbol{\theta}) A^t, \text{clip}(r_t(\boldsymbol{\theta}, 1 - \epsilon, 1 + \epsilon) A_t \right\} \right] \tag{3.20}$$

clip is the clipping function that bounds all values in an input vector to a range $[1 - \epsilon, 1 + \epsilon]$, with ϵ being a hyperparameter. This limits the maximum gradient to a small value and is used to replace TRPO's hard constraint based on the Kullback-Leibler divergence. An additional loss $L_{\text{vf},t}$ is added to train the value function. Let $V_{\boldsymbol{\theta}}(s_t)$ be the predicted value for state s_t at timepoint t and v_t be the target value at timepoint t. The value function loss $L_{\text{vf},t}$ is given as:

$$L_{\text{vf},t} = \left(V_{\boldsymbol{\theta}}(s_t) - v_t \right)^2 \tag{3.21}$$

An additional entropy term $S(\pi_0, s_t)$ is added to the term to regulate the probability distribution of the agent π.
The loss function for the PPO algorithm is then defined as a weighted sum of $L_{\text{clip},t}$, $L_{\text{clip},t}$ and $S(\pi_0, s_t)$. Let c_1 and c_2 be scalar constants.

$$L_t(\boldsymbol{\theta}) = \mathbb{E} \left[L_{\text{clip},t}(\boldsymbol{\theta}) - c_1 L_{\text{vf,t}}(\boldsymbol{\theta}) + c_2 S(\pi_0, s_t) \right] \tag{3.22}$$

The PPO algorithm is widely used in Reinforcement Learning[37] and in the field of Robotics[19]. It can be used both for discrete action spaces and continuous action spaces.

4. Design

We now discuss the architecture of AutoGFE. We start by giving a high-level overview of the algorithm. We then start by discussing the environment, which includes the concept of transformation trees and state the problem of Feature Engineering in the context of Markov Decision Processes (MDPs). Afterwards we cover the agent that receives a dataset as input and produces as transformation to transform the dataset. We then propose additional components to add domain-specific information and local context. Afterwards we discuss the Reinforcement Learning (RL) setting and how to train AutoGFE. Finally we give some short notes on implementational details.

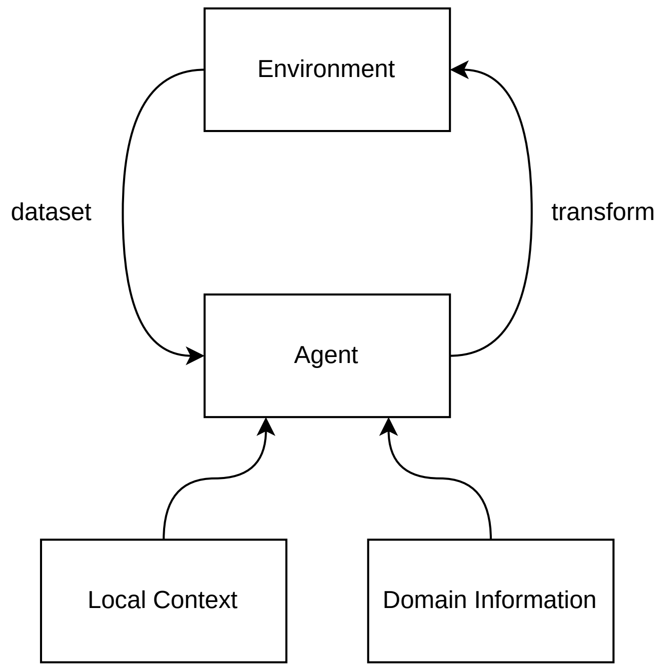

Figure 4.1: Basic loop of our algorithm. The algorithm starts with the initial datasets and loops until a certain performance criteria or a specified number of loops have been reached.

4.1 High-level Overview

For now we assume a trained model, consisting of an agent π. Let D be an exemplary input dataset. AutoGFE receives the dataset as input and produces a transformation function. This transformation function can be applied to transform the original dataset D to produce a new dataset D'. This procedure can be repeated until a certain number of iterations or a pre-defined performance threshold is reached.

The AutoGFE agent π predicts a probability distribution by generating a fixed-length representation $f_{\text{dr}}(D)$ of the dataset D. This dataset representation can be enhanced with additional information as we discuss in a later section. We use the dataset representation $f_{\text{dr}}(D)$ as input to a Multilayer Perceptron with softmax output layer to produce a probability distribution p over the set of transformations. This probability distribution assigns to each transformation t the probability p_t that it will be useful when applied to dataset D. We evaluate the probability distribution to obtain a concrete transformation. The process of evaluating the probability function can be seen in pseudo-code in 2.
The initial dataset D is the transformed using transformation t. These steps are repeated until a good performance is reached or a specified number of transformations has been applied. This basic outline of the algorithm can also be seen as pseudo-code in 1.

Algorithm 1 AutoGFE - Basic

```
1: procedure AutoGFE($D_{\text{start}}, n_{\max}$)
2:     // $D_{start}$ is the initial dataset
3:     // $n_{max}$ is the maximum number of transformations
4:
5:     $D \leftarrow D_{\text{start}}$
6:     $i \leftarrow 0$
7:     while $i < n_{\max}$ do
8:         // Generate a representation of D
9:         $\boldsymbol{x} \leftarrow f_{\text{dr}}(D)$
10:        $\boldsymbol{x} \leftarrow \text{enhance}(\boldsymbol{x}, D)$
11:
12:        $\boldsymbol{p} \leftarrow f_a(\boldsymbol{x})$      $a \in \{\text{PPO}, \text{REINFORCE}\}$
13:
14:        // Obtain a transformation
15:        $t \leftarrow \text{evaluate}(\boldsymbol{p}, D, \text{false})$
16:
17:        // Check if transformation is valid
18:        if $t = -1$ then return D
19:
20:        // Apply transformation
21:        $D \leftarrow apply(D, t)$
22:        $i \leftarrow i + 1$
23:    return $D$
```

The *evaluate* function receives as input a probability and a dataset D and produces a transformation. It also checks whether the transformation can be applied to the dataset. If not, it rescales the probability function and samples anew. If no transformation can be produced, it return a default value and the algorithm terminates.

Algorithm 2 Evaluation of a probability distribution $\boldsymbol{p}$. Evaluation can be done either greedily or stochastic.

```
procedure EVALUATE($\boldsymbol{p}$, $D$)
    $i = 0$
    while $i < |\boldsymbol{p}|$ do
        $t \leftarrow$ sample($\boldsymbol{p}$)

        // Check if D can be transformed by t
        if can_transform($t, D$) then return t

        // remove t as possible transformation
        $\boldsymbol{p}_t = 0$
        for all $j \in \{1 \dots |\boldsymbol{p}|\}$ do
            $\boldsymbol{p}_j = \frac{\boldsymbol{p}_j}{\sum_{k=1}^{|\boldsymbol{p}|} \boldsymbol{p}_k}$

        $i \leftarrow i + 1$

    // If there is no transformation that can be applied to D, return -1
    return $-1$
```

We propose two optional components, f_{local} and f_{domain} to enhance the dataset representation by including local context and domain-specific information respectively. These additional components are fully composable and AutoGFE can be used with none, one or both of them. The pseudo-code for the *enhance* procedure can be found in algorithm 3.

Algorithm 3 Optional enhancement of the dataset representation.

```
procedure ENHANCE($\boldsymbol{x}$, mode, D)
    $f_{\text{local}}$      // Enables additional local context
    $f_{\text{domain}}$     // Enables additional domain information

    if mode = none then
        return $\boldsymbol{x}$
    else if mode = local-context then
        return $f_{\text{local}}(\boldsymbol{x}, D)$
    else if mode = domain-specific then
        return $f_{\text{domain}}(\boldsymbol{x})$
    else if mode = both then
        return $f_{\text{domain}}(f_{\text{local}}(\boldsymbol{x}, D))$
```

4.2 Environment

In this section we discuss the environment, the underlying data structure and how the agent interacts with the environment.
The agent interacts with the environment by either applying a transformation or by deleting a set of features from a dataset. The state of the environment at each point in time is defined by the currently active dataset.

We use the transformation tree as an environment. A transformation tree is a datastructure that captures dependencies between datasets. Our transformation tree follows the outline established in [24] with slight modifications.
Each node in the transformation tree represents an individual dataset. (We henceforth use the term dataset to refer to both a node in the transformation tree as well as to the dataset itself. It should become clear from the context which sense is meant.) The root dataset is the base dataset that before FE. Each edge between nodes represents a relation between datasets. An edge $D_1 \to D_2$ indicates that dataset D_2 has been obtained by modifying dataset D_1 in some way. Edges are either expansion-type or reduction-type. The edges correspond to the actions an agent can take at each point in time.
Expansion-type edges represent transformations that have been applied to one or more sets of ordered features in the parent dataset to generate one or more new features. In this case, the child dataset is the dataset consisting of all features in the parent dataset plus the newly generated feature. Expansion-type edges can be completely specified by a transformation t and a set of lists of features $S = \{(r_i)_1^a \mid \forall i : r_i \in D_{\text{parent}}\}$. The target value of the child dataset is the same as the target value of the parent dataset.

$$\begin{aligned} D_{\text{child}} &= D_{\text{parent}} \cup \{t(s) \mid s \in S\} \\ v_{\text{target}}(D_{\text{child}}) &= v_{\text{target}}(D_{\text{parent}}) \end{aligned} \tag{4.1}$$

Reduction-type edges represent the removal of one or more features from a dataset. The child node of a reduction-type edge is a dataset containing all features from its parent dataset minus the ones defined in the edge. Reduction-type edges are specified by a set of features $S \subseteq D_{\text{parent}}$.

$$D_{\text{child}} = D_{\text{parent}} \setminus S \tag{4.2}$$

Note that the expansion-type edges are defined using lists of features while the reduction-type edges are defined using sets of features. This is relevant as some transformations might not be permutation invariant, such as the transformation representing division ($\frac{1}{2} \neq \frac{2}{1}$). The actual structure of the tree in regards to the types of edges at each layer depends on the agent used to decide which transformation is used at which point in time. This enables the transformation tree to handle both rule-based approaches (which might choose to prune the dataset regularly, for example on every second step) and approaches in which the agent learns to apply transformation. An exemplary transformation tree can be seen in figure 4.2 on page 23.

To compare different datasets and to make statements about which datasets is better in terms of performance we define a measure of performance. In this paper we will

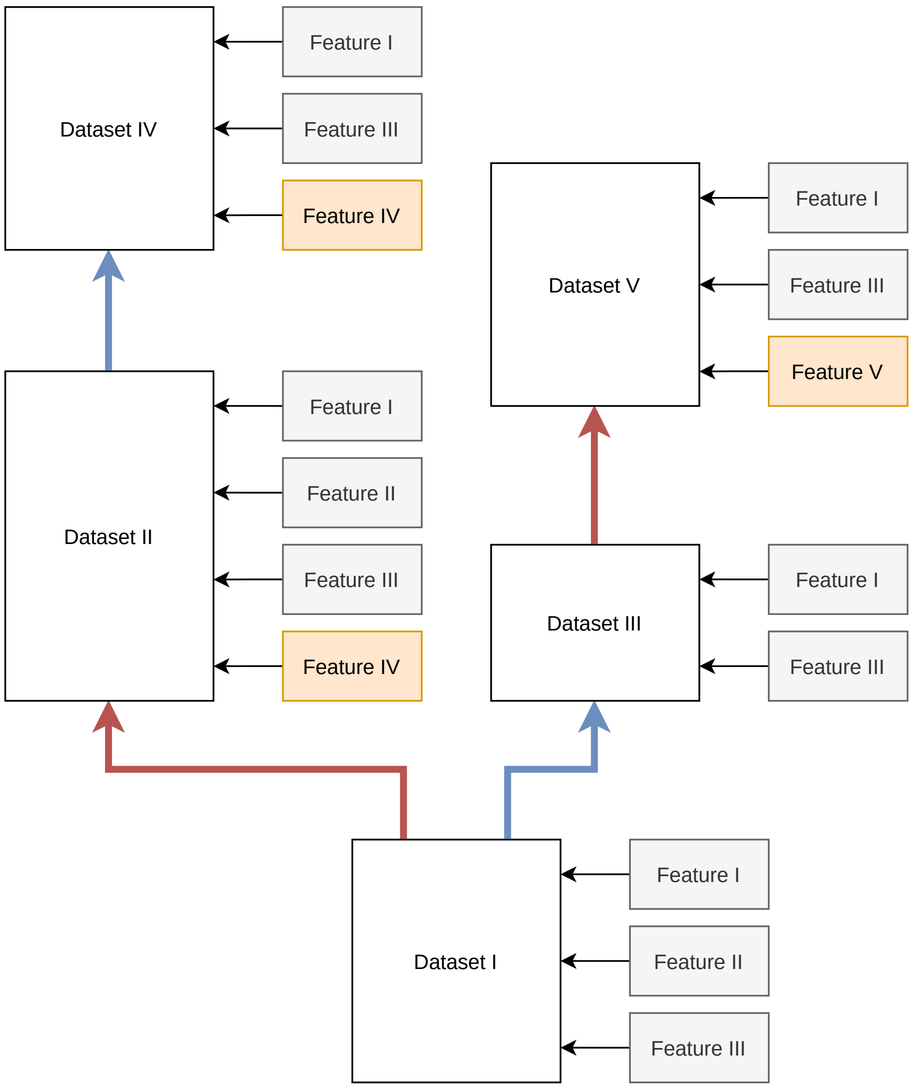

Figure 4.2: Example of a transformation tree; Legend: grey features are raw data, orange features have been transformed, red arrows are edges of the expansion type, blue arrows are edges of the reduction type.

refer to this measure $s : \mathcal{D} \to \mathbb{R}$ as the *scoring function.* We score datasets by the f-score they show on simple classification algorithms after subtracting the f-score of the root dataset to account for the baseline.
We subtract the score of root dataset to remove the dependencies of the score on the base performance of the root dataset. F-scores are calculated using K-fold cross-validation with $K = 8$. As classifiers we use a a Decision Tree model.

$$\text{score}_x(D) = \text{Fscore}_{\text{DT}}(D) - \text{Fscore}_{\text{DT}}(D_{\text{base}}) \tag{4.3}$$

We chose to utilize Decision Trees as an example of classifiers with relatively low complexity because using more complex classifiers, such as Support Vector Machines with kernel functions or Deep Neural Networks, could be seen as the introduction of an additional FE step, which would add bias towards those classifiers.

Transformation trees lend themselves well to be interpreted as Markov Decision Processes. As previously stated MDPs are defined as a quintuple of values $(\mathcal{S}, \mathcal{A}, t, r, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, t is the transition function, r is the reward function an γ is the discount factor.
The state space $\mathcal{S}$ is the space containing all possible datasets. $\mathcal{S}$ can in praxis be limited to contain only those sets that can be derived from the root datasets by using transformations $t \in \mathcal{T}$ on suitable lists of features contained in that dataset. The state at each point in time is therefore a dataset that has been derived from the original (root) dataset.
The action space $\mathcal{A}$ is defined as a union of two sets, $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$. $\mathcal{A}_1$ is a set of tuples of transformations and features that can be used as inputs to these transformations. This set is equal to the set of all possible expansion-type edges in the transformation tree. Analogous $\mathcal{A}_2$ is the set of sets of features, corresponding to the reduction-type edges in the transformation tree.
The transition function is realized by, applying an action to a dataset as seen in the transformation tree. We have a special case, in which the transition function is deterministic, meaning if $\forall s'' \neq s' : t(s, a, s') = 1 \implies t(s, a, s'') = 0$.
The choice of reward function r depends on multiple factors and severely influence the performance of RL algorithms. We therefore discuss it thoroughly in a later section. γ is a hyperparameter that can be optimized during training.

Alternative approaches define the state-space $\mathcal{S}$ as the set of all possible (partial) transformation trees as opposed to the set of possible datasets. In this case, the action space can be designed as in the node-focused mode, with the addition that each transformation $t \in \mathcal{T}$ also needs to specify the dataset on which the transformation is applied. The transition is expanded similarly. A reward function can be defined for example as the score of the best dataset in the current transformation tree. The algorithm proposed in [23] uses a similar formulation.

4.3 AutoGFE Agent

We will now discuss the agent. The agent transforms dataset into vector-based representations and uses those to produce probability distributions to obtain transformations.

We will in sequence start by discussing the process of producing dataset representation. We will then introduce a component to capture local context and domain information. Lastly we cover the topic of generating a probability distribution from the dataset representation.

4.3.1 Dataset Representation

Since the agent that produces a probability distribution is an Neural Network, we need to represent datasets as fixed-sized vectors. Datasets might have any number of features and different datasets might have differing numbers of datapoints. We therefore need an approach that is both expressive and flexible. AutoGFE approaches this problem by first transforming all features in a dataset $r \in D$ into *feature representations* and then aggregating the feature representations to obtain a *dataset representation.*

We refer to the component that generates feature representation as *feature representation generator* f_{fr}, to the component that aggregates the feature representation as *feature aggregator* AGG_{dr}, and to the whole process as *dataset representation generator* $f_{\text{dr}} : \mathcal{D} \to \mathbb{R}^n$.

The feature representation generator is separated in two distinct components. Since the number of datapoints in each feature f may differ between datasets we first generate a fixed-length representation of that feature $f_{\text{rfr}}(v(f))$, which we call the *raw feature representation.* We discuss several approaches to generate raw feature representations in the next section. The raw feature representation is further processed by a Multilayer Perceptron MLP_{fr} to obtain the feature representation $f_{\text{fr}}(v(f))$. The aggregation function AGG_{dr} along with another Multilayer Perceptron MLP_{dr} is used to transform the feature representations into the dataset representation. This process is indicated in equation 4.4.

$$\begin{aligned} f_{\text{fr}}(f) &= \text{MLP}_{\text{fr}}(f_{\text{rfr}}(v(f))) \\ f_{\text{dr}}(D) &= \text{MLP}_{\text{dr}}\left(\text{AGG}_{\text{dr}}\big\{f_{\text{fr}}(v(f))|f \in D\big\}\right) \end{aligned} \tag{4.4}$$

The raw feature representation generator is a function $f_{\text{rfr}} : \mathbb{R}^+ \to \mathbb{R}^n$ where $+$ may denote any number larger than zero and $n \in \mathbb{N}$. A suitable raw feature representation should capture as much information about the data distribution underlying the features as possible. It should also yield suitable representations of fixed dimensionality independently from the number of datapoints in the feature.

In this section we discuss some functions that are suitable raw feature representation generators, along with their perceived advantages and disadvantages.

- **Sampling Feature Values**: This approach represents an original feature with l datapoints as a vector of dimensionality n by randomly sampling n elements from the feature. In this case the choice of n must be chosen such that the underlying data distribution is captured sufficiently. In extreme cases where $l \ll n$ or $n \ll l$ the resulting representation might be skewed. We obtain

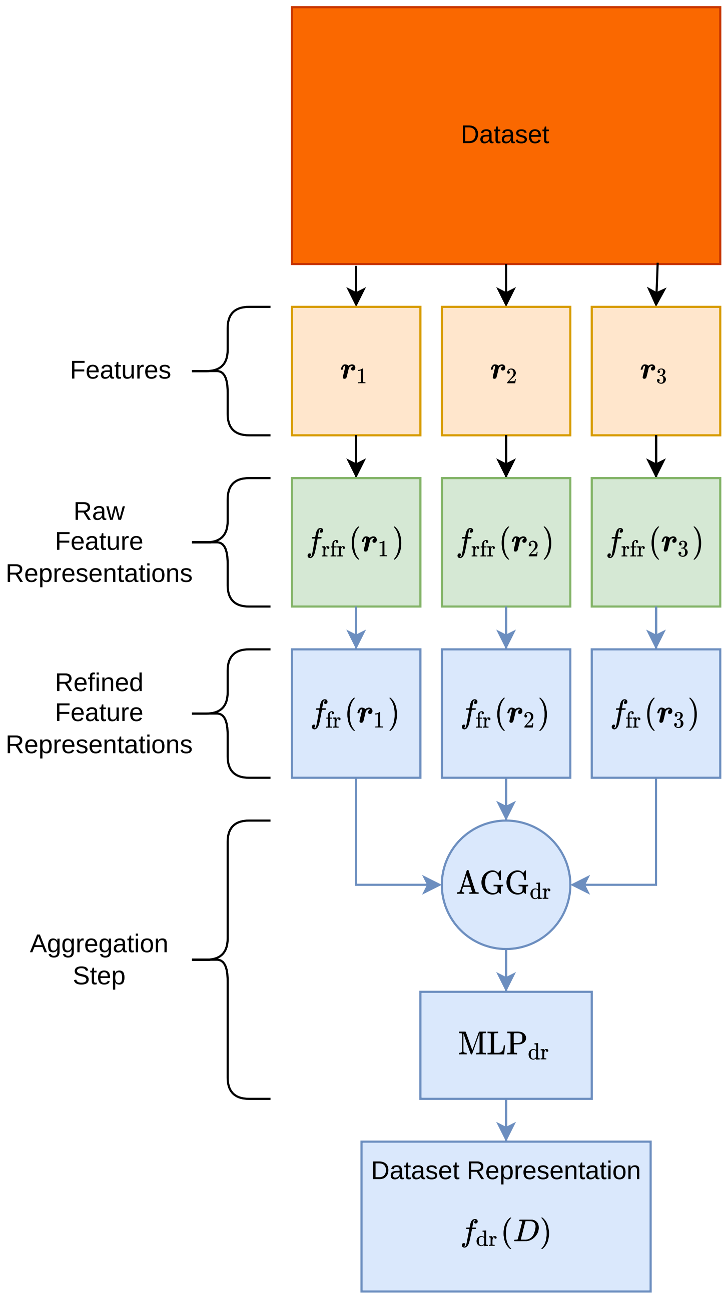
Dataset
Features
$\boldsymbol{r}_1$
$\boldsymbol{r}_2$
$\boldsymbol{r}_3$
Raw Feature Representations
$f_{\text{rfr}}(\boldsymbol{r}_1)$
$f_{\text{rfr}}(\boldsymbol{r}_2)$
$f_{\text{rfr}}(\boldsymbol{r}_3)$
Refined Feature Representations
$f_{\text{fr}}(\boldsymbol{r}_1)$
$f_{\text{fr}}(\boldsymbol{r}_2)$
$f_{\text{fr}}(\boldsymbol{r}_3)$
Aggregation Step
AGG_{dr}
MLP_{dr}
Dataset Representation
$f_{\text{dr}}(D)$

Figure 4.3: Exemplary illustration of how dataset representations are generated.

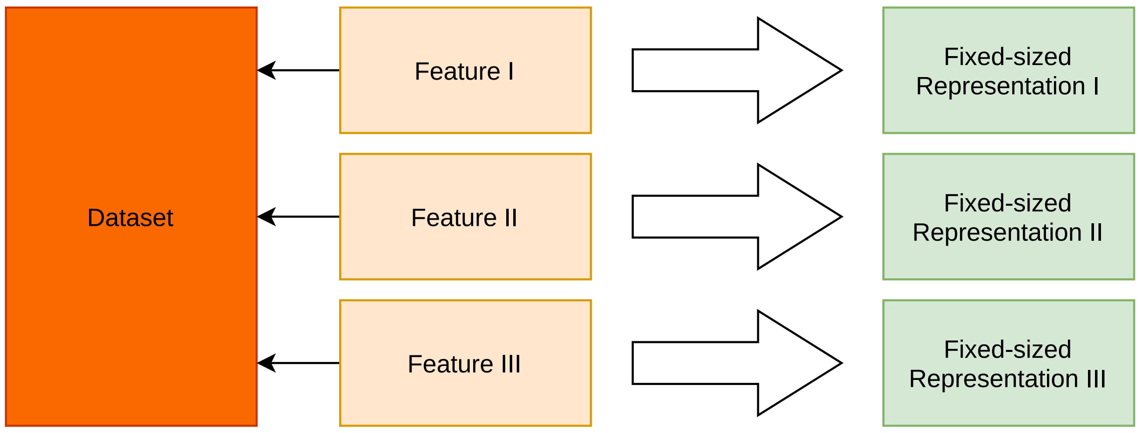
Dataset
Feature I
Feature II
Feature III
Fixed-sized Representation I
Fixed-sized Representation II
Fixed-sized Representation III

Figure 4.4: Fixed-sized representations are generated for each feature in the dataset.

consistency in the representation by sorting the randomly sampled values according to value and using equation 4.5 to normalize.

$$f_{\text{rfr}}(f) := \frac{f_{\text{rfr}}(v(f)) - \mu(v(f))}{\sigma(v(f))} \tag{4.5}$$

Here $\mu(v(f))$ is the mean of all feature values in $f_{\text{rfr}}(v(f))$ and $\sigma(v(f))$ is the standard derivation. We require normalization to balance out different scales.

- **Function Fitting**: In this case a function, such as a polynomial function, is fit to the data, for example by minimizing some distance norm. The representation is then defined as the parameters of the function. This method suffers from high computational cost compared to the other representation generators, since the function needs to be fitted on each feature while also requiring expensive computations, depending on the specific function. A special case is the fitting of a Gaussian function. Gaussian functions are able to capture a large number of natural data distributions and can be specified with only two parameters, μ and σ. We considered this idea but testing indicated insufficient performance.

- **Hand-Crafted Statistics**: Data scientists often find themselves relapsing to using hand-crafted statistics when needing good representations in a well-known domain. These statistics might perform well in certain domains, but we will refrain from using such heuristics since their usage contradicts the general objective of this thesis.

- **Quantile Data Sketch**: Naumann et al. [34] and Wang et al. [51] use quantile data sketches to represent features values. Quantile data sketches are obtained by normalizing all feature values to a predefined range and sorting them into a number of bins of equal size. The feature representation is then obtained as the distribution of the datapoints in each bin normalized over the sum of all datapoints. In this way, quantile data sketches can capture data distributions if the number of bins is chosen to be sufficiently big.

- **Autoencoders**: Autoencoders, initially proposed in [28], are a specialized type of Artificial Neural Networks which are used for representation learning. Autoencoders are trained to minimize reconstruction error on datapoints, in which datapoints are used as both input and output. Autoencoders typically contain some type of low-dimensional hidden layer which can be used to split the Autoencoder into encoder and decoder component to encode and decode datapoints sampled from an underlying data distribution. Autoencoders are used widely in many fields of science and engineering to reduce data dimensionality and learn data representations[56]. We opted not to use Autoencoder because of runtime constraints and added complexity.

After the individual features $r \in D$ are represented as fixed-sized vectors $f_{\mathrm{fr}}(r)$ we aggregate the feature representations to obtain a representation of the whole dataset $\mathrm{AGG}_{\mathrm{dr}} : \mathbb{R}^{+\times n} \to \mathbb{R}^m$. The number of features per dataset is not fixed. Some datasets have as little as one feature, while other datasets have hundreds.

We now discuss some approaches to aggregate sets of feature representation to generate dataset representations.

- **Limiting the number of retained features**: A simple way to represent dataset as fixed-sized vectors is to limit the number of features. In this case an integer $\frac{m}{n}$ is defined as the number of retained features and the vector $\mathrm{AGG}_{\mathrm{dr}}(f_{\mathrm{fr}}(f))$ is obtained by stacking the feature representations. If the initial number of features is too low, duplicates of existing features may be introduced to the dataset. During feature generation, each newly generated feature must be evaluated for usefulness and replaces a less useful feature in the retained set to keep the number of features fixed. FICUS [30] and ExploreKIT [21] utilize similar approaches. Performing a search over possible feature combinations has the disadvantage of requiring much computational power and time as the performance of each combination of features must be evaluated to determine which feature is the least useful and may be discarded. In practical implementations randomized heuristics can be used to sidestep this. We do not explore this approach because of the high computational cost.

- RNNs such as LSTM [16] are a specialized form of Recurrent Neural Networks able to process sequential data. They have been used successfully in a wide variety of tasks such as Speech Recognition [12] and the design of drugs [13]. The problem with using LSTMs or Recurrent Neural Networks in general is their sequential nature. To apply RNN to our problem, one had to impose an arbitrary order onto the features. We therefor opted for **Graph Neural Networks**, which are related circumvent the problem of permutation invariance. In this case, each feature is seen as a node in a simple graph. Using Graph Neural Networks in this setting is in many cases equivalent to performing simple aggregation operations, such as taking the sum or mean of the individual node representations.

4.3.2 Representing Domain Information

In this section we introduce a function $f_{\mathrm{domain}} : \mathbb{R}^m \to \mathbb{R}^m$ to enhance the dataset representation by capturing domain-specific knowledge. We assume that, for a given FE task, a number of datasets on the same domain is available for which FE has already been performed using AutoGFE. Therefore, transformation trees and a history of the sequences of transformations that were applied exist for each of those datasets. We use that information to improve the performance of AutoGFE.

We use a type of recursive representation to capture additional information by using Graph Neural Networks to update the dataset representation. We use the model introduced in equation 4.4 and illustrated in figure 4.3 to generate initial dataset representations for all nodes in each transformation tree. We then use GIN convolution layers to transform the dataset representations. After a number of convolutions the dataset representation of the root node is used as a representation of the whole transformation tree. Using a sufficiently large number of convolution layers allows us to capture information from a suitably large subsection of the graph. We generate a representation of the domain by taking the average of all transformation tree representations.

Let $s_{\text{domain}} = \{T_1, T_2, ...\}$ be a set of transformation trees in a common domain and $k \in \mathbb{N}$ the number of convolutions. Let $s_A = \{A_i\}$ be the set of adjacency matrices, where A_i is the adjacency matrix for transformation tree T_i. We then generate a representation $\boldsymbol{x}(s_{\text{domain}})$ as seen in algorithm 4.

Algorithm 4 Reduce Transformation Tree

```
1:  procedure REDUCE($s = \{T_1, ...\}$, $k$, $s_A = \{A_i \dots\}$)
2:      $y \leftarrow \{\}$
3:      for all $i \in \{0 \dots |s|\}$ do
4:          $D \leftarrow s_i$
5:          $A \leftarrow s_{A,i}$
6:
7:          // Generate initial representation and apply k GIN layers
8:          $\boldsymbol{x} \leftarrow f_{\text{dr}}(D)$
9:          for all $t \in \{1 \dots k\}$ do
10:             $\boldsymbol{x} \leftarrow \text{GIN}^{(t)}(\boldsymbol{x}, A)$
11:
12:         $y \leftarrow y \cup \{\boldsymbol{x}\}$
13:     return mean$\{y\}$
```

As discussed in chapter 3 we can use the choice of number of GIN convolution layers to influence how much information we include when generating domain representation. For each convolution layer additional datasets will be included in the domain representation. In that way we can control the trade-off between runtime complexity and information gain.

We add the additional information to the existing dataset representation, by stacking both vectors and reducing them using a one-layer MLP.

$$\begin{aligned} \boldsymbol{l} &= \text{reduce}(s_{\text{domain}}, k, s_{\text{a}}) \\ f_{\text{domain}}(\boldsymbol{x}) &= \text{MLP}_{\text{do}}\left(\begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{l} \end{pmatrix}\right) \end{aligned} \tag{4.6}$$

Where $\boldsymbol{l}$ is the domain-specific information, s_{domain} is the set of in-domain transformation trees, k is the number of GIN convolution layers and s_A is the set of adjacency matrices.

4.3.3 Representing Local Context

We use an approach similar to the one introduced in algorithm 4 to capture local context at each timestep to improve our decisions previous explorations in the decision tree.
At each timestep during the application of AutoGFE we can generate and randomly expand a transformation tree rooted at the currently active dataset. By using algorithm 4 to transform the dataset representation, we can capture information from this subtree to improve the performance of the transformation prediction step.
Note that for $n_{\text{exploration}} = 0$ or $n_{\text{convolutions}} = 0$ the enhanced dataset representations

Algorithm 5 Representing local context

```
1: procedure ENHANCE_LOCAL($\boldsymbol{x}$, D, $n_{\text{explorations}}$, $n_{\text{convolutions}}$)
2:     $y \leftarrow$ new transformation tree rooted at $D$
3:     for $n_{\text{explorations}}$ do
4:         // Expand the tree randomly
5:         $y$.randomly_expand_tree()
6:
7:     // Generate a representation of the local subtree
8:     $\boldsymbol{x} \leftarrow \text{reduce}(\{y\}, n_{\text{convolutions}}, \{A\})$
9:
10:    return $\boldsymbol{x}$
```

is equal to the dataset representation without local context since no convolutions are applied.
Algorithm 5 is similar to a Monte Carlo random tree search, but we do not directly use the results of the search as decision basis but instead use it as input to a heuristic.

4.3.4 Generating Probabilities

Probabilities are produced from dataset representations by a Multilayer Perceptron. We use three-layer Perceptrons with ReLu activation functions. The activation function in the last layer is a softmax function.

For the REINFORCE algorithm we use a simple three-layer Multilayer Perceptron with Rectified Linear Unit (ReLu) activation functions. After the last layer we apply the softmax function.

$$f_{\text{REINFORCE}}(f_{\text{dr}}(D)) = \text{softmax}(\text{MLP}_{\text{pr}}(f_{\text{dr}}(D))) \tag{4.7}$$

Since PPO is an Actor-Critic model the agent π_{PPO} consists of an actor and a critic component. Similar to the agent in the REINFORCE algorithm we use a three-layer Multilayer Perceptron with ReLu activation functions for the actor. For the critic we also use a three layer Multilayer Perceptron with ReLu activation functions. Since the value is a scalar, the last layer has only one node and we do not apply softmax to the result.

$$\begin{aligned} f_{\text{PPO}}^{\text{ACTOR}} &= \text{softmax}(\text{MLP}(f_{\text{dr}}(D))) \\ {}^{\text{CRITIC}}_{\text{PPO}} &= \text{MLP}(f_{\text{dr}}(D)) \end{aligned} \tag{4.8}$$

4.4 Reinforcement Learning Setting

Training is done by randomly generating trajectories and updating the policy based on the relevant loss function (REINFORCE or PPO). The procedure can be seen as pseudo-code.

Algorithm 6 AutoGFE - Training

```
procedure TRAINING($d_{\text{training}}$, $n_{\text{steps}}$, $n_{\text{episodes}}$, $n_{\text{traj}}$, $n_{\text{trajlen}}$)
    // $d_{training}$ is the set of training datasets
    // $n_{steps}$ is the number of training steps
    // $n_{traj}$ is the number of trajectories
    // $n_{trajlen}$ is the length per trajectory

    $\pi \leftarrow$ new agent
    $i \leftarrow 0$
    while not converged $\wedge$ $i < n_{\text{steps}}$ do
        // Generate batch of training data
        $b \leftarrow \emptyset$
        for $j \in \{1 \dots n_{\text{traj}}\}$ do
            // Generate a rollout of length $n_{trajlen}$ using $\pi$ on random dataset
            $D \leftarrow \text{sample}(d_{\text{training}}, 1)$
            $x \leftarrow \text{rollout}(D, \pi, n_{\text{trajlen}})$
            $b \leftarrow b \cup x$

        // Train policy $n_{episodes}$ times
        for $j \in \{1 \dots n_{\text{episodes}}\}$ do
            $\pi \leftarrow \text{train}(\pi, b)$

        $i \leftarrow i + 1$

    return $\pi$
```

4.4.1 Choice of Reward Function

The performance of most Reinforcement Learning algorithm depends heavily on the choice of reward function . Finding good reward functions is imperative to building Reinforcement Learning Models. We introduce and discuss several approaches and important properties that should be fulfilled.

We model our reward functions to be dense. In this context, a dense reward function is a function that returns a non-zero reward on every step. Its opposite, sparse reward functions, return rewards only on some steps, giving zero rewards on most

other steps. Sparse reward functions are especially useful in settings in which a clear terminal state exists. Dense reward functions also ease learning by continually providing hints to the agent during training[32] in the form of improvements or diminishments.
We require the reward function to incentivize relative gains as opposed to results. For example, if we defined the reward function as the scoring function of the newly generated node we might be able to find well-performing datasets, but once such a dataset was found, an agent with an score-based reward function might be incentivized to continue to explore this subtree even if there were no actual gains as compared to the parent node or even if the score would decrease slightly. It would not try to start searching in regions where the general score is lower but the per-step gain is larger.
All reward functions depend on the classification score as input and produce some scalar output.

A naive approach would be to use the scoring function as reward function. In this case the reward at each timestep is equal to the classification performance of the newly discovered node. Let u be the newly generated nodes, obtained at timestep t. The the score at timestep t is defined as:

$$r(u) := \text{score}(u)$$

A modified variant is the Relative Classification Score. In this case, the reward is given as the difference between the classification score of the previous dataset and the classification score of the newly found node.

$$r(u) := \text{score}(u) - \text{score}(v)$$

This is meant to incentivize the agent to find transformations which yield large gains at each individual timestep. The classification score could also be compared to the best found score among explored nodes.

$$r(u) := \text{score}(u) - \max_{v \in E}(\text{score}(v))$$

A reward function of this type was previously used in Q-learning to facilitate Feature Engineering for classification with the Q-learning algorithm[23].
Most of the previously discussed reward functions suffer from gradient problems during gradient.

For AutoGFE we use loss functions based on loss functions commonly used in robotics and supervised learning.

- **Binary Gain**: In this case the the reward at each timestep is a positive scalar if the overall performance increases and a negative scalar if it decreases.

$$r(u) := \begin{cases} 1 & \text{if score}(u) > \text{score}(v) \\ -1 & \text{otherwise} \end{cases} \tag{4.9}$$

  As such it could be seen as a normalized variation of the relative classification performance approach above. A similar reward function was used in a previous work in the context of Supervised Learning[33].

- **Staggered Gain**: Let u be the newly generated node and v be the node that was previously generated. The staggered reward function gives differing returns depending on how well the newly explored node performs.

$$r(u) := \begin{cases} 10 & \text{if score}(u) = 1 \\ 1 & \text{if score}(u) > \text{score}(v) \\ 0 & \text{otherwise} \end{cases} \tag{4.10}$$

  Similar approaches are commonly used in different Reinforcement Learning settings, such as by [2] and [19] in the field of Robotics.

During training, the rewards obtained while exploring the environment are further processed before being used for training. The rewards in a series of results are weighted with a scalar value γ to obtain weighted rewards.

$$r^{(i)}_{\text{weighted}} = r^{(i)} + \sum_{t=1}^{\infty} \gamma^t r^{(i+t)} \tag{4.11}$$

Here $r^{(t)}$ is the reward at timestep t and γ is a hyperparameter. Weighted rewards are widely used in Reinforcement Learning and improve the long time performance as compared to short time performance [47]. Furthermore discounted rewards in a batch are normalized by subtracting the mean and dividing by the standard deviation. Let $r \in D$ be a feature.

$$\boldsymbol{r}_{\text{norm}}(r) = \frac{(v(r) - \mu_{\boldsymbol{r}}(v(r)))}{(\sigma(v(r)) + 10^{-6})} \tag{4.12}$$

10^{-6} is a small scalar constant to ensure mathematical stability. μ and σ are mean value and standard deviation respectively.

4.5 Implementational Details

We implemented our algorithm in python3, using the PyTorch Deep Learning library[35] for Neural Network implementations. We use data structures and mathematical operations implemented in the numpy mathematics library[14] to represent features and transformations. We used pandas[49] for high level dataset representation, analysis and modification.
Datasets were saved in the CSV file format.

We implemented a cache system to minimize the need for duplicate evaluation of dataset performances to lower training time. Features we cached individually based on their chain of derivation. This speeds up training time but can not be used for evaluation.

5. Experiment

In this section we describe the various experiments we run. We evaluate AutoGGE in various configurations by comparing it to to the performance of the dataset before FE, to advanced classificators that implicitly perform FE and to other state-of-the-art algorithms such as LFE [33].

5.1 Datasets

We used datasets from the UCI Machine Learning Repository [7], the *LIBSVM* repository and from the Data Science website *kaggle.com*. Some datasets were imported from *sklearn*[36]. In table 5.1 we added a sample of datasets we used. Due to size constraints we will not include the full list of datasets, but it can be found with source links on the GitHub belonging to this paper.

We did some preprocessing for the datasets. Very large datasets with more than 10000 datapoints were randomly sampled to yield datasets with exactly 10000 datapoints to increase runtime performance. Categorical features where transformed into one-hot encodings.

We trained using about 60 datasets. About 15 datasets we used for validation. During training the set of all non-validation datasets was randomly split into training, testing and validation sets in a 0.8 to 0.2 ratio. The set of validation datasets was kept separate for later evaluation and was not used for training nor for optimization of the algorithm. The training dataset was used for hyperparameter tuning.

Datasets were scored using a Decision Tree model implemented in sklearn [36] with its default parameters. We report mean improvement off R-score over multiple independent runs of our algorithm compared to the respective performance of the base dataset for each set of parameters.

Dataset	Source	#datapoints	#features	Citation
california housing	sklearn	20640	8	[36]
cancer	sklearn	596	30	[36]
diabetes	sklearn	442	10	[36]
digits	sklearn	1797	64	[36]
linnerud	sklearn	20	3	[36]
wine	sklearn	178	13	[36]
abalone	UCI	4177	8	[7]
accelerometer	UCI	153000	5	[40]
balance Scale	UCI	625	4	[7]
balloons	UCI	16	4	[7]
blood transfusion	UCI	748	5	[7]
credit approval	UCI	690	15	[7]
ecoli	UCI	336	8	[7]
fertility	UCI	100	10	[7]
iris	UCI	150	4	[7]
ionosphere	UCI	351	34	[7]
lymphography	UCI	148	18	-
madelon	UCI	4400	500	[7]
soybean (large)	UCI	307	35	[7]
SPECT heart	UCI	267	22	[7]
GES Classification	Zenodo	-	3	[50]

Table 5.1: A sample of datasets we used along with meta-information. All dataset are classification tasks. All features in all datasets are numerical. Target features are not included in the number of columns. Citation is of the source website of no more specific citation was given.

Transformation	Arity	Datatypes
square root	1	NUMERICAL
logarithm	1	NUMERICAL
absolute	1	NUMERICAL
round	1	NUMERICAL
sigmoid	1	NUMERICAL
tanh	1	NUMERICAL
normalization	1	NUMERICAL
multiplication	2	NUMERICAL, NUMERICAL
division	2	NUMERICAL, NUMERICAL
addition	2	NUMERICAL, NUMERICAL
subtraction	2	NUMERICAL, NUMERICAL

Table 5.2: List of Feature Transformations used in experiments.

5.2 Transformations

We used transformations that were common in most of the related literature.
Transformations, such as *log, square, square root, multiplications, normalization, addition, subtraction, division, sigmoid* and *tangens hyperbolicus* were used in both [33] and [23]. Most of these transformations were also used in [15]. We opted to not use more complex transformations, such as binning or nominal expansion (used in [23]). We chose not to use such transformations because they are more computationally expensive and might add bias towards these more complex transformations. We used only unary and binary transformations to circumnavigate problems in complexity and because most transformations with high arity can be decomposed into unary or binary transformations (e.g. $a + b + c \implies ((a + b) + c)$).
A comprehensive list of used transformations can be found in table 5.2. All transformations were implemented as transformation on numpy ndarrays.

5.3 Hyperparameters

In the experiments we evaluate the performance depending on the following parameters: Learning algorithm (REINFORCE or PPO), reward function (binary gain or staggered gain) and raw feature representation generator (random sampling or quantile data sketch). For experiments B and C we additionally used parameters to add domain information and local context respectively (enabled or disabled).
Before the experiment we fixed the dimensionality of the raw feature representations to 64, the number of trajectories per batch during training to 8 and the maximum length of a trajectory to 16. We trained for 500 episodes and updated the policy thrice during each episode. We used a discount factor of $\gamma = 0.4$ for discounted rewards.
GIN layers, when needed, were used with $\epsilon^{(k)} = 0.1$ and a one-layer MLP with the same input and output dimensions was used to transform the aggregated transformation.
All MLP components used the ReLu activation function. The number of neurons per layer for the MLP_{fr} Multilayer Perceptron was set to 64 neurons for the first

layer and 32 layers for the second layer. For $\mathrm{MLP}_{\mathrm{dr}}$ we set the first layer to 32 and the second layer to 16. The third (output) layer was set to the number of transformations plus 1, the additional dimension representing removal of features. For PPO we use equation 3.22 with a clipping value of $\epsilon = 0.2$ as loss function and an architecture for the critic with an 32-dimensional, an 16-dimension and a 1-dimensional layer. The last layer in the critic did not use an activation function.
Back-propagation was performed using the ADAM optimizer [25] with learning rate $\alpha = 0.5 * 10^{-3}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

To limit the growth of datasets, we limited the number of features that could be generated during each timesteps. If for a given dataset and transformation more than 8 new features could be generated we randomly sampled 8 new features from among them to be added.

All experiments were performed on a mid-range computer with a AMD Ryzen 7 3700X processor. GPU Acceleration was not used. This illustrates the ability of AutoGFE to perform FE without the need for specialized software and hardware. The average training duration was eight to ten hours depending on the choice of parameters.

5.4 Experiment A: Baseline

Experiment A establishes baselines for our algorithm. We use the general architecture introduced in chapter 4 without using enhancements (local context aggregation and domain information) to establish grounds for comparison of the effect of the two advanced components.
Each datapoint in the evaluation was obtained by running AutoGFE five times for ten steps on a given dataset and retaining the best results. This was repeated 5 times for each set of parameters over the validation dataset for each set of parameters. The mean increase in F-score along with standard deviation is reported.
We investigate different combinations of feature representation functions f_{rfr} and reward functions. A stochastic exploration scheme was employed to select transformations by sampling a probability distribution generated by the agent. We evaluate using Decision Tree classifiers.
We also compare our algorithm with the performance of the Random Forest classifier as an example of a classifier that implicitly performs FE.

5.5 Experiment B: Local Context and Domain-specific information

Experiment B war designed to investigate the impact of the inclusion of local context and domain-specific information on the model. We compare different combinations of these additional components and report on the results.
We selected the hyperparameters for the four best performing models from experiment A and trained each parameter set with the additional components. We report

Dataset	Citation	Improvement Cognito	Improvement ReCog
Australian Credit	[7]	7%	5.490±2.205%
svmguide1	[18]	2%	2.846±3.077%
svmguide3	[18]	29%	7.528±2.102%
Ionosphere	[7]	8%	6.431±1.633%
Pima Diabetes	[7]	11%	9.4±2.2%
German Credit	[7]	9%	8.2±1.2%

Table 5.3: Comparison of the results of our reproduction ReCog to the ones generated by Cognito as reported in [24]. Khurana et al. only published results for eight dataset. From those one was a regression dataset and one dataset was proprietary. Citation is of the source website of no more specific citation was found. Improvement of ReCog is mean and standard deviation over 10 runs.

mean results and standard variation over 5 best-of-five runs.

5.6 Experiment C: Full Model

Experiment C was designed to establish the performance of our model. We selected the best set of parameters from experiment B.
We report performance on various datasets.
We also compare our model to other procedures such as an algorithm based on [24]. Khurana et al. did not publish a full specification of their algorithm. We therefore use the search algorithms based on [24] in our implementation of the transformation tree with our set of transformations. To differentiate our reproduction from the original algorithm we will denote the reproduction as ReCog (*Re*production *Cog*nito) and the original model as Cognito. We used the depth-first traversal mode, generating one new feature per step. After each step a random feature is discarded. We optimized hyperparameters to produce good results. The resulting algorithm is less sophisticated and performs slightly worse than Cognito on the datasets evaluated in [24]. A comparison of the performance of Cognito and ReCog can be seen in table 5.3.
Our implementation of the ReCog model can be found in the *GitHub* repository belonging to this thesis.

We also compare performance of our model with results of other state-of-the-art FE algorithms reported in [22] (AutoLearn) and [33] (LFE).

6. Evaluation

6.1 Experiment A

In experiment A we tested the different hyperparamters for AutoGFE. There are no clear outliers in terms of performance, but there are some subtle differences.

REINFORCE performs better in most configurations when directly compared to PPO. The reward function does not seem to influence the performance.
Concerning raw feature representation functions f_{rfr}, quantile sketch array performs better than random sampling in most cases, but also offers the single worst performance among all tested datasets.

Model			Results
Reward	Representation	L.A.	Mean Imp.
binary_reward	random_sampling	REINFORCE	0.027±0.015
binary_reward	random_sampling	PPO	0.023±0.019
staggered_reward	random_sampling	REINFORCE	0.020±0.022
staggered_reward	random_sampling	PPO	0.027±0.014
binary_reward	quantile_data_sketch	REINFORCE	0.038±0.027
binary_reward	quantile_data_sketch	PPO	0.024±0.029
staggered_reward	quantile_data_sketch	REINFORCE	0.033±0.016
staggered_reward	quantile_data_sketch	PPO	0.032±0.022

Table 6.1: Results with different sets of hyperparameters and a stochastic evaluation scheme. Columns from left to right: RL reward function, raw representation generator f_{rfr}, learning algorithm (REINFORCE or PPO) and mean improvement in F-score on validation dataset with standard derivation.

6.2 Experiment B

In table 6.2 we can see the performance of our algorithm using the advanced components.
We notice that the performance decreases when local context aggregation is enabled, both when it is enabled without the domain-specific context and with it. The best performance is reached when only the domain-specific context component is enabled.
Standard deviation increases when the domain-specific context component is enabled.

Model		
Local context	Domain-specific context	Mean Imp.
disabled	disabled	0.032±0.018
enabled	disabled	0.024±0.012
disabled	enabled	0.035±0.021
enabled	enabled	0.032±0.025

Table 6.2: AutoGFE with components enabled. The mean improvement is the mean difference in F-score between the base dataset and the result of AutoGFE over the validation set.

6.3 Experiment C

In table 6.3 the performance of our algorithm along with the performance of classification algorithms that implicitly perform FE is shown for a sample of datasets. Note that AutoGFE performs comparably to the Random Forest Classifier.
The comparison of our algorithm with other state-of-the-art algorithms can be found in table 6.4.

Dataset	Baseline		AutoGFE
	Without FE	Random Forest	Performance
ecoli	0.785	0.872	0.832±0.003
ionosphere	0.866	0.937	0.895±0.005
iris	0.953	0.962	0.972±0.004
skl_boston	1.000	1.000	1.000±0.002
skl_digits	0.930	0.947	0.951±0.001

Table 6.3: Comparison of AutoGFE with complex classification algorithms. The last column holds the relative gain in performance of AutoGFE compared to the dataset before FE.

We can see that all four FE algorithms improve the overall F score of the dataset. For AutoGFE, we see scores that are comparable with the other models. and on many datasets we show better performance than other models.

Dataset	Baseline				AutoGFE	
	Without FE	ReCog	LFE	FEPM	Performance	Var.
default_credit	0.715	-	-	0.831	0.726	±0.004%
fertility	0.789	0.891	0.861	-	0.866	±0.000%
ionosphere	0.931	0.939	0.925	0.941	0.899	±0.005%
lymphography	0.781	0.768	0.719	-	0.799	±0.004%
pima_diabetes	0.697	0.730	0.760	0.756	0.750	±0.002%
spambase	0.893	-	0.947	0.961	0.899	±0.001%
spect_heart	0.850	0.920	0.956	0.788	0.888	±0.007%
svmguide3	0.735	0.795	-	0.776	0.758	±0.006%

Table 6.4: Comparison of AutoGFE with other state-of-the art algorithms as reported in [23][33]. All scores are evaluated using the Decision Tree Classifier. The last column holds the variance in performance. Dataset names may differ between publications.

Overall we see the capability of AutoGFE to positively impact the F score achieved with simple classification algorithms.

7. Conclusion and Future Work

With this thesis we presented a new approach to apply the techniques of Reinforcement Learning to the field of Feature Engineering. We used techniques of Reinforcement Learning and Graph Neural Networks with the transformation tree datastructure to enable flexible training with minimal requirements. Our model was able to successfully generate transformed datasets that have a measurable impact on performance in the task of classification. AutoGFE is highly composable and can be extended depending on the application environment. We also showed a way to leverage previously unused information to improve performance of FE. Our experiments show comparable performance on many datasets compared to other state-of-the-arts algorithms.

There are some unexplored aspects of our work that might be interesting to be investigated. First of all, our usage of a policy based on Multilayer Perceptrons is very basic. It might prove interesting to use more complex Neural Network Architectures. An idea that comes to mind is to use Attention Modules to learn specific queries for different transformations. We also used only older Reinforcement Learning algorithms, such as REINFORCE and PPO. More modern algorithms might boost performance both in training and in production. As indicated in the paper it might be interesting to apply the different forms of Autoencoders both to generate feature representations as well as to encode domain information.

Bibliography

[1] Peter W. Battaglia et al. “Interaction Networks for Learning about Objects, Relations and Physics”. In: (2016). arXiv: 1612.00222 `[cs.AI]`.

[2] Greg Brockman et al. *OpenAI Gym.* 2016. eprint: arXiv:1606.01540.

[3] Kyunghyun Cho et al. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078.

[4] George V. Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of Control, Signals and Systems* 2 (1989), pp. 303–314.

[5] Ofer Dor and Yoram Reich. “Strengthening learning algorithms by feature discovery”. In: *Information Sciences* 189 (2012), pp. 176–190.

[6] B. L. Douglas. *The Weisfeiler-Lehman Method and Graph Isomorphism Testing.* 2011. arXiv: 1101.5211 `[math.CO]`.

[7] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository.* 2017.

[8] Wolfgang Ertel and Nathanael T Black. *Grundkurs Künstliche Intelligenz.* Springer, 2016.

[9] Wei Fan et al. “Generalized and heuristic-free feature construction for improved accuracy”. In: *Proceedings of the 2010 SIAM International Conference on Data Mining.* SIAM. 2010, pp. 629–640.

[10] Vincent François-Lavet et al. “An introduction to deep reinforcement learning”. In: *arXiv preprint arXiv:1811.12560* (2018).

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

[12] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. “Speech recognition with deep recurrent neural networks”. In: *2013 IEEE international conference on acoustics, speech and signal processing.* Ieee. 2013, pp. 6645–6649.

[13] Anvita Gupta et al. “Generative recurrent networks for de novo drug design”. In: *Molecular informatics* 37.1-2 (2018), p. 1700111.

[14] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362.

[15] Jeff Heaton. “An empirical analysis of feature engineering for predictive modeling”. In: *SoutheastCon 2016.* IEEE. 2016, pp. 1–6.

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

[17] Ronald A Howard. “Dynamic programming and markov processes.” In: (1960).

[18] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. *A practical guide to support vector classification*. 2003.

[19] Shirin Joshi, Sulabh Kumra, and Ferat Sahin. “Robotic grasping using deep reinforcement learning”. In: *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*. IEEE. 2020, pp. 1461–1466.

[20] James Max Kanter and Kalyan Veeramachaneni. “Deep feature synthesis: Towards automating data science endeavors”. In: *2015 IEEE international conference on data science and advanced analytics (DSAA)*. IEEE. 2015, pp. 1–10.

[21] Gilad Katz, Eui Chul Richard Shin, and Dawn Song. “Explorekit: Automatic feature generation and selection”. In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE. 2016, pp. 979–984.

[22] Ambika Kaul, Saket Maheshwary, and Vikram Pudi. “Autolearn—Automated feature generation and selection”. In: *2017 IEEE International Conference on data mining (ICDM)*. IEEE. 2017, pp. 217–226.

[23] Udayan Khurana, Horst Samulowitz, and Deepak Turaga. “Feature engineering for predictive modeling using reinforcement learning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[24] Udayan Khurana et al. “Cognito: Automated Feature Engineering for Supervised Learning”. In: *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*. 2016, pp. 1304–1307.

[25] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 `[cs.LG]`.

[26] Sanjay Krishnan and Eugene Wu. *AlphaClean: Automatic Generation of Data Cleaning Pipelines*. 2019. arXiv: 1904.11827 `[cs.DB]`.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[28] Yann Lecun. “PhD thesis: Modeles connexionnistes de l’apprentissage (connectionist learning models)”. In: (1987).

[29] Zhou Lu et al. “The Expressive Power of Neural Networks: A View from the Width”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017.

[30] Shaul Markovitch and Dan Rosenstein. “Feature generation using general constructor functions”. In: *Machine Learning* 49.1 (2002), pp. 59–98.

[31] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 `[cs.LG]`.

[32] Abdalkarim Mohtasib, Gerhard Neumann, and Heriberto Cuayáhuitl. “A Study on Dense and Sparse (Visual) Rewards in Robot Policy Learning”. In: *CoRR* abs/2108.03222 (2021). arXiv: 2108.03222.

[33] Fatemeh Nargesian et al. “Learning Feature Engineering for Classification.” In: *Ijcai*. 2017, pp. 2529–2535.

[34] Felix Naumann et al. “Attribute classification using feature analysis”. In: *icde*. Vol. 271. 2002.

[35] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[36] Fabian Pedregosa et al. “Scikit-learn: Machine learning in Python”. In: *Journal of machine learning research* 12.Oct (2011), pp. 2825–2830.

[37] Antonin Raffin et al. “Stable-Baselines3: Reliable Reinforcement Learning Implementations”. In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8.

[38] Martin Riedmiller. “Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method”. In: *European conference on machine learning*. Springer. 2005, pp. 317–328.

[39] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2017. arXiv: 1609.04747 `[cs.LG]`.

[40] Gustavo Scalabrini Sampaio et al. “Prediction of Motor Failure Time Using An Artificial Neural Network”. In: *Sensors* 19.19 (Oct. 2019), p. 4342.

[41] Benjamin Sanchez-Lengeling et al. “A Gentle Introduction to Graph Neural Networks”. In: *Distill* (2021). https://distill.pub/2021/gnn-intro.

[42] Franco Scarselli et al. “The graph neural network model”. In: *IEEE transactions on neural networks* 20.1 (2008), pp. 61–80.

[43] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 `[cs.LG]`.

[44] John Schulman et al. “Trust Region Policy Optimization”. In: *ArXiv* abs/1502.05477 (2015).

[45] Asad Ali Shahid et al. “Learning continuous control actions for robotic grasping with reinforcement learning”. In: *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE. 2020, pp. 4066–4072.

[46] Hava T. Siegelmann and Eduardo Sontag. “Turing computability with neural nets”. In: *Applied Mathematics Letters* 4 (1991), pp. 77–80.

[47] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[48] Richard S Sutton et al. “Policy gradient methods for reinforcement learning with function approximation”. In: *Advances in neural information processing systems*. 2000, pp. 1057–1063.

[49] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020.

[50] Fabián Villena and Jocelyn Dunstan. *GES classification dataset*. 2021.

[51] Lu Wang et al. “Quantiles over data streams: An experimental study”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 2013, pp. 737–748.

[52] Christopher John Cornish Hellaby Watkins. “Learning from delayed rewards”. In: (1989).

[53] Ronald J. Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine Learning*. 1992, pp. 229–256.

[54] Zonghan Wu et al. “A comprehensive survey on graph neural networks”. In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), pp. 4–24.

[55] Keyulu Xu et al. “How Powerful are Graph Neural Networks?” In: *ArXiv* abs/1810.00826 (2019).

[56] Junhai Zhai et al. “Autoencoder and Its Various Variants”. In: *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2018, pp. 415–419.