Pontifícia Universidade Católica
do Rio de Janeiro

Gabriel Baruque

Classification with Missing and Costly Features

Tese de Doutorado

Thesis presented to the Programa de Pós–graduação em Engenharia Elétrica, do Departamento de Engenharia Elétrica da PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Engenharia Elétrica.

Advisor: Prof. Wouter Caarls

Rio de Janeiro
September 2024

Pontifícia Universidade Católica
do Rio de Janeiro

Gabriel Baruque

Classification with Missing and Costly Features

Thesis presented to the Programa de Pós–graduação em Engenharia Elétrica da PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Engenharia Elétrica. Approved by the Examination Committee:

Prof. Wouter Caarls
Advisor
Departamento de Engenharia Elétrica – PUC-Rio

Prof. Ronaldo Ribeiro Goldschmidt
IME

Prof. Leonardo Alfredo Forero Mendoza
UERJ

Prof. Raul Queiroz Feitosa
Departamento de Engenharia Elétrica – PUC-Rio

Prof. Marley Maria Bernardes Rebuzzi Vellasco
Departamento de Engenharia Elétrica – PUC-Rio

Rio de Janeiro, September the 24th, 2024

Gabriel Baruque

Bachelor in Electronic Engineering and Master in Electrical Engineering from Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (CEFET-RJ).

Bibliographic data

Baruque, Gabriel

Classification with Missing and Costly Features / Gabriel Baruque; advisor: Wouter Caarls. – 2024.

114 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Engenharia Elétrica, 2024.

Inclui bibliografia

1. Engenharia Elétrica – Teses. 2. Informações Custosas. 3. Aprendizado por Reforço. 4. Transformers. 5. Classificação. 6. Valores Faltantes. I. Caarls, Wouter. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Engenharia Elétrica. III. Título.

CDD: 621.3

To my wife, parents, and brother, for their support
and encouragement.

Acknowledgments

First of all, I would like to thank God for the opportunity to follow this path, which has been filled with challenges and opportunities for growth.

I cannot express how grateful I am to my advisor, Wouter Caarls, whose patience and guidance have supported me during times of uncertainty and lack of confidence.

I also want to thank my bachelor's advisor, Luciana Faletti Almeida, and my master's advisor, Rodrigo Tosta Peres, both of whom have influenced my appreciation for this field of knowledge in the best possible ways. People with great hearts truly make a difference in the world.

My family has always been a source of encouragement, and for that, I am grateful to all of them.

Lastly, I thank my wife Nathália de Souza Nascimento Baruque, who supported me through both the good and the challenging moments.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and the National Council for Scientific and Technological Development - CNPq, under projec number 140059/2020-2.

Abstract

Baruque, Gabriel; Caarls, Wouter (Advisor). **Classification with Missing and Costly Features**. Rio de Janeiro, 2024. 114p. Tese de Doutorado – Departamento de Engenharia Elétrica, Pontifícia Universidade Católica do Rio de Janeiro.

In the field of Machine Learning, classification problems remain among the most relevant issues as they are present in a wide variety of scenarios and environments, such as in industry, retail companies, and healthcare. Virtually every company needs to solve a classification problem at some point in their solution or service, whether it be a simple day-to-day issue or a data-driven problem with significant impact.

Different types of problems arise from the concept of classification. One such problem is classification with a budget, or classification with costly features. This problem is characterized by the cost required to collect information for classification, where each piece of information (feature) has an associated cost, whether related to time, money, or any scarce resource.

To solve classification problems in general, Machine Learning methods such as artificial neural networks, decision trees, Bayesian-based methods, deep learning, and others have seen a significant increase in use in recent years due to their high performance in predictions for most use cases. The specific case of Classification with Costly Features has not been the target of extensive research, and thus, few methods have been developed to overcome this problem.

One possible way to handle the Classification with Costly Features problems is by modeling it as a sequential decision-making problem, and applying Reinforcement Learning, as done in some works. However, research that approaches this problem with Reinforcement Learning usually does not train the model in a problem-oriented way, or apply different models for different objectives in this context.

In order to be suitable for more complex problems, Deep Learning techniques were incorporated into Reinforcement Learning methods, what is called Deep Reinforcement Learning.

The objective of this thesis is to develop and enhance Deep Reinforcement Learning methods in problems of Classification with Costly Features, in a flexible way so that the model can be used on different datasets with little or no modification to its parameters, and with problem-oriented and efficient training, leveraging already known information.

To achieve this goal, two Deep Reinforcement Learning methods were developed to classify six different datasets. In the course of the research, an additional classification method for samples with missing values was developed

as a proof of concept. Reference methods were used for comparison with the proposed ones.

The results achieved demonstrate that the proposed methods for Classification with Costly Features have better or comparable outcomes to the reference methods. The method for classification with missing values, in general, outperformed the reference methods.

Keywords

Costly Features; Reinforcement Learning; Transformers; Classification; Missing Values.

Resumo

Baruque, Gabriel; Caarls, Wouter. **Classificação com Características Faltantes e Custosas**. Rio de Janeiro, 2024. 114p. Tese de Doutorado – Departamento de Engenharia Elétrica, Pontifícia Universidade Católica do Rio de Janeiro.

Na área de Aprendizado de Máquina, problemas de classificação ainda permanecem como um dos mais relevantes problemas, uma vez que estão presentes em uma grande variedade de cenários e ambientes, como na indústria, empresas de varejo, e na área da saúde. Virtualmente, toda empresa necessita resolver um problema de classificação em algum ponto de sua solução ou serviço, seja um problema simples do dia-a-dia ou um baseado em dados com grande impacto.

Em termos de classificação, surgem diferentes tipos de problema advindos desse conceito. Um deles é a classificação com orçamento, ou classificação com informações custosas. Esse problema é caracterizado pelo custo necessário para coletar informações para a classificação, onde cada pedaço de informação (atributo) possui um custo atrelado, seja relacionado a tempo, dinheiro, ou qualquer recurso escasso.

Para solucionar problemas de classificação em geral, métodos de Aprendizado de Máquina como redes neurais artificiais, árvores de decisão, métodos baseados em Bayes, aprendizado profundo, e outros, têm visto um grande aumento em sua utilização nos últimos anos, devido ao seu alto desempenho em predições na maioria dos casos de uso. O caso específico da Classificação com Informações Custosas não tem sido alvo de inúmeras pesquisas, e por isso, poucos métodos foram desenvolvidos para superar esse problema.

Uma possível forma de lidar com problemas de Classificação com Informações Custosas é modelá-los como um problema de tomada de decisão sequencial e aplicar Aprendizado por Reforço, como feito em algumas pesquisas. No entanto, a pesquisa que aborda esse problema com Aprendizado por Reforço geralmente não treina o modelo de forma orientada ao problema, e aplica modelos diferentes para diferentes objetivos dentro desse contexto.

De modo a ser utilizado em problemas mais complexos, técnicas de Aprendizado Profundo foram incorporadas em métodos de Aprendizado por Reforço, o que é chamado de Aprendizado por Reforço Profundo (Deep Reinforcement Learning – DRL).

O objetivo desta tese é desenvolver e aprimorar métodos de Aprendizado por Reforço Profundo em problemas de Classificação com Características Custosas, de forma flexível, para que o modelo possa ser utilizado em diferentes conjuntos de dados com pouca ou nenhuma modificação em seus parâmetros,

e com um treinamento eficiente e orientado ao problema, aproveitando informações já conhecidas.

Para alcançar tal objetivo, dois métodos de DRL foram desenvolvidos para classificar seis conjuntos de dados diferentes. No decorrer da pesquisa, mais um método de classificação para amostras com valores faltantes foi desenvolvido como prova de conceito. Métodos de referência foram utilizados para comparação com os propostos.

Resultados alcançados demonstram que os métodos propostos para CwCF possuem resultados melhores ou comparáveis aos métodos de referência. O método de classificação com valores faltantes se mostrou, em geral, superior aos métodos de referência.

Palavras-chave

Informações Custosas; Aprendizado por Reforço; Transformers; Classificação; Valores Faltantes.

Table of contents

List of figures

List of tables

List of Abreviations

RL – Reinforcement Learning

DRL – Deep Reinforcement Learning

CwCF – Classification with Costly Features

ANN – Artificial Neural Network

FFNN – Feed-Forward Neural Network

SotA – State-of-the-Art

NLP – Natural Language Processing

My epigraph

Epigraph Author, *Epigraph book.*

1
Introduction

In our daily tasks, we often need to give something a label. For instance, a physician needs to label the symptoms of a patient to provide a diagnosis; a marketing employee may be tasked to divide a company's customers into a few groups based on how much they are willing to spend; a logistic team in a delivery company needs to label the products in three groups, based on their probability of getting lost during transportation. Those examples are classification tasks.

Predicting the label based on the subject's available information is important: one could be prepared for the outcome beforehand. However, this is not always an easy task as the patterns that map from information to the desired outcome may be hard for humans to see. This process can be automated with Machine Learning (ML) classification models, as information is increasingly available in the form of datasets. These models aim to classify new data based on previously available information, learning patterns that may be used to predict the labels.

In the real world, gathering features, or in other words, information, is mandatory to predict the label about specific data. With no information about a subject, it is infeasible to believe that any better-than-chance prediction is achievable. Gathering those features often requires a cost in terms of scarce resources, which can be a quantifiable one, such as time and money, or resources that are harder to quantify, such as discomfort, e.g. in patient examinations.

1.1
Classification with Costly Features

ML models are usually designed to receive all available features at once without considering the acquisition cost for the information received. However, creating models that do account for real-world limitations has been increasingly important for the ML community. This increasing importance is the consequence of ML models being used more often in real-life applications, where limited resources, e.g. time and money, are crucial for service provider companies, healthcare providers, etc. Specific scenarios can be, for instance, automated medical diagnosis [1, 2], and text classification with a budget in

terms of the number of words used at test-time [3], imposing a cost-sensitive scenario in both cases.

This problem can be found in the literature under different names: Budgeted ML, Resource Constraint Learning, and Test-time Cost-sensitive Learning [93, 94]. Classification with Costly Features (CwCF) is the name given to a specific case of that scenario, where each feature of a dataset has its respective cost to be "measured" (used) [13, 14]. If a model uses all available features from a dataset, the final cost is the sum of all individual feature costs. If, after some feature selection phase, the same model uses only half of the features, the cost will be the sum of the respective used features only.

Studies addressing the CwCF problem have been present in the literature since 2002 [84] and 2007 [85]. While these approaches ultimately achieve the goal of executing a sequence of decisions that includes feature selection and data classification, they do have limitations. Specifically, [84] employs a Q-learning-based approach alongside Bayesian decision-making problems. This method uses three different networks for various objectives, resulting in a complex model where the final outcome is contingent upon the intermediate results from each network, allowing for error propagation. In contrast, [85] interprets the problem as a Partially Observable Markov Decision Process (POMDP), necessitating the training of a Hidden Markov Model to overcome challenges posed by this modeling, such as state discretization and the inability to avoid repeated actions. In contrast, the models proposed in this thesis do not require an auxiliary model and, depending on the problem formulation, are capable of avoiding repeated actions

It is important to note that creating efficient models that do account for test-time constraints regarding limited resources narrows down the distance between ML and real-world applications, ranging from requiring less expensive hardware, time, or information [5, 6, 93, 94]. These are examples of applications that can fit many different problems, in different fields of real-world scenarios. To solve this kind of problem, ML algorithms must not only be trained to achieve the best possible performance for the task (e.g. classification) but also be trained to consume/require the least amount of a limited resource as possible [5, 14, 46, 93, 94].

Currently, some algorithms that address the problem of test-time budget constraints can use structured prediction [4], decision trees or random forests [5, 6, 7] and Reinforcement Learning [8, 9, 10, 11, 12], recently achieving State-of-the-Art (SOTA) performance in [13, 14].

Existing methods designed for CwCF problems [84, 85, 13, 14] usually employ standard strategies for action selection and target updates, overlooking

the specific characteristics of the problem at hand. In this regard, the models proposed in this thesis aim for efficient training concerning both action exploration and target updates, while also implementing promising models [36] with a natural representation of the problem within a Reinforcement Learning (RL) framework.

In light of the scenario described, the primary contribution of this thesis is to develop and train novel state-of-the-art models for CwCF problems, capable of dynamically selecting features or classifying with gathered information. In addition, the secondary objective is to construct three distinct models: the first leverages Deep Reinforcement Learning (DRL) specifically tailored for CwCF challenges, the second employs Transformers to provide a natural representation of the same problem, and the third serves as an intermediary step, utilizing a Transformer-based model for classification tasks involving missing features. The research involved data collection and preprocessing, rigorous training protocols, and thorough evaluation using relevant performance metrics. The proposed models were benchmarked against existing methods to demonstrate their performance.

1.2 Motivation

The motivation for this research started in the area of healthcare, aiming to decrease costs and provide a personalized policy of examinations. However, this setup can be applied not only in healthcare, but also in many different scenarios, as long as the main objective is classifying with a lower cost of some scarce resource. For instance, still in the healthcare example, with more examinations a physician has more information to provide a diagnosis, however it incurs in more expenses, which ultimately are more likely to fall on the patient, leading to increased healthcare costs and potential financial burden for those seeking medical care.

In this paradigm, the examinations provide the features, and it is important to weigh the cost in terms of monetary value against the final well-being of the patient in terms of providing a correct classification for treatment. This setup can be applied not only for monetary cost, but many other factors, for instance, examination risks, patient discomfort, time spent, and more.

1.3 Contributions

This thesis aims to achieve better performance on test-time constrained problems, specifically in classification with costly features (CwCF), using Deep

Reinforcement Learning (DRL) applications. This research contributes with two novel DRL approaches for CwCF problems. One approach is achieved using a supervised learning paradigm concurrent with the training of a Deep Q Network (DQN) agent. The other leverages the power of Transformers to train a DQN agent for CwCF. As an additional contribution, a transformer was also trained to classify instances with missing values using only supervised learning, as a way of validating its capability to deal with incomplete information.

The first novel approach brings attention to the possibility of joining together two usually separate ML paradigms, Reinforcement Learning and Supervised Learning. By doing so, a sequential decision problem (e.g. choosing the best features dynamically), can be solved along with a supervised problem (e.g. classification) while the constrained resource is being considered during training. This approach allows for efficient training, allowing the usage of information known *a priori*, and implementing an efficient training policy.

The second novel approach tackles the same problem, but this time we use a transformer model instead of feed-forward neural networks. Transformers are particularly suited for handling irregularly-sized inputs such as samples with missing data. In this approach, the power and flexibility of transformers are used to solve the CwCF problem, handling the sequential decision-making, usually designed as a Markov Decision Process (MDP), as a fixed sequence per sample. To the best of our knowledge, this is the first time a transformer model has been used to handle a CwCF problem.

Each proposed model is compared agains a set of baselines, comprising well established models. For the first model, a two-step classifier baseline was developed, where the first step is a feature selection process and the second step is a Neural Network classifier. For the intermediate model, the transformer classifier is compared against XGBoost [15], and a combination of Multiple Imputation by Chained Equations (MICE) [16, 17] and a classifier. Lastly, for the transformer model, the same baseline used with the first model is considered.

1.4 Organization

This thesis is organized as follows: Chapter 2 presents a literature review of Reinforcement Learning (RL) and Deep Learning (DL) methods used throughout the research. In Chapter 3, the RL model for CwCF is proposed. Chapter 4 focuses on the development of the transformer model for classification with missing values. Chapter 5 discusses the integration of the transformer model within an RL framework for CwCF. Each of Chapters

3 through 5 includes its own related work, baselines, methodologies, and experimental results. Finally, Chapter 6 provides the conclusion of the thesis.

2
Background

This chapter gives a formal explanation of the Deep Learning (Neural Networks) and Reinforcement Learning (RL) methods used in this research, based on literature about the topics.

2.1
Neural Networks

Artificial Neural Networks (ANN) [18] have been used in machine learning for many decades, being now a reliable approach in many different use cases. They are inspired by biological brains, where neurons integrate information coming from senses or other neurons, depending on the synaptic weight of their connection.

The solid performance achieved by ANNs makes them a good tool for many tasks, such as classification, regression, temporal series prediction, decision support, and other types of applications. One of the greatest strengths of ANNs is their ability to deal with complicated inputs, serving not only as a function approximator but also as a powerful feature extractor, especially in the case of Deep Neural Networks [19], which integrate many layers of neurons. In that case, information is extracted automatically during training, and not handcrafted. For instance, shapes and colors in a medical image do not need to be filtered manually to be fed to the Deep Learning model, instead, the image is fed as it is and the model is responsible for extracting the features, learned during the training process. Due to this characteristic, they are used to learn patterns from many different data inputs, such as tabular datasets, time series, audio, images, videos, electric/electronic signals, and virtually any kind of data, given appropriate pre-processing and an appropriate architecture related to the data type.

As ANN improvements and new applications were developed throughout the years, today this technology is suited for a myriad of different objectives. Face recognition [20], fraud detection [21], cloud classification using clustering [22, 23], and financial time-series forecasting [24, 25] are just a few examples. To suit different objectives, ANN architectures can change drastically, ranging from feed-forward, recurrent, and convolutional. As in the current thesis we

do not use image data, convolutional networks will not be discussed. Similarly, as measurement vectors have a fixed context length, we consider transformers instead of recurrent networks to process them.

2.1.1 Feed Forward Neural Networks

Feed Forward Neural Networks (FFNN) are the simplest architectures for an ANN, however, they are still largely used today due to their easy implementation and reliable performance in many real-world problem-solving tasks. This architecture is especially used with tabular datasets, since each feature from a data instance can be directly used as an input for the FFNN. They are also known to be general function approximators, since they can approximate any mathematical function, with more or less accuracy, depending on a set of parameters, such as activation functions and amount of nodes and weights.

The general architecture of FFNNs is composed of three main parts, as shown in Fig.2.1: the input, a layer of nodes fed with the data instance feature values; the output layer, which receives the ANN intermediate response as an internal input and outputs the actual estimated desired value (e.g. classification or regression result); and the hidden layer(s), which are responsible for increasing the network complexity and mapping the inputs to the desired output, lying in between the input and output layers.

Hidden and output layers are composed of weights, biases, and nodes with a sum block and an activation function. From a FFNN with L layers, let's consider layer l with dimension J, and layer $l-1$ with dimension D. The output of a node in layer l is denoted by

$$x_j^l = F\left(\sum_{d=1}^{D} x_d^{l-1} w_{dj}^l + w_{0j}^l\right) \tag{2-1}$$

where x_j^l is the output of node j in layer l, x_d^{l-1} is the output of node d in layer $l-1$, w_{dj}^l is the weight between the output of node d in layer $l-1$ and node j in layer l, w_{0j}^l is the bias associated with node j in layer l, and $F(\cdot)$ represents the activation function. In other words, the output of a node is the weighted sum of the previous layer nodes' activations, adding a bias term, and then applying an activation function.

Note that in this notation x can represent both the output of a node in layer l, as in x_j^l, and the input values of that node in the same layer, which are also the outputs of the previous layer, as in x_d^{l-1}. When calculating the first layer outputs, x_i^0 would be the raw input values of a data instance.

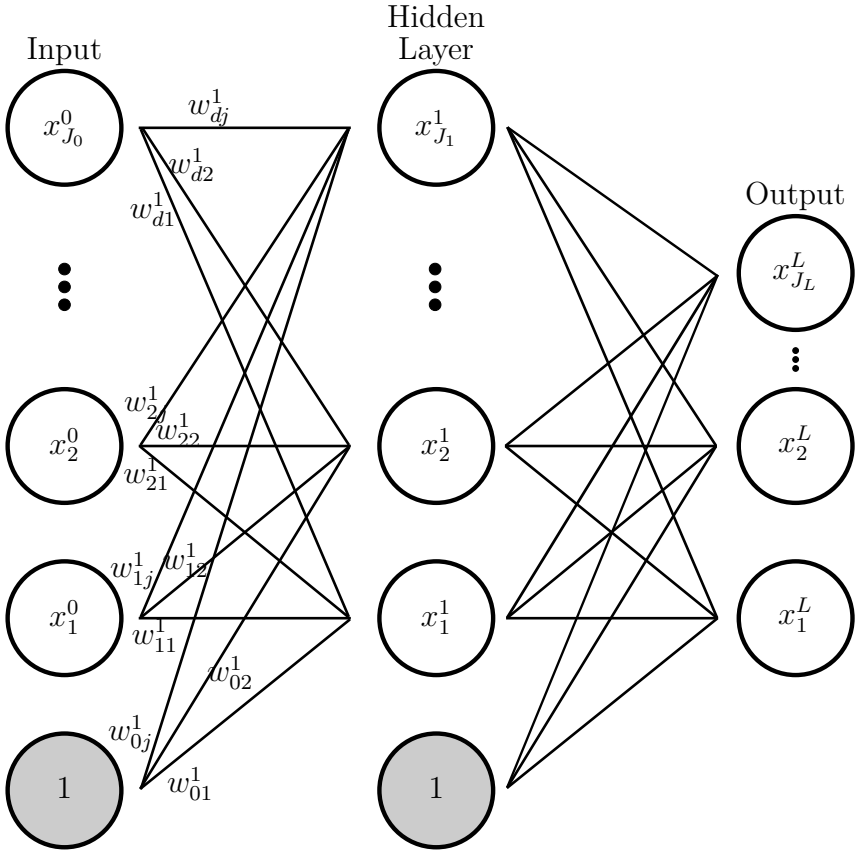


Figure 2.1: General architecture of an FFNN.

To achieve the role of a general function approximator, the FFNN's should have at least one hidden layer[26]. In principle, 2 layers are sufficient to represent any possible region[27, 28].

To offer any benefit over networks with no hidden layers, an ANN should have the ability to represent nonlinear functions, and to do so, the activation function used in the hidden layers is required to be non-linear as well. The role of this activation function is to provide a non-linear output for each node, and, finally, to the Neural Network itself. Otherwise, using a linear activation function in all layers, the output would also be a linear function, lacking representative power to be considered a general function approximator.

2.1.2 Activation Functions

Activation functions are also an important choice when dealing with FFNNs, especially in the last layer of the architecture (output layer). When training an ANN in a classification problem, one probably wants a network that outputs the probability of each class to be the right one, as opposed to a regression problem, where the predicted value, with the same magnitude as the

target, is what is expected to be outputted from an FFNN. Moreover, despite the output layer activation function having to be specifically set depending on the problem, the hidden layer nodes should also have an activation function that encourages learning while being mathematically differentiable. Due to that, different activation functions can be used in hidden layers and output layers. Although there are many possible choices, we only present the ones used in this research.

Linear activation functions are defined as

$$\text{Linear}(x) = x \tag{2-2}$$

and are commonly used to pass on the exact value of weighted inputs and bias calculated, outputting a proportional value. It is commonly used as an output layer function for regression problems, where the aim is to predict a real value of a specific magnitude. The Softmax is also usually used as an output activation function, defined mathematically as

$$\text{Softmax}(x_j) = \frac{e^{x_j}}{\sum_J e^{x_k}}, \tag{2-3}$$

which ensures that all outputs are proportional, so that their sum is exactly one, making the output vector a probability distribution. This function is generally used for multi-class classification problems. Lastly, the Rectified Linear Unit (ReLU) activation function is probably the most used for hidden layers. It is a computationally efficient function that has an overall good performance in most use cases. It can be mathematically expressed by

$$\text{ReLU}(x) = \max(0, x). \tag{2-4}$$

2.1.3 Training

Like many other supervised learning methods, Neural Networks learn by minimizing the difference between the predicted value and the real value of the data instances, i.e. minimizing error over a data set. This error is given by a loss function $L(\mathbf{w})$, a common choice for regression being the mean squared error

$$L(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - \hat{y}_i(\cdot; \mathbf{w})\right)^2,$$

where $\mathbf{w}$ are the weights, i.e. parameters of the network, N is the number of data instances, $\hat{y}_i(\cdot; \mathbf{w})$ is the i-th predicted output, and y_i is the i-th real value.

The loss function is minimized iteratively by the network weight updates.

This is where in fact the learning occurs. The weight update rule can be defined as

$$\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}. \tag{2-5}$$

To calculate the weight updates $\Delta\mathbf{w}$, many algorithms leverage the gradient of the error (or the loss). The Gradient Descent algorithm is the most basic optimization algorithm for ANN's and to calculate the updates $\Delta\mathbf{w}$ in 2-5, the algorithm moves in the opposite direction of the gradient in order to minimize the loss (hence gradient descent), such that the update is

$$\Delta\mathbf{w} = -\alpha\nabla L(\mathbf{w}), \tag{2-6}$$

where $\alpha > 0$ is the *learning rate*, and the gradient $\nabla L(\mathbf{w})$ is calculated through backpropagation[29]. A learning rate that is too big can lead to no convergence, never finding the optimal values. On the other hand, a learning rate that is too small can lead to slow learning, making it impracticable.

In the Gradient Descent algorithm, the updates are made after each evaluation of the whole data set, always requiring a new evaluation (with updated weights) to continue iterating until convergence. This is called *batch-learning*. In order to make the algorithm more feasible, an online version was developed. The most known is the *Stochastic Gradient Descent* (SGD), which follows the same principles, while, instead of evaluating the whole data time after time, it implements an online version, updating weights based on evaluations of a random single instance at a time or multiple instances at a time, in a mini-batch fashion. The most used optimization algorithm today is an extension of SGD, called Adam [30].

2.1.4 Validation and Testing

When training an FFNN, and most supervised machine learning models, it is often advisable to split the data into three groups, where each one is used for a different purpose. The first split is the *training set*, which is used to feed the learning model. In fact, the model learns the patterns only with respect to the training data. The second split is the *validation set*, which is used to evaluate the model performance. The validation set does not serve as information for the model to learn, as opposed to the previous set, but it functions as a gauge, measuring how well the model would behave in a real prediction with unseen data. This split is usually used as a tool to check if the model is learning (during development) and to avoid over-training (overfitting). And lastly, there is the *test set*, which in fact has only unseen data from the dataset, simulating an exact real scenario. The test set is usually used as a

final and definitive performance evaluation.

2.1.5 Architecture

The more hidden layers (or nodes) there are in an ANN architecture, the more complexity is added to the approximated function. However, indiscriminately increasing the number of hidden layers does not mean that the approximator will have a better performance. Too many hidden layers (or nodes) can lead to an over-complex model that will learn the patterns necessary to have a good performance in training data. But it may be too specialized, so much that it will fall in performance for unseen patterns, i.e. validation and test data. This behavior is contrary to the main objective of an ANN, generalization, and it is known as overfitting.

The best ANN architecture for a specific case scenario remains an unsolved problem in the ML field. In that case, to overcome the overfitting problem, since the "right" architecture is never known for sure, a training technique is used to maintain the generalization of the network, the *Early-stopping*. Early-stopping has the main objective of stopping the training whenever the model starts to learn the training set *without* any improvement on the validation set. This behavior means that the model has stopped learning the true data distribution and is memorizing the training set, becoming better at patterns already seen, but not improving the performance on unseen data (and actually getting worse). Usually, a *patience* parameter is set, which controls after how many epochs (when all training data is passed through the model) with no improvements the training will stop, while keeping the last best model.

2.2 Transformers

In recent years, a new ANN paradigm has emerged: the Transformer [36] architecture. The industry has been taken over by new transformer-based models, and companies run in an unspoken contest to create the best model. This recent architecture was originally built to handle Natural Language Processing (NLP), specifically, a machine translation problem. It achieved surprising performance and provided a way to quantify the relationships between words, from input, output, or between both.

This architecture has some core concepts, and the most important can be considered a technique called Attention [36], which, in a way, learns context

relationships between elements of input sequence(s), storing it in an "Attention Matrix". The next sections explain the core concepts used in a Transformer.

2.2.1 Tokens

Considering that the transformer was created to deal with machine translation (i.e. words), and ML models can not handle words naturally, tokens are a way to map words/characters/pieces of words to numerical values that can be used by a machine learning model. There are several tokenization techniques, and the performance varies depending on many aspects, such as the model used and language structure. A simple visual representation is shown in Figure 2.2. As we do not deal with NLP in this thesis, we will refrain from being more detailed about this specific technique.

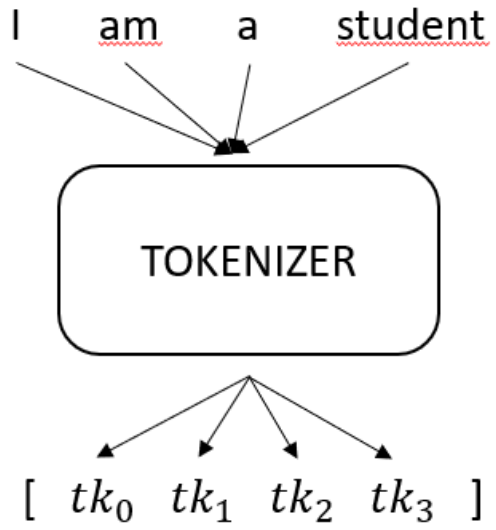


Figure 2.2: Tokenization example. Each word is transformed into a numerical representation. The full sentence is represented by a vector formed by these values.

2.2.2 Embeddings

While tokens can be considered a "raw" representation of an input, embeddings are a way to improve this representation, by transforming them into learnable float vectors, that can allow the model to capture relationships and similarities about inputs and their contexts. This is true not only for tokens, but for any input that goes through an embedding process. The example in Figure 2.3 shows how we can "decompose" information from one value to a vector with more elements.

It is important to highlight that this embedding layer is trained together with the rest of the model modules, aiming to learn the best possible embedding to maximize performance. In the case of categorical inputs such as token IDs, it is a table, while for real-valued inputs such as feature values, it is generally a dense neural layer.

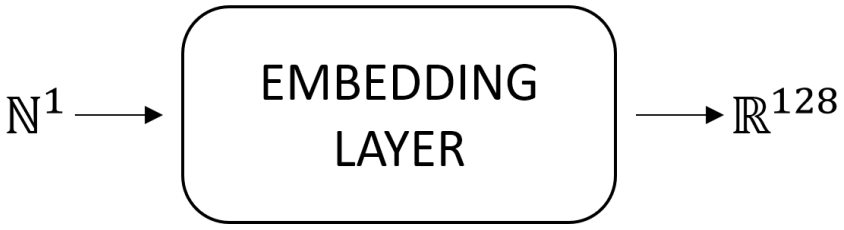


Figure 2.3: Embedding layer. In this example the layer is increasing the dimension of each token to a learned representation.

2.2.3 Encoder

This structure is a stack of layers, capable of extracting contextual information, e.g. through self-attention, from the input embeddings. In other words, this structure learns relationships between the input elements, through the self-attention mechanism, and builds a representation (usually in a space with a smaller dimension) that will feed the decoder.

The encoder is not always needed, especially when input data is already well structured and has no sequential dependency, e.g. tabular data or static images. On the other hand, there are problems where a decoder-only architecture is better suited, especially in problems handling input comprehension, such as text classification and sentiment analysis.

Figure 2.4 shows the structure of the original transformer encoder.

2.2.4 Decoder

The decoder structure aims to generate the actual outputs of the model. It takes the encoder outputs as inputs alongside its previous outputs, usually functioning in an auto-regressive way. As mentioned, it can work without information from the encoder, generating sequential outputs based solely on current input. It relies on cross-attention between the encoder and decoder (if the encoder is present), as well as self-attention.

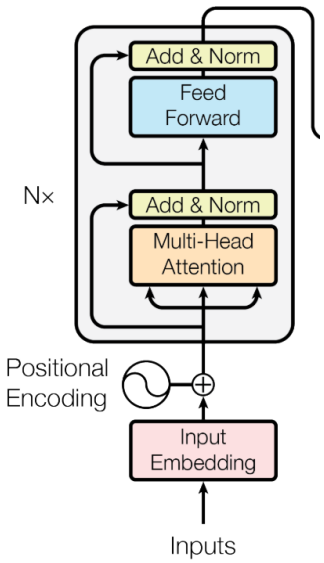


Figure 2.4: Encoder from the original transformer [36]. Inputs are a combination of embeddings and positional encodings. In each encoder block, a multi-head attention layer is followed by a feed-forward layer.

2.2.5 Attention

The attention mechanism, introduced in [37, 38], for Recurrent Neural Networks, was created to overcome limitations with the fixed size context length, allowing the model to focus on relevant parts of the input.

Later, this mechanism was improved to work with transformers and is considered the core of this architecture. It creates a form of communication between inputs, making context relationships, which was crucial for machine translation, and later found that it is a powerful approach in many other scenarios. Following is a description of this mechanism.

Initially, each input embedding is multiplied by three trainable matrices (linear layers in practice), W_Q, W_K and W_V, generating three matrices, the query Q, the key K, and the value V. These matrices usually have the same dimensionality as the embedding vectors d_{emb}, even though it is not necessary, and can be thought of as input abstractions, each one having its own role. The query Q can be understood as a token communication asking for a kind of information. The key K can be understood as what kind of information the respective token contains. Whereas the value V can be understood as the shared information between tokens.

Next, there is a communication between each query with all keys. This communication is made by the Q and K dot products. In practice, for auto-regressive cases, only communications with previous tokens are allowed, more on that later. This generates a matrix with relationships, or affinities, between

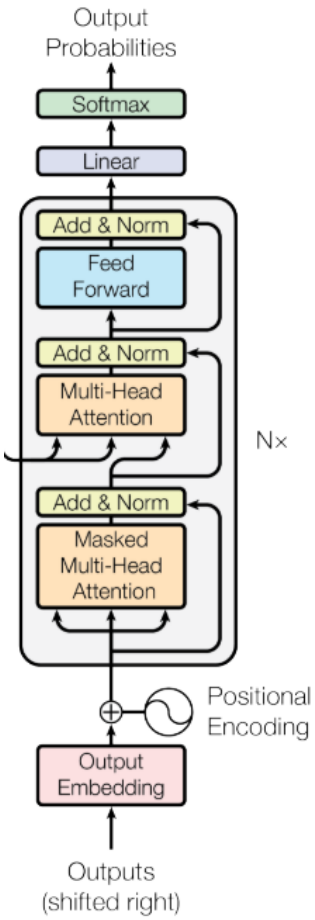


Figure 2.5: Decoder from the original transformer [36]. Similar to the encoder, the inputs are a combination of embeddings and positional encodings. In each decoder layer, there is a masked multi-head self-attention layer followed by a multi-head cross-attention layer and lastly a feed-forward network layer.

queries and keys. To maintain stability, the matrix values are divided by the square root of the matrices dimension (usually $\sqrt{d_{emb}}$). Each row is normalized through a softmax layer, which reveals how much of that token (key) should be aggregated in that position (query).

With the resulting matrix in hand, the final step is the product between this matrix and the value matrix. The outcome is a vector for each input token, which later goes through a feed-forward layer. The attention mechanism can be mathematically formulated as

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_{emb}}}\right)V. \tag{2-7}$$

This process defines one layer of the attention mechanism. Usually, a transformer model has many layers stacked (6 in the original paper), each with trainable parameters W_Q, W_K and W_V. This allows for a more complex representation of the input, where shallow layers learn to represent simpler

concepts, e.g. meaning of words, and deeper layers learn to represent more complex concepts, e.g. expressions and ideas [39]. The described mechanism is represented in Figure 2.6

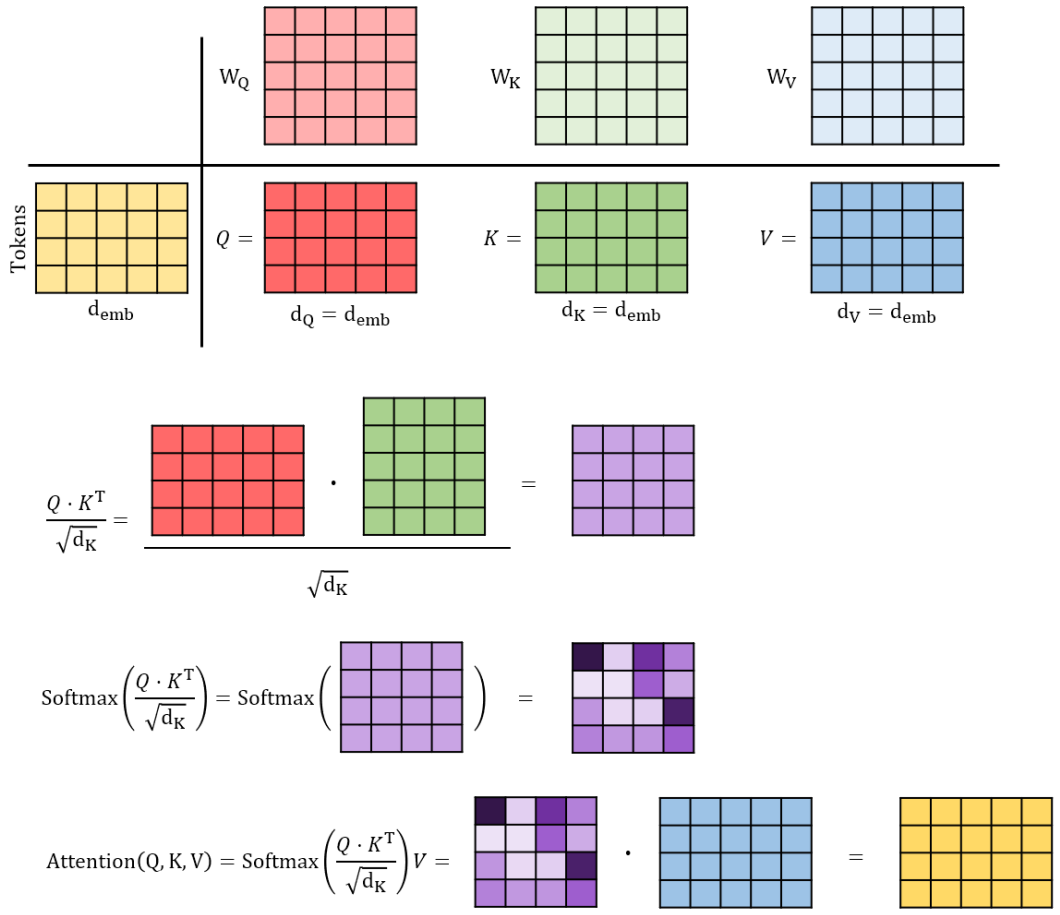


Figure 2.6: Visual Representation of the Attention Mechanism, inspired by [73]. The weights matrices are learned. One can see how Q, K, and V are related in this visual representation.

2.2.6 Multi-head Attention

In [36], authors use multiple attention heads to process embedding vectors. This allows context information to be interpreted in different ways, learning different relationships between input vectors.

To achieve multi-head attention, the same attention mechanism described above is repeated, in parallel, a number of times. In the original paper, 8 heads are used. With that, not one, but many attention scores from equation 2-7 are created. This allows for different representation spaces in the same attention layer, each having the freedom to focus on different tokens and building different context relationships.

Finally, as each attention head produces a matrix called Z, result of equation 2-7, they are concatenated and then multiplied by a learnable matrix W^o, with such a dimension that brings the output back to the original shape, allowing it to serve as the input for the next layer. Figure 2.7 depicts this process.

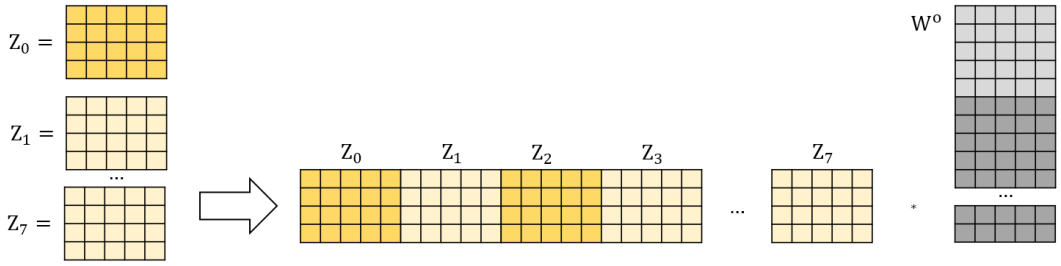


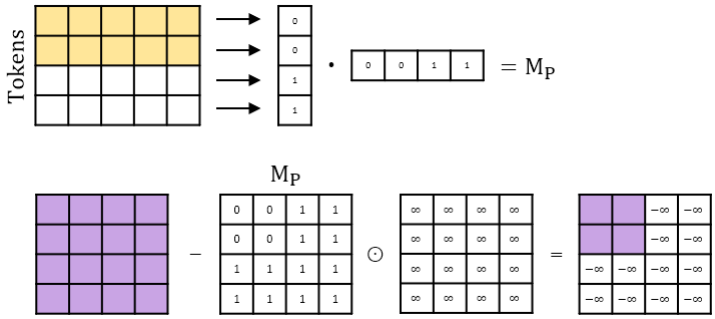
Figure 2.7: Multi-headed attention. Each attention head produces a matrix Z. These matrices are concatenated and multiplied by W^o to bring the dimension back to the original shape.

2.2.7 Masks

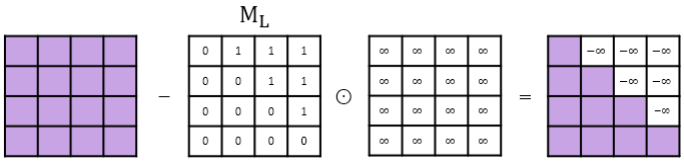
Masks are a way to tell the model what in the sequence is supposed to be ignored. This has mainly two applications. First, it can be used in varying length inputs, and second to prevent attending to future information in the attention block.

Considering that a model is not naturally allowed to deal with varying length inputs, the transformer model achieve this feature by using what is called a *padding mask* M_P. If an input sequence is smaller than the one considered by the model, the excessive elements are initially populated with placeholder values. Inside the attention block, right before a softmax is applied to the attention matrix ($Q \cdot K^T$), any key-query combination related to these placeholder values is replaced with negative infinity, inducing a zero after the softmax. With that, they are essentially disregarded in attention calculations. This process is represented in Figure 2.8a.

The second application can be called "look-ahead mask", or triangular mask. This mask applies sequential limitations during attention, or in other words, it ensures that queries do not communicate with future keys, maintaining the temporal bounds between tokens. In practice, also before the softmax, values outside the lower triangular matrix are set to negative infinity so that after softmax they are disregarded. This masking process is represented in Figure 2.8b.



(a) Padding mask



(b) Look-ahead mask

Figure 2.8: How masks are applied in attention mechanism. In (a) the padding mask is depicted, showing which elements of the attention matrix are used after the mask is applied. In (b) the look-ahead mask is represented, not allowing queries to communicate to future keys.

2.3 Reinforcement Learning

Reinforcement learning (RL) is a computational approach to understanding and automating goal-directed learning and decision-making [31]. It is based on the way animals learn behaviors: by trial and error, reinforcing successful behaviors and suppressing unsuccessful ones. The formulation of an RL problem is deeply attached to the Markov Decision Process (MDP), which is a flexible mathematical framework to model sequential decision-making. As such, the objective of an RL *agent* is to solve an MDP problem, using available information from the *environment* and deciding which *actions* to take at each timestep t, to accomplish a goal, specified in terms of *rewards*. We will now briefly introduce these and other key concepts of RL, which are illustrated in Figure 2.9.

The *agent*, in an RL framework, is responsible for interacting with the environment, gathering information, and taking the right actions to achieve the goal. The agent is the part where learning occurs, which takes place through the use of experience gathered over time, once it tries different actions and stores the outcomes, learning the right policy to achieve a goal. The agent interacts with the environment performing an *action*. It is what transitions the agent from one state to another. The actions follow a *policy*, a set of rules that determines the decisions (actions) made by the agent in order to achieve

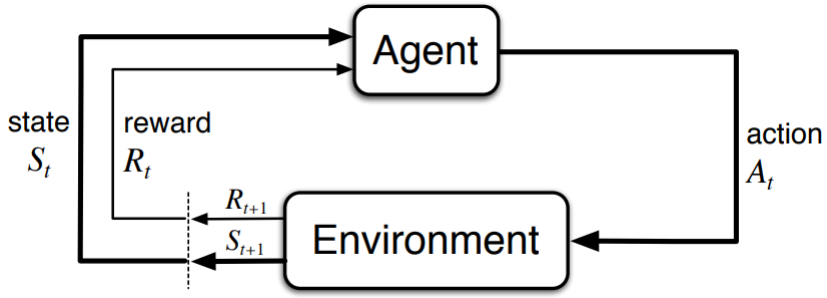


Figure 2.9: Representation of the interaction between agent and environment. The agent, which is able to "sense" the current state, interacts with the environment through an action. The outcome of such interaction is a change of state and a reward. Figure extracted from [31]

the goal. It is the core of an RL solution, as it controls the behavior of the agent, aiming to achieve higher rewards over time.

Surrounding the agent is the *environment*, i.e., everything that the agent may interact with. It is through the interaction (actions) with the environment that the agent changes its state, to gather experience and learn the optimal policy to achieve a goal. The *state*, can be interpreted as the information regarding the environment, which the agent is capable of measuring. It is what the agent senses about the environment. And finally, the *reward* is a goal-related numerical feedback given by the environment to the agent, as a response to an action.

2.3.1 Markov Decision Process (MDP)

The MDP is formulated as a tuple consisting of $< \mathcal{S}, \mathcal{A}, T, R, \gamma >$, where $\mathcal{S}$ is the set of all possible states $s \in \mathcal{S}$, $\mathcal{A}$ is the set of all possible actions $a \in \mathcal{A}$, T is a transition function $T(s, a, s') \rightarrow Pr(s'|s, a)$ that maps a state and action to a probability distribution of going to next state $s' \in \mathcal{S}$, R is the reward function defined by $R(s, a, s') = r$ which defines the reward signal for each possible transition between states and γ is the discount factor, that assumes values in the range $[0, 1]$ and is responsible for balancing the importance between immediate rewards and delayed rewards.

In this research, all MDP setups are considered to have the Markov Property, which determines that the probability of each possible value of s' and r depends only on information available on the immediately previous state s, or in other words, in all timesteps, represented by index t, the state s must provide all necessary information about agent-environment interaction that is necessary to select the optimal action. With that in mind, the given definitions

of reward and transition function should suffice to calculate any outcome from the MDP dynamics

$$p(s', r|s, a) = Pr(s_t = s', R_t = r|s_{t-1} = s, a_{t-1} = a). \tag{2-8}$$

RL algorithms aim to maximize expected discounted reward through time (also called the *return*), and there are a few different approaches to doing so. In this research, the focus is on the maximization of return by leveraging action-value functions (Q-functions), which is an estimate of how good it is to be in a specific state and take a specific action. This estimate is calculated based on the discounted return, which is defined as

$$G_t = \sum_{k=0}^{\mathrm{T}} \gamma^k r_{t+k+1}, \tag{2-9}$$

where t represents the current timestep of an episode, k represents the subsequent timesteps, T is the final timestep, γ is the discount factor and r_{t+1} is the reward related to the current state and action.

Let π be the policy of an RL agent. It is defined as the probability of taking an action a in the current state s, as in

$$\pi(a|s) = Pr(a|s_t = s). \tag{2-10}$$

In order to achieve the best possible behavior, the RL agent must follow a policy that is optimal, in other words, a policy that maximizes expected discounted returns. Let π^* be the optimal policy, it is defined as

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}[G_t]. \tag{2-11}$$

Note that to optimize expected return, the model can not follow only the optimal policy since the beginning of the training. At first, the policy learned is not optimal, as the learned value function is not the true one. If the model always follows a policy that returns the maximum expected return *exploiting* the information gathered so far, it is likely to fall in a local minimum. In order to better estimate the value function, it needs a policy that *explores* different actions for all states, gathering more information and consequently improving the value function estimate. At the same time, always exploring different actions will not lead the agent to the goal state, and the optimal policy (the objective of an RL model) will not be followed. This is a well-known problem in RL known as "the *exploration-exploitation* dilemma". A good strategy, usually used, is to mainly follow an *exploration* policy when training begins, slowly giving place to an *exploitation* policy, that always looks for the actions which maximize expected return.

A widely used approach to balance the two policies is the ε-greedy policy.

In this policy, an action is chosen randomly according to a probability ε. The ε-greedy policy follows the rule:

$$\pi_\varepsilon(a|s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}|} & \text{if } a = \arg\max_{a'} Q(s, a') \\ \frac{\varepsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases} \tag{2-12}$$

where $|\mathcal{A}|$ is the number of possible actions in $\mathcal{A}$, and ε is an exploration rate between 0 and 1. As such, the policy is greedy most of the time, but with probability ε chooses an exploratory action.

2.3.2 Action-Value Functions (Q)

As mentioned above, the *action-value function* is an estimate that measures how good it is to be in a state $s \in \mathcal{S}$ and take action $a \in \mathcal{A}$. In other words, the action-value function Q measures the *expected return* of a tuple (s, a) following policy π, such that

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t | s_t = s, a_t = a]. \tag{2-13}$$

We can write Equation 2-9 in a recursive form, deriving

$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots \\ &= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + \cdots) \\ &= r_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2-14}$$

and allowing us to rewrite Equation 2-13 into

$$\begin{aligned} Q^\pi(s, a) &= \mathbb{E}_\pi[r_{t+1} + \gamma G_{t+1} | s_t = s, a_t = a] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1})]. \end{aligned} \tag{2-15}$$

Equation 2-15 is the *Bellman equation* for Q_π. It determines that when following policy π, the Q-value of the current state and action is dependent on the Q-value of the state and action in the next step, following the same policy, along with the immediate reward.

As stated before, to solve an RL problem, the agent aims to find a policy that leads to the highest discounted return possible in an MDP, that is, finding a policy that maximizes expected return. Let the optimal policy be called π^* and the optimal Q-value be Q^*. The optimal policy π^* leads to the maximization of return through time, and to achieve that, the actions that maximize returns must be chosen for the next states. Therefore, from Equation

2-15, we get the *Bellman optimality equation*.

$$\begin{aligned} Q^*(s,a) &= r_{t+1} + \gamma \max_{a'} \mathbb{E}_\pi[G_{t+1}|s_t = s, a_t = a] \\ &= r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \end{aligned} \tag{2-16}$$

In order to deal with an increased state and action space, *function approximators* are often used in combination with RL. They are particularly helpful mainly because the time and memory needed to deal with a big state space is usually an obstacle for RL algorithms that do not leverage approximators. The key feature added by the function approximators is the *generalization* ability, i.e., when using them, there is no need to encounter all possible states and actions, because they generalize from similar ones. Many different function approximation methods can be used with RL to achieve the mentioned generalization. They can be parametric or non-parametric functions and generally fall into the *supervised learning* methods. This type of method expects to receive a tuple of *(input, output)*, where the input is the available information and the output is the true value related to the input, aiming to learn the mapping between them and reproduce it for unseen inputs, as described in Section 2.1.3.

The outputs mentioned are called "targets" in an RL scenario, being an updated estimate of the Q-value. During training, these Q-value estimates need to be improved as the agent experience increases, thus they are updated frequently. This update is based on Equation 2-16, and for parametric function approximations, the update rule depends on the function parameters θ:

$$Y_t^{\mathrm{QL}} = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) \tag{2-17}$$

This is the target used in *Q-learning*, when using function approximation [31]. In this setup, the function approximator estimates the Q-values for the next state and action $Q(s_{t+1}, a'; \theta)$, according to the current parameters θ. The approximator is then trained, either after each timestep or based on a memory of experiences (batch learning), by the loss function, which in many cases is the mean squared error between the current estimate and the expected output (the target in that case), as in

$$L(\theta) = \left(Q(s_t, a_t; \theta) - Y_t^{\mathrm{QL}}\right)^2, \tag{2-18}$$

where L is the loss function and Y_t^{QL} is the Q-learning target value.

2.3.3 Deep Q-Network (DQN)

One of the biggest RL breakthroughs in recent years is the use of Artificial Neural Networks for approximating the Q function. In the work of Mnih et al. [32, 33], not only an ANN was used as a function approximator, but also as a feature extractor for raw images. This approach was derived from standard Q-learning.

A few additions to the original Q-learning were developed besides the usage of an ANN. One of those additions is the *experience replay* ($\mathcal{D}$). It works as a memory for all transitions happening during the episodes, at each timestep, storing the agent's experience as a tuple $(s_t, a_t, s_{t+1}, r_{t+1})$. When training the ANN, usually after each timestep, a group of tuples extracted from $\mathcal{D}$ is randomly selected. Then this group, called *mini-batch*, is used to carry out training steps.

Besides the use of experience replay, Mnih et al. also applied a *target network*. In order to calculate the Q-learning target update in a setup using a parameterized function approximator (such as an ANN), the parameters of such approximator are also used to estimate the Q-values, i.e. the Q-value targets used to train the approximator, and the Q-values estimates are dependent on the same approximator parameters. This is problematic, as it makes both the targets and the estimated Q-values highly correlated, leading to instability and divergence. The target network aims to address that problem, being a delayed copy of the Q-network (the learning agent). After a defined number of updates, the weights from Q-network are copied to the target network. While calculating the target value, instead of using the original network to estimate Q-values, as in Eq. 2-17, it uses the target network to make those estimates:

$$Y_t^{\text{DQN}} = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-), \tag{2-19}$$

where θ and θ^- are respectively the original Q-network and target network parameters, and Y_t^{DQN} is the DQN target value.

In practice, the action is not an input to the network. Instead, the network outputs one Q value per action. Y_t^{DQN} only defines the target value for the taken action a_t, but for training we need a target *vector* y_t, with the same dimension as set $\mathcal{A}$, indicated by the index i. This vector is constructed by keeping the values of all other actions constant, such that

$$y_{t,i}^{\text{DQN}} = \begin{cases} Y_t^{\text{DQN}} & \text{if } a_i = a_t \\ Q(s_t, a_i; \theta) & \text{otherwise} \end{cases}. \tag{2-20}$$

2.3.4 Double DQN (DDQN)

As shown in [34], the DQN algorithm suffers from maximization bias, which is an overestimation of the Q-values. This happens because Q-values are estimates. As such, their true values are initially unknown and inaccuracies are necessarily going to happen during learning. These inaccuracies lead to overestimation due to the max operator in the target update equation 2-19. This leads to destabilization in training.

To overcome that problem, in [34], the authors continued using the target network from the DQN algorithm to estimate the Q-values, although the chosen action was changed to the one that maximizes the original Q-network output, as in

$$Y_t^{\text{DDQN}} = r_{t+1} + \gamma Q\left(s_{t+1}, \arg\max_{a'} Q(s_{t+1}, a'; \theta); \theta^-\right). \tag{2-21}$$

2.3.5 Dueling DQN

One more improvement made in Deep Reinforcement Learning was developed by [35]. In this approach, a different network architecture was created in order to address both state value $V(s)$ and action advantages $Q(s,a) - V(s)$. The state value and the advantage values are decoupled inside the network architecture, but aggregated in the end, with final outputs the same as in DQN. When separating the two estimates, the model can learn which states are valuable regardless of how actions will affect them.

With that, the changes between the conventional DQN (or DDQN) and the dueling architecture were minimal, and no adaptation between them is needed, besides a change in the network architecture.

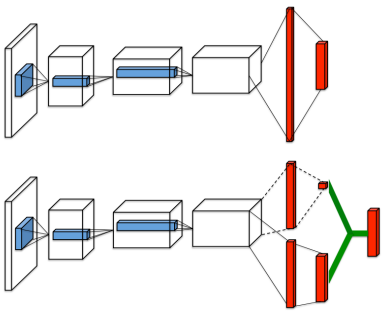


Figure 2.10: The architecture representation of a common DQN (top) and the Dueling DQN (bottom). Figure extracted from [35]

Figure 2.10 shows the dueling architecture. The network is divided into

two streams, one leading to a single value, the state value, and the other leading to a vector, the advantage values (one for each action). They are merged to output the Q-values, according to the network parameter θ, such that

$$Q(s,a;\theta) = V(s;\theta) + \left(A(s,a;\theta) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a';\theta) \right). \tag{2-22}$$

In Equation 2-22, the advantage function $A(s,a)$ follows the definitions

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \tag{2-23}$$

and

$$V_\pi(s) = \mathbb{E}_{a\sim\pi}[Q_\pi(s,a)] \tag{2-24}$$

from [35].

As such, it is possible for the model to learn separately the value of a state and the advantage of choosing each action. This setup makes it possible for the model to learn the relation between each action advantage and the average advantage value. This is especially helpful in scenarios where the actions might not always affect the environment, and states by themselves have more meaningful information. In [35], the authors showed that the dueling architecture improved convergence time and performance compared to the standard DQN.

2.4 Datasets

In order to assess the viability and performances of proposed methods in this work, a group of tabular datasets is used to conduct experiments. In total, six datasets were used. One is a synthetic dataset, while others are datasets with real-world data.

Four of them are extracted from [40], they are: "Pen Digits" [41] ,"Wine" [42], "Miniboone" [43], and "Beans" [44]. The "Healthrisk" dataset is extracted from [45], and preprocessed to have classes that relates the body and birth ages. All datasets are suited for a classification problem since all targets are composed of categories. The "Miniboone" and "Wine" datasets were present in related works [13, 14], hence they were chosen. Other datasets were searched with the objective of different amounts of classes and samples while having only continuous features. Moreover, datasets do not have missing features originally, and a preprocessing step was carried out in all datasets, normalizing features in the range $[0, 1]$. The datasets detailed characteristics can be seen in table 2.1

Table 2.1: Datasets Characteristics

	Features	Classes	Samples
Synthetic	2	2	20000
Wine	13	3	178
Miniboone	50	2	130065
Healthrisk	15	3	702
Pen Digits	16	10	10992
Beans	16	7	13611

3
Deep Reinforcement Learning with embedded supervised learning for Classification with Costly Features

In this chapter, a Deep Reinforcement Learning (DRL) approach is proposed, aiming to deal with CwCF problems. Later, a baseline is defined, along with experiments for comparison between models.

As noted in Chapter 1, the original motivation stemmed from the ever-increasing costs associated with healthcare. Consider a scenario in which a patient undergoes multiple medical examinations for a diagnosis. While conducting numerous tests can provide a comprehensive view of the situation, it may not always be the most cost-effective approach. To address this, a decision support algorithm could be implemented to optimize the sequence of information gathered for diagnosing a disease. Notably, this concept of optimizing the sequence of measured features for classification extends beyond healthcare. For instance, sensor systems with memory limitations could benefit from a similar approach, processing only the necessary information to complete a task. As well as real time applications can benefit from this paradigm, acquiring only enough information for a task.

3.1
Related Work

Today, most state-of-the-art RL algorithms [74, 75] leverage the power of function approximators, especially neural networks, to obtain the best results and improve representations, as mentioned in the previous chapter. The models that rely on both Reinforcement Learning and Neural Networks are known as Deep Reinforcement Learning (DRL) methods.

Usually, DRL algorithms are not used to learn classification patterns. That is due to the fact that they require far more training time than other machine learning methods, such as statistical learning, ANN's, tree-based methods and others. Another reason is that DRL algorithms are suited for sequential problem solving, as they are built on top of MDP's, therefore a classification problem should be modeled as such before a DRL algorithm could be used.

Although not usually used directly for classification, due to the sequential

nature of the method, DRL is suited for different kinds of tasks involving classification, like active learning [76] and feature selection [77]. Nevertheless, related work has explored the idea of using DRL methods to serve both as a classifier and for sequential feature acquisition.

In this approach, called here *DRL multi-step classification*, the agent can choose which feature to read, and at some point decide to which class the data belongs. In other words, the actions may serve for acquiring information sequentially and for classification, terminating the task.

3.1.1 DRL Multi-step Classification

In this approach, each episode comprises a data instance. As mentioned, the actions serve to choose which information to retrieve and also to classify the data retrieved so far. Usually, in this kind of problem, the main objective is two-fold: increase classification performance, while constraining the use of resources. So besides the usual classification error minimization, the algorithm also aims to use the least amount of possible scarce resources (e.g. features, time, money, etc), minimizing cost and therefore learning to trade-off between these two objectives. In this scenario usually, each feature has an intrinsic cost based on the scarce resource considered, which impacts how the model prioritizes each of them.

An approach to that problem was implemented by [46], where the authors introduced the modeling of a classification problem as an MDP, with a sequential feature acquisition, aiming to minimize classification error as well as the number of features used. In this work, the states were defined by a tuple containing two n-dimensional vectors (x, z), where x is drawn from $\mathcal{X} \in \mathbb{R}^N$ and represents the input vector being classified, and z is drawn from $\mathcal{Z} = \{0; 1\}^N$ representing the selected features in the first vector. The values in x are x_n, with $1 < n < N$ and in z the values are $z_n = 1$ for selected features and $z_n = 0$ otherwise. As for the actions, the set of possible actions in state (x, z) is denoted by $\mathcal{A}(x, z)$ and they are divided into two sets: $\mathcal{A}_\mathrm{f} = \{\mathrm{f}_1, \mathrm{f}_2, \ldots, \mathrm{f}_N\}$ and $\mathcal{A}_y = \mathcal{Y}$, the first representing the actions that would select a feature to be used, such that $a = \mathrm{f}_n$ corresponds to choosing feature n and the latter corresponding to assigning a label to the current data instance, terminating the episode.

Another work [47] was published in the same year which also modeled the classification problem as an MDP, but with different characteristics. In this work, the objective was to investigate how the proposed RL algorithm would perform in a classification task. The MDP states are formed by a vector three

times the size of the input, i.e. given that x is a data instance from a dataset, with length N, and each element of x is represented by x_n. The state is a vector composed of three equal parts, called 'buckets', with total length $3N$. The first bucket, B1, is a copy of x, the second, 'attend' bucket B2, is initially composed of zeros and the third, 'ignore' bucket B3, is initially also a copy of x. The actions comprise either setting an element from B2 such that $B2_n = x_n$, or setting an element from B3 such that $B3_n = 0$. The idea is to have an agent for each class, which would return a positive reward if the class is the one represented by it and a negative reward otherwise, based on the learned augmented representation. The authors used an extension of the Actor-Critic Learning Automaton (ACLA)[48] algorithm, using neural networks.

More recently a DRL algorithm for Classification with Costly Features (CwCF) was developed in [13] and extended in [14]. The CwCF arises when, to classify a data instance, each feature used incurs an intrinsic cost. The objective of the first work was to provide a flexible framework to classify data instances in a dynamic way, choosing features based on acquired values, while minimizing classification error and the cost necessary to classify. To achieve that, the authors modeled the classification problem as in [46], modifying the reward function to accept a different cost for each feature. For the actions, the authors maintained the modeling of the same research, with actions that choose a feature to "measure", $\mathcal{A}_\mathrm{f}$, or actions representing the classifications, $\mathcal{A}_y$. Also, the DQN algorithm was used, leveraging the recent improvements over DRL methods, such as experience replay, Double DQN, and Dueling architecture, which provided a robust algorithm for budget-constraint classification during the test phase. They also implemented an action that used a pre-trained 'high-performance classifier' (HPC) as a terminating step for data instances that were considered more difficult by the DRL algorithm. In [14], the authors improved on top of the previous proposes, adding the option to set a specific average budget (as opposed to the previous indirect average budget) or a 'hard budget', in which the agent was not allowed to exceed the specified budget for features used.

Another RL approach to deal with the CwCF problem is implemented in [78]. In this work, authors leverage the Opportunistic Learning paradigm, using a DQN model as a learning agent. The problem is modelled similarly to other cited approaches in terms of states and actions. Differently, it bases its prediction probabilities on Monte Carlo Dropout, deriving what authors called prediction uncertainty to assess the impact of feature acquisition.

Learning happens in an online fashion, and two networks are implemented, one for prediction, other for estimating action values. The first, as

the name suggest, is responsible for making predictions, based on the prediction uncertainties. The latter is responsible for estimating the values of each feature being acquired. The representations learned from the prediction network are shared with the value network, more specifically, outputs of each layer in the prediction network serve as input for the adjacent layer in value network. Three datasets were used as benchmarks, and results showed good performance compared with the baselines proposed.

Lastly, in [79], the author propose a similar MDP modelling of the problem, with a cost-sensitive loss function, composed of a budget and a loss function. The loss provides the model with a hard-budget constraint, and can be cost-sensitive, i.e. answers with final lower budget are more valuable than the opposite, this is achieved by considering the information gain provided by the feature acquisition related to the added cost. In this approach, states store features, respective values and total costs, and actions can only be features. The classifier, a logistic regression, is trained in an interleaved step with the policy. This is done because the policy training depends on the classifier entropy, while the classifier depends on the distribution over states induced by the policy.

Differently from the mentioned work, the proposed approach applies training strategies that effectively utilize available information about the data during training, for instance, updating all actions per training step, exploring a specific set of actions, and limiting actions that clearly do not provide any benefit for the objective.

3.2 MDP model

The use of RL for sequential problems can be considered one of the natural choices because RL algorithms are naturally suited for this kind of problem. In recent decades, DRL has been extremely fruitful, achieving state-of-the-art performance in different applications [75, 74]. Following that, this thesis proposes a DRL method to solve the stated problem, sequentially gathering information with costs and ultimately classifying a data instance with as little information as possible, for different tabular data sets.

A problem should be first modeled as an MDP to be suited for a DRL approach. Given that the available information, in many cases, comes in the form of tabular data sets, in this work the MDP was modeled for this kind of data. The proposed formalization for the MDP is presented next.

Let $\mathcal{X} \in \mathbb{R}^N$ be the input vector distribution with dimension N to be classified, and $\mathcal{Y} \in [1, C]$ be the class distribution related to $\mathcal{X}$, comprising C possible different classes. The tuple (x, y) drawn from $(\mathcal{X}, \mathcal{Y})$ comprises

the environment of each episode for the agent. The states are defined as set $\mathcal{S} \in \mathbb{R}^N$, where the initial state is always denoted by $s = \{-1\}^N$, representing the unread data features, since data is normalized in the range $[0, 1]$. When a feature is read, it receives the corresponding value of the input vector, such that $s_n = x_n$, with $n \leq N$ representing the vector dimension. As for the actions, they are represented by the set $\mathcal{A} = \mathcal{A}_\mathrm{m} \cup \mathcal{A}_\mathrm{y}$, where $\mathcal{A}_\mathrm{m}$ is the set of measurement actions and $\mathcal{A}_\mathrm{y}$ is the set of classification actions. Differently from [46], states are represented by a single vector with a placeholder value being used for unmeasured features.

Measurement actions $a_\mathrm{m} \in \mathcal{A}_\mathrm{m}$ are responsible for updates in states and represent information being acquired. They are defined in such a way that $\mathcal{A}_\mathrm{m} = \{\mathrm{f}_1, \mathrm{f}_2, \ldots, \mathrm{f}_N\}$, where each possible measurement action is attached to a feature value, causing the agent to read it, and update the corresponding dimension in the state vector with the same value of the feature read. On the other hand, classification actions $a_\mathrm{y} \in \mathcal{A}_\mathrm{y}$ are the ones that actually set the data instance as belonging to one of the C classes in the data set. Each classification action, defined as $\mathcal{A}_\mathrm{y} = \{\mathrm{c}_1, \mathrm{c}_2, \ldots, \mathrm{c}_C\}$, is attached to one class in the data set. They terminate the episode since the agent's objective is to classify the data correctly, serving as the last step of the episode. To implement that, the full action vector is comprised of both sets $\mathcal{A}_\mathrm{m}$ and $\mathcal{A}_\mathrm{y}$, reaching a total size of $N + C$ possible actions, where actions in the first N elements are measuring actions, and actions in the $[N + 1, N + C]$ range are classification actions. Figure 3.1 visually describes the action vector.

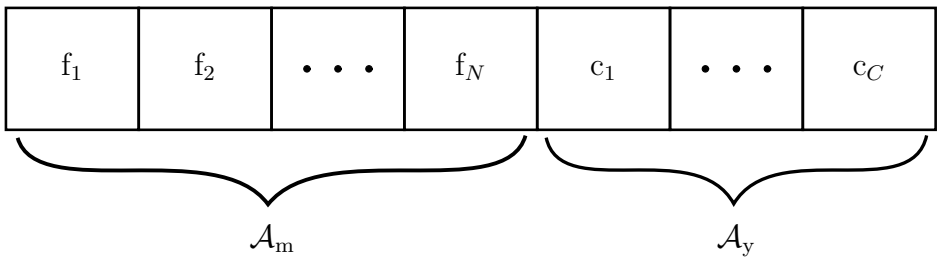


Figure 3.1: Visual description of the action vector. In each timestep, only one element can be chosen. measurement actions are represented by f, indicating the next information to be measured. Classification actions are represented by c, indicating the data instance label, predicted by the model.

The transitions can be formalized as

$$s_{t+1,n} = \begin{cases} x_n & , \text{if } a_t = \mathrm{f}_n \\ s_{t,n} & , \text{otherwise} \end{cases}. \tag{3-1}$$

where the classification actions lead to the terminal absorbing state, and t is the current timestep.

As mentioned before, in the CwCF problem each feature has an intrinsic cost. In this setup, the agent is free to 'measure' any of the available features in the data or classify them with the available information gathered so far. Each measuring step returns a cost, or negative reward, to the agent, relative to the chosen feature, whereas classification actions receive a high negative reward if the class was chosen wrong or a symmetric positive reward if the class was chosen correctly. For instance, in a healthcare scenario, these rewards might depend on the disease itself, and many other factors. It even may be necessary to have higher costs for wrong classifications if the disease is very aggressive, i.e. there could be more nuances to the rewards in order to depict better the problem at hand.

Without a lack of generality, the costs of all features are fixed with the same value $\mathbf{c}_n = -3$, and the reward for right and wrong classifications are also fixed, at 100 and -100, respectively. Note that in a real-world context, the costs associated with features are inherent to the features themselves rather than arbitrarily assigned. Higher costs for a feature may discourage the model from selecting it, as this could lead to a lower final reward. However, during training, that same feature might prove to be crucial for classification, potentially being favored over others despite its higher cost.

It is also important to note that the change for different individual costs would not bring any new perspective on how the proposed algorithm works, as the process would be maintained, only considering the new costs. Due to simplicity and increased ease of understanding, fixed costs were the ones implemented. Following that, the reward function is defined as

$$R(s,a) = \begin{cases} -\mathbf{c}_n & , \text{if } a \in \mathcal{A}_\text{m} \\ 100 & , \text{if } a \in \mathcal{A}_\text{y} \wedge a = y \\ -100 & , \text{otherwise} \end{cases} . \tag{3-2}$$

The fixed values are somewhat arbitrary, but note that they are scale-invariant. As such, an implementation where the classification rewards are unit-sized but the feature costs are set to $c_n = 0.03$ would have the same effect. However, it is very important to notice how delicate and subjective the costs for features and for classification could be, depending on a variety of factors. For instance, it is very straightforward to set a cost for features if it is representing time or money. A different scenario arises if the classification costs are supposed to reflect the patient's well-being.

Another important point about the proposed setup is that all the

environments achieve the Markov Property, that is, the next state only depends on the previous one (current): $s_{t+1} = f(s_t, a_t)$. The distribution from which the states are drawn is static, being, in the first step, the distribution of s_n taken randomly from $\{x_n | x \in \mathcal{X}\}$ if $a_1 = \mathrm{f}_n$. For subsequent steps, the distributions are conditioned on the features which have already been measured. From the agent's point of view, there is no difference between choosing x_n at the beginning of the episode, and drawing the new measurement from all samples that are consistent with the already measured features.

3.3 Implementation

To optimize the described MDP, the Dueling DDQN (DDDQN) was used. This implements all RL-related improvements covered in the previous chapter. As for the architecture, following [33], the neural networks are implemented with two hidden layers with 400 and 300 nodes respectively. The output layer is set according to the number of features and classes for each dataset, in such a way that the number of output nodes is $N + C$, the same as the number of possible actions. The Mean Squared Error (MSE) loss function was used, the hyperparameters mainly followed [33] and were manually tuned, leading to a learning rate ranged from 0.005 to $5e-7$ in an exponential decay, respectively from the beginning to the end of training, and Adam [30] was the optimizer algorithm used.

In this implementation, the targets follow Equation 2-21 for non-terminal states, and as there are no future rewards to discount for, in terminal states only the reward remains. Let y be the DDQN target vector and y_i its elements. The actions corresponding to a certain element y_i of the feature vector are given by

$$a_i = \begin{cases} \mathrm{f}_i \in \mathcal{A}_m, & \text{if } i \leq N \\ \mathrm{c}_{i-N} \in \mathcal{A}_y, & \text{otherwise} \end{cases}.$$

The target vector then becomes

$$y_{t,i} \leftarrow \begin{cases} Y_t^{\mathrm{DDQN}}, & \text{if } a_t = a_i \in \mathcal{A}_{\mathrm{m}} \\ R(s_t, a_t), & \text{if } a_t = a_i \in \mathcal{A}_{\mathrm{y}} \\ Q(s_t, a_i), & \text{otherwise} \end{cases} \tag{3-3}$$

As the exploratory policy, the ϵ-greedy exploration schedule was chosen. It follows Equation 2-12, and was implemented with an exponential decay on ϵ, depending on how many training episodes the model will train for, in such a way that, once the algorithm reaches half the maximum number of episodes,

the exploration is fixed to a low level of $\epsilon = 0.1$, i.e. *exploiting* the best policy so far 90% of the time and *exploring* 10%.

3.4 Embedded Supervised Learning

Considering a complete tabular data set, with no missing values, all information about a data instance is available from the beginning of the episode. Usually, with such information, supervised learning models are more suited for classification. However, if one is trying to use the least amount of information possible to achieve good accuracy in classification, these methods are often designed to use all features at once. Another workaround could be to use feature-selection methods to limit the usage of features with a low impact on performance. This usually is a good way to handle this limitation. However, this is a global measure; specific data instances might be harder to classify, requiring more information (features) than the ones chosen after feature selection, or easier, requiring less information.

Using Reinforcement Learning in such a scenario has the benefit of treating each instance differently, acquiring more or less information as needed. A trivial approach would be to model the MDP as stated in the previous subsection and train the agent to solve it. However, that would disregard the fact that the labels are already available since the beginning of the episode.

To leverage the already-known information about the data class, a new approach is implemented in this version of DDDQN. Similar to supervised learning, in all episode steps, the target estimates related to classification actions, y_i for $i > N$, are updated. For these actions, as they lead to a terminal state, $Q(s', \cdot) = 0$ and only the reward needs to be considered for the updates, following Equation 2-21. The target update $y_{t,i} \leftarrow R(s_t, a_y)$ is carried out for all classification actions, together with the update for the measuring action chosen in the current step. The idea is that this new approach helps the DDDQN algorithm to learn not only the best features (in a dynamic fashion) based on the values measured so far, but also to learn the right classification for all possible states. Doing so guarantees that all Q-values for classification actions are updated based on all visited states. This is possible because, as the input comes from a fixed data set, all information regarding the sample is already known, similar to supervised learning, but still benefiting from the sequential nature of reinforcement learning. In other words, in every timestep during training, the Q-values for the classification actions are going to be updated, following

$$y_{t,i} \leftarrow \begin{cases} Y_t^{\text{DDQN}}, & \text{if } a_i \in \mathcal{A}_{\text{m}} = a_t \\ R(s_t, a_i), & \text{if } a_i \in \mathcal{A}_{\text{y}} \\ Q(s_t, a_i), & \text{otherwise} \end{cases} . \tag{3-4}$$

Normally it would be impossible to update the value of actions different from the chosen action a_t. In our case, the immediate reward for all classification actions can be calculated, as the class of the instance used for the current episode is known. As classification is also a terminal absorbing state, no further information about the transition or reward functions is necessary.

With that approach, the DRL agent is free to learn the best policy for measurement actions, as usual, benefiting from the exploratory policy, while learning the state labels in every episode. The difference between a usual RL algorithm applied to the case of CwCF and the proposed implementation is shown in Figure 3.2. We can see how, for a normal DRL algorithm, only the Q-value corresponding to the chosen action is updated. Opposed, in the proposed DDDQN, the Q-values related to classification actions are updated along with the Q-value corresponding to the chosen measurement action.

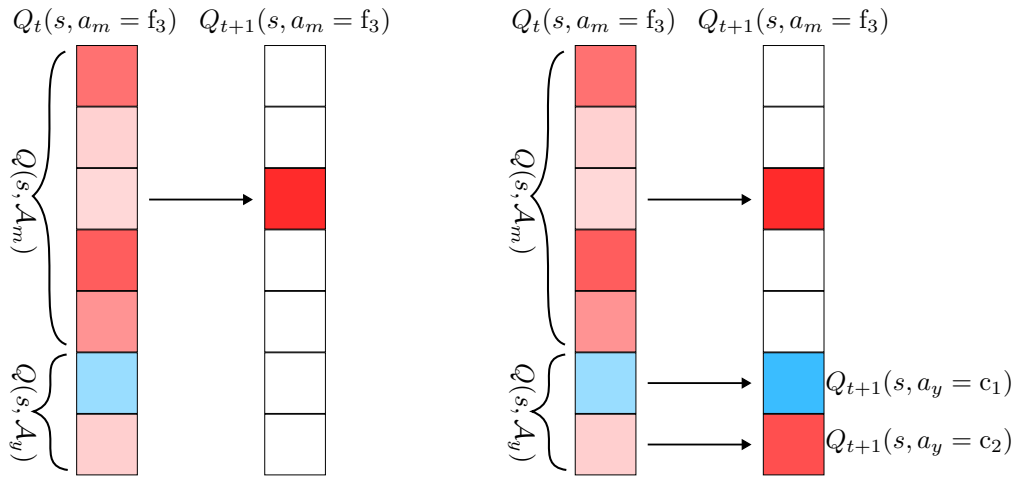


Figure 3.2: Differences between a usual DDDQN implementation (left) and the proposed method (right). Note that in the proposed one, besides the Q-value related to the chosen action, the Q-values related to classification actions are also updated. This happens in all steps during training.

3.4.1 All-action updates

As discussed, in a conventional MDP setup only one Q-value can be updated, as the only future state known is the one returned from the environment, given an action. Again, in our scenario, as all information about the data

is known, the transition function is trivial given the setup. Hence, the future states s_{t+1}, needed for the target updates, can be calculated at any timestep.

With that in mind, the "all-action updates" version is implemented, leveraging the already known information to apply Q-value updates for all available actions at every learning step. This leads to a more computationally complex but sample-efficient training. After experiments, this achieved better results, and is considered the main approach in this chapter.

Along with that, one can argue that this approach decreases the impact of the exploration/exploitation paradigm over the training. This is probably true, since the model no longer relies only on the policy used to update Q-values. That said, the policy schedule is still important to build a diverse set of transitions at the beginning of the training, while focusing on what is working best at the end. Notably the states being visited are still determined by the policy, even though the updated values are not. Figure 3.3 exemplifies the "all-action updates" approach.

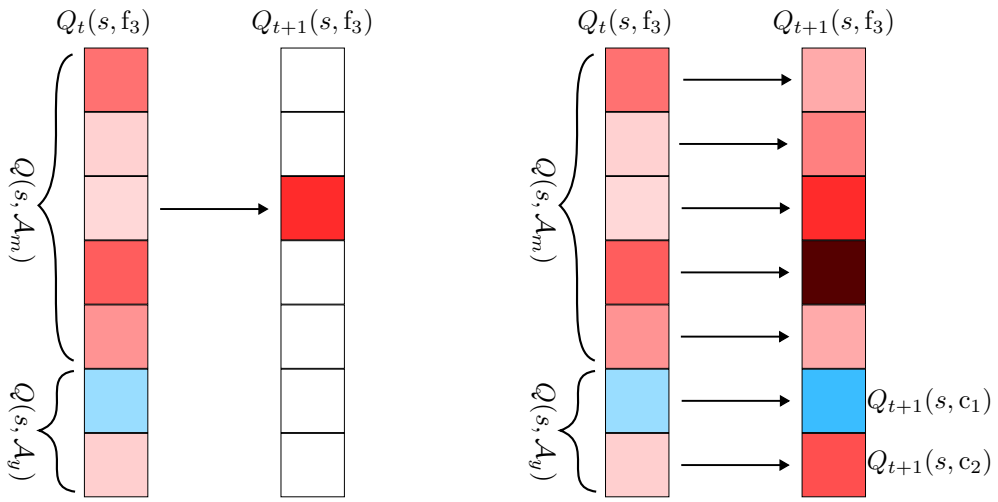


Figure 3.3: Differences between a usual DDDQN implementation (left) and the "all-actions updates" method (right). With this approach, all Q-values are updated, not only the ones related to chosen actions and classification actions.

3.5 Training

During training, different hyperparameter values and setups were tested to achieve good performance and speed up learning for all datasets. The objective is to use the same parameters throughout all dataset experiments, showcasing how DRL algorithms are flexible and robust for different datasets. Manual tuning was applied on, with initial hyperparameters values based on

the original DQN. The same manual tuning strategy was applied to the baseline method, to maintain a fair comparison between models.

The *target network* update interval, for instance, was set to 1500 episodes for all datasets. This was chosen after some empirical tests and carried out for all experiments. Also, three network training steps are done after each episode is finished, again for all datasets.

Some modification to the regular RL training procedure were also implemented as part of the experiments, aiming to improve final performance, both for accuracy and accumulated return. These implementation options are described next.

3.5.1 Training Options

Probably the most important implementation in training was the option to limit actions to the ones that have not yet been chosen in the current episode. In this setup, the agent is not allowed to acquire a feature already measured. Once the feature is measured in the current episode, it becomes unavailable to the agent. This is not usually seen in RL/DRL applications but has been increasingly used, such as in [49, 50], recent DRL breakthroughs for game play. This helps as measuring the same feature would lead the agent to the same state it currently is while not gathering any new information from it. It is always a sub-optimal action. Experiment performances improved when this option was used. In Figure 3.4 we can see an example of an episode, where, after taking an action, it is not possible anymore for the agent to repeat said action. Also, in each timestep since the beginning of the episode, there is the option for the agent to classify the data, terminating the episode, however, usually, it must measure some features before the classification in order to have an increased chance of success.

Another implementation was used regarding actions. In this one, the agent is not allowed to take any classification action a_y during exploration based on the ϵ-greedy strategy. As the classification actions are always 'clamped' during training, i.e. they are updated in every timestep, there is no need for them to be explored. To take those actions during exploration would mean a wasted update for a timestep. Implementing that limitation leads to a more efficient exploration of the state space, forcing the agent to go through as many different states as possible. On the other hand, for *exploitation*, classification actions are allowed. As they terminate the episode, it allows the agent to stop accumulating negative rewards by reading other features, i.e. classifying in advance, once enough information is accumulated and the best

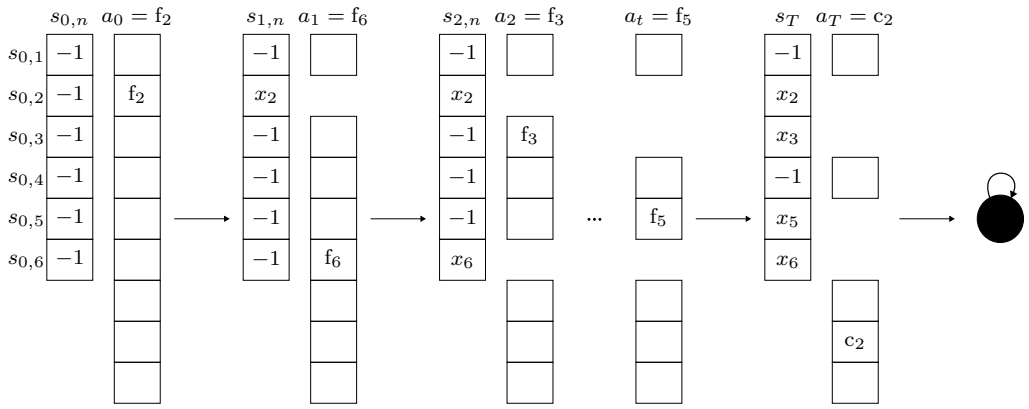


Figure 3.4: Visual representation of states changing according to the chosen action. The last action, a classification action, leads to the terminal absorbing state.

action is the classification one.

For comparison, a second exploration schedule was implemented. The softmax exploration was tested in different experiments. Using it led to slightly better performance for two datasets, but worse performance on others. To ensure that all datasets had the same setup, the ϵ-greedy method was applied to all of them.

3.6 Experiments

In this section, the details regarding the setup for each experiment will be covered, along with the baseline and proposed DDDQN agent performances.

Each experiment on a dataset comprises forty different runs, with different training, validation, and test splits. The number of episodes used is related to the dataset size. For all but one dataset, the amount of training episodes used is $30k$. The biggest dataset "Miniboone", required more episodes to find a representative policy (convergence), therefore $200k$ training episodes were used. Due to this increased amount of training episodes, the exploration decay and learning rate were changed accordingly.

The performances shown in this chapter are the average of the mentioned runs. It is also important to note that, for all experiments, the cost of all features is set to a fixed value of $\mathbf{c}_n = -3$. This value will ultimately influence how many features the algorithms may choose for the classification, affecting both the proposed method and the baselines in a similar way. It is important to clarify that the fixed cost is an example cost, and should not matter for comparison between models. In a real scenario, these values should reflect the actual considered cost for each specific information retrieval.

All experiments conducted for this thesis were performed on a Linux platform with the following specifications: Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz, 256GB RAM, and a GTX 1080 TI. The programming language used was Python 3.7.9. TensorFlow 2.3.0 was employed for the experiments in Chapter 3, while PyTorch 1.12.0 was utilized for the experiments in Chapters 4 and 5.

3.6.1 Baseline

To provide some comparable results and assess the proposed method's performance, a baseline method was developed. The method is composed of Recursive Feature Elimination (RFE) [51] as its feature-selection algorithm with both Logistic Regression and Linear Discriminant Analysis (LDA) as the related classifiers. To ultimately classify the data, an ANN was used.

RFE [51] is a feature-selection *wrapper* method. As the name suggests, it wraps around the learning algorithm, using the prediction performance as an evaluation criterion. This method aims to, recursively, select smaller feature sets based on each feature's importance, which is derived from the learning algorithm (a classifier in this approach). First, the classifier is fit to the training data, using all features. Then, based on the classifier performance, RFE eliminates the least important feature. These steps are repeated until the determined number of features remains. Note that, during the process, RFE naturally ranks the features, so, if one sets RFE to keep only one feature, it would return the importance of all features.

Logistic Regression [52] is a well-known classification algorithm that uses statistical learning, leveraging the logistic function, to predict a class for a set of continuous features. It is primarily used for binary cases, but can be used for multi-class problems as well. The LDA classifier[52] is yet another very known statistical learning method, which leverages prior knowledge of the class probability distribution to allocate data to a class.

The baseline method is implemented in two steps: feature selection and classification. In the first step, as mentioned, the RFE method is used. It wraps around two classification methods, the Logistic Regression and the LDA. The two feature importance ranks are extracted and used in the next step. Now, with both ranks in hand, the ANN classifier comes in place. In fact, for each ranking, a different ANN model is trained, from all to one feature. During training, the validation datasets are used to apply early stopping. Lastly, the models that achieved the highest performance on validation data per number of features are evaluated on the test set. Therefore, after training, there is one

model saved per number of features.

The performance metric used to compare the baseline method and the proposed DDDQN implementation had to be custom-made, as the problem's main objective is not only to achieve good accuracy but also to achieve low cost, which can be more simply understood as high returns. Consequently, the accuracy alone would not be appropriate to be used for comparison between the baseline and the DDDQN agent. To address that, an equivalent reward function was defined, following

$$E(F, acc) = -\mathbf{c} * F + acc * 100 - (1 - acc) * 100, \tag{3-5}$$

where E is the equivalent reward, F is the number of features used, acc is the model accuracy and $\mathbf{c}$ is the fixed cost for all features. Note that, if one intends to implement different costs for different features, trivially, it is only needed to change the first factor $-\mathbf{c} * F$ to $\sum_{\mathrm{f}=1}^{F} -\mathbf{c}_{\mathrm{f}}$. This equation translates the performance for the baseline method to the corresponding value returned from the DDDQN agent, accounting for accuracy and the costs of each feature used.

The architecture used for the ANN classifier is the same as used in the DRL agent, with two hidden layers with 400 and 300 nodes, respectively. The difference is only in the output layer, which has only C instead of $N + C$ nodes and the loss used is the Categorical Cross Entropy. The full algorithm can be seen in Algorithm 1.

3.7 Results

In this section, a deeper look at the proposed method is made using a synthetic dataset. Then, the results of subsequent experiments are shown for the validation data. Lastly, the test set results are given. All results shown in this section are regarded as the "all-action updates" version of the model.

It is important to note that, for the baseline model, the performance presented is the one that achieves the highest return on the validation dataset, considering all possible numbers of features based on RFE. This ensures that the chosen baseline represents the optimal trade-off between the number of features used and the resulting model performance. By selecting this optimal point, we can ensure that the baseline reflects the best achievable performance for the given dataset.

Algorithm 1 NN baseline classifier algorithm

Input: dataset D = $\{(x_0, y_0), (x_1, y_1), ..., (x_K, y_K)\}$
Split D into training, validation and test sets
Execute Logistic Regression ranking with RFE
Execute LDA ranking with RFE
for i=1 **to** F **do**
    Randomly Initialize parameters θ
    Train NN with i features (following Logistic Regression ranking).
    Save performance on validation set
    Save model
end for
for i=1 **to** F **do**
    Randomly Initialize parameters θ
    Train NN with i features (following LDA ranking).
    Save performance on validation set
    Save model
end for
for i=1 **to** F **do**
    Choose best model for i used based on validation set
    Save test performance
end for

3.7.1 Verification

To verify that the proposed method works as expected, a synthetic dataset was created to serve as a proof of concept. It has two features and two classes, each of which is generated by a normal distribution with different covariance matrices, graphically shown in Figure.3.5.

The data set was constructed such that, depending on the value of the first measurement, a second measurement is either necessary or not. For example, if feature 0 is measured first, and the value is above 30, we can be sure that the class is 1. However, if it is around 20, we need to measure the feature 1 to know the class for sure.

Figure 3.6 shows the trained DRL agent policy, in different states of the dataset. The x-axis represents feature 0, and the y-axis represents feature 1. The left column represents states where feature zero is not yet measured, and similarly, the bottom row represents states with feature one not yet measured. Through the graph, it is clear how the DRL agent behaves in different states. Even with less information (i.e. only one feature measured), depending on the value already measured, the agent already classifies. This shows that the agent reads only enough features to get a high 'confidence' (i.e. high Q-value) for classification actions. Note that this might hurt accuracy at some level while improving the cost for each classification.

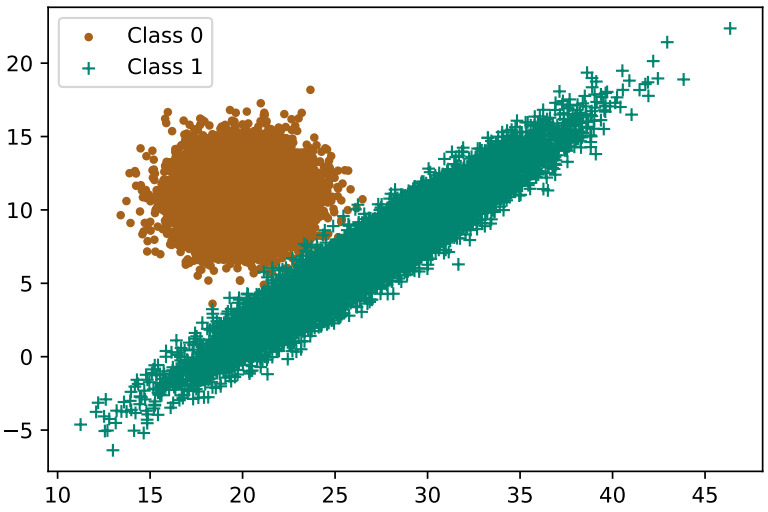


Figure 3.5: Visual representation of the Synthetic dataset, composed of two features and two classes. The x-axis represents feature 0, and the y-axis represents feature 1.

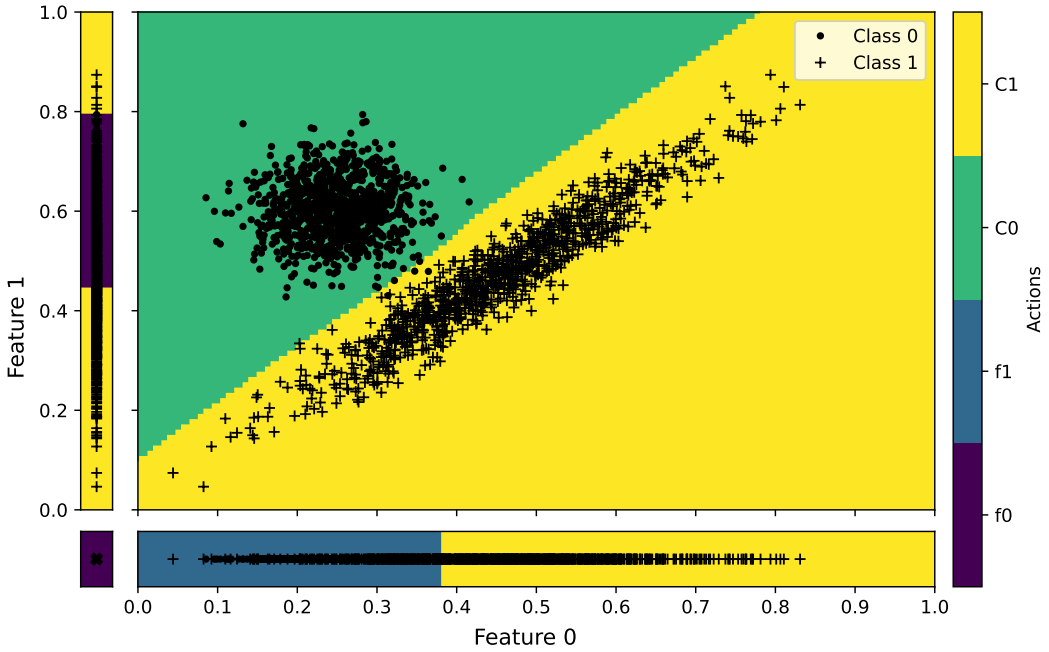


Figure 3.6: DRL policy on 'Synthetic' dataset. Actions M0 and M1 are measurement actions for the respective features. C0 and C1 are classification actions for the respective classes. In regions with mixed classes, the model chooses to measure another feature, while in regions with one majority class, the model classifies early.

Figure 3.7 shows the maximum Q-value achieved by the agent. We can interpret this graph as regions of more or less 'confidence' regarding what the agent thinks it is supposed to do. It is clear that a 'less confident' region arises between both classes, behaving like a decision boundary.

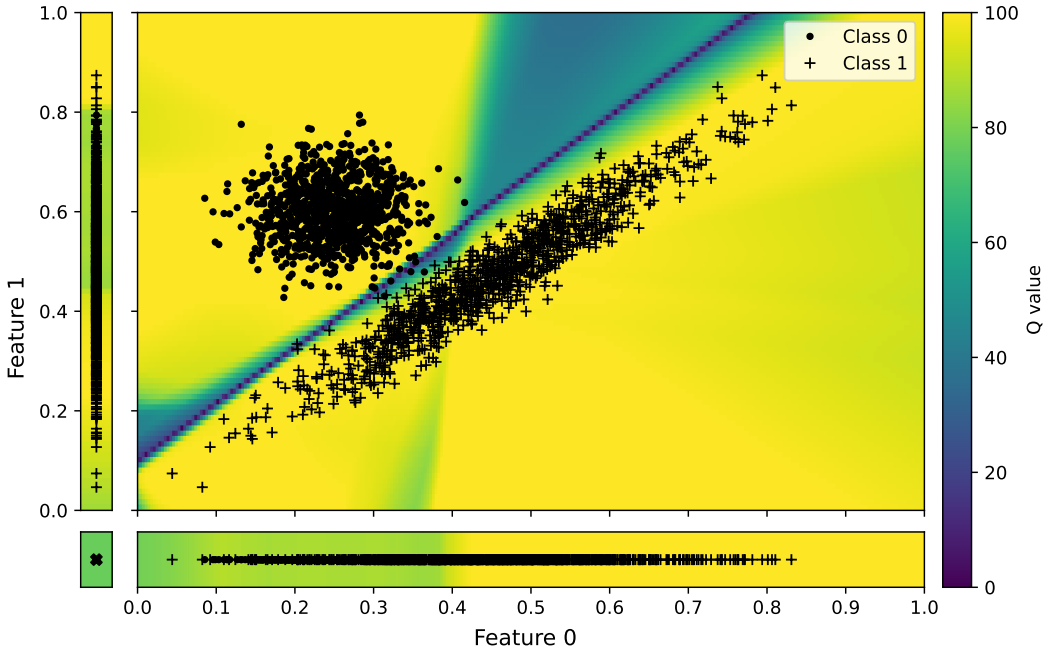


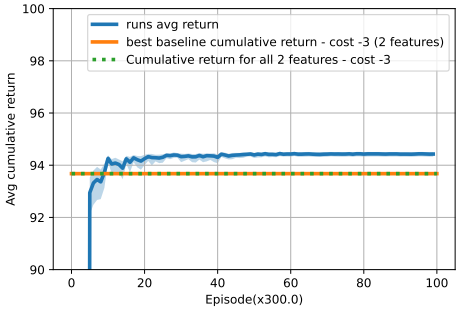
Figure 3.7: Agent maximum values of Q. This can be interpreted as the model's "confidence". Note how there is a region of "low confidence" exactly between the two classes distribution.

3.7.2 Learning curves

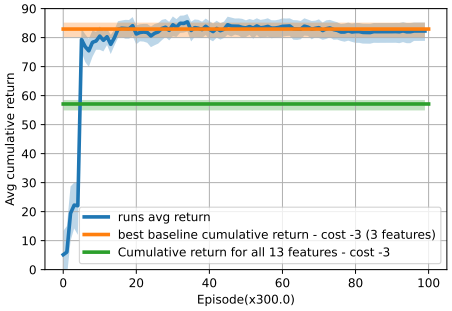
In Figure 3.8, the accumulated return plots for all six experiments are shown. In these graphs, the x-axis represents all validation set performances through the training time, while the y-axis represents the mean cumulative return. The proposed method had a clear better performance in four datasets, while achieving a similar performance in two. Moreover, the DDDQN agent holds a flexible and dynamic way to classify data instances, gathering more information only as needed, and classifying 'on the fly'.

Synthetic dataset

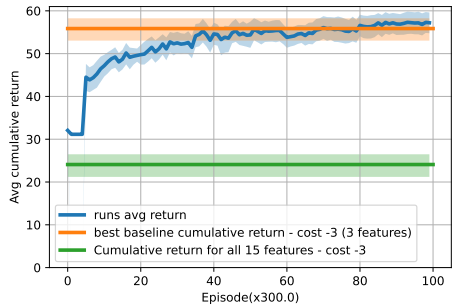
The performance of the proposed method on the synthetic data set introduced in Section 3.7.1 is given in Figure 3.8a. The DDDQN agent



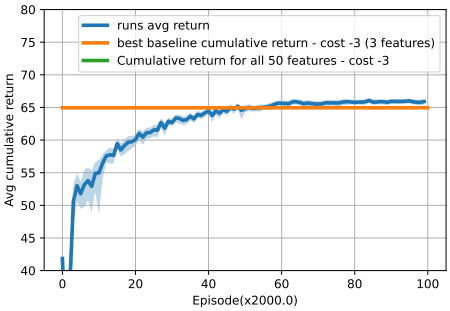
(a) Average cumulative returns on Synthetic dataset



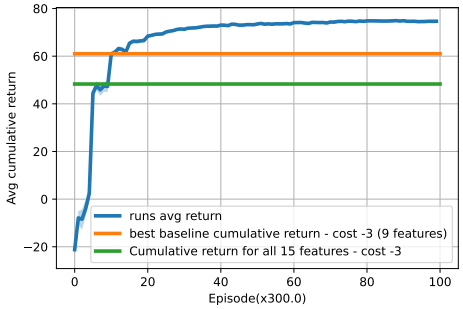
(b) Average cumulative returns on 'Wine' dataset



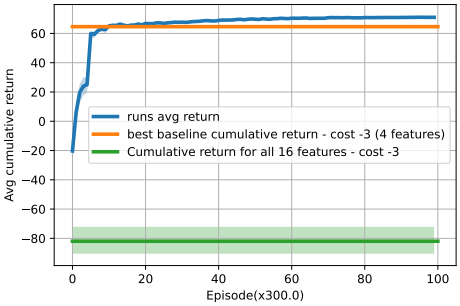
(c) Average cumulative returns on 'Healthrisk' dataset



(d) Average cumulative returns on 'Miniboone' dataset



(e) Average cumulative returns on 'Pen Digits' dataset



(f) Average cumulative returns on 'Beans' dataset

Figure 3.8: Performances on validation data

outperforms the baseline in average return. For accuracy, the baseline performs better by a small amount (see also Section 3.7.3).

Wine dataset

In this dataset, the performances were comparable between the baseline and the DDDQN agent. In Figure 3.8b we can see how average returns in the validation set are very similar between the proposed method and the baseline. The wider overlapping confidence interval shows that the outcome is less stable. In this data, as both accumulated return and accuracy have very close performance, with overlapping confidence intervals for the baseline and the DDDQN agent, it can be considered a tie.

Healthrisk dataset

As for the Healthrisk dataset, again, both methods have similar performance for accumulated returns and accuracy. In Figure 3.8c it can be seen the returns performances. For accuracy, both models achieved similar results, and the confidence intervals are, as in the previous experiment, overlapping.

Miniboone dataset

In this dataset, the proposed model achieved a better performance in accumulated return, while the baseline achieved better results accuracy-wise. The return plot during training can be seen in Figure 3.8d. Due to the bigger number of samples, confidence intervals were small and not overlapping, indicating a better statistical result.

The original proposed model (without the "all-action updates") performed poorly against the baseline in this setup, and since it is a dataset with more features, it is likely that the model took advantage of the more frequent updates.

Pen Digits dataset

The DRL agent performed considerably better than the baseline in this experiment, both for accuracy and accumulated return, as shown in Figure 3.8e. We hypothesize that this particular dataset has some characteristic that is suited for the sequential acquisition of features, achieving better performance than the others.

Beans dataset

Likewise the previous experiment, the proposed method also achieved better performance both for accuracy and accumulated return. Results are shown in Figure 3.8f.

3.7.3 Performances on test data

The performances on test data are showed in Tables 3.1 and 3.2. It can be seen that regarding the returns, the proposed method clearly outperforms the baseline in four datasets ('Synthetic', 'Miniboone', Pen Digits' and 'Beans'). For the other two, the returns performances are very similar, with confidence intervals overlapping ('Wine' and 'Healthrisk'). Regarding accuracy, as expected, the DDDQN agent performance is slightly below the baseline in five datasets, sometimes with overlapping confidence intervals ('Wine' and 'Healthrisk'), and better in 'Beans' data.

Even with such close comparison, note that the proposed algorithm has the advantage of dynamically choosing features, classifying 'on the fly' and therefore adding more flexibility to the model. Moreover, for most data instances, the proposed method already classifies after one or two feature measurements, while the baseline always needs all chosen features after feature-selection is carried out. Lastly, it is important to stress again that the baselines were trained using all possible number of features based on RFE, and the model with the best return was selected. This extensive training and selection process is not necessary in the proposed DRL approach.

Table 3.1: Cumulative Return Performances on test data

	Cumulative Average Returns		
	All features Baseline	Baseline	DRL
Synthetic	$93.68\ \binom{93.8}{93.6}$	$93.68\ \binom{93.72}{93.62}$	$\mathbf{94.52}\ \binom{\mathbf{94.58}}{\mathbf{94.44}}$
Wine	$57.11\ \binom{58.4}{54.8}$	$\mathbf{82.94}\ \binom{\mathbf{85.17}}{\mathbf{80.17}}$	$\mathbf{81.51}\ \binom{\mathbf{83.93}}{\mathbf{78.73}}$
Healthrisk	$24.09\ \binom{26.4}{21.2}$	$\mathbf{55.86}\ \binom{\mathbf{58.25}}{\mathbf{53.04}}$	$\mathbf{57.05}\ \binom{\mathbf{59.26}}{\mathbf{54.87}}$
Miniboone	$-60.55\ \binom{-60.4}{-60.8}$	$64.95\ \binom{65.12}{64.77}$	$\mathbf{66.38}\ \binom{\mathbf{66.70}}{\mathbf{66.05}}$
Pen Digits	$48.31\ \binom{48.4}{48.2}$	$61.02\ \binom{61.33}{60.64}$	$\mathbf{66.94}\ \binom{\mathbf{67.51}}{\mathbf{66.36}}$
Beans	$36.85\ \binom{37.3}{36.4}$	$64.63\ \binom{65.07}{64.18}$	$\mathbf{71.12}\ \binom{\mathbf{71.56}}{\mathbf{70.75}}$

Table 3.2: Accuracy Performances on test data

	Average Accuracy		
	All features Baseline	Baseline	DRL
Synthetic	$0.998\,(^{0.999}_{0.998})$	$\mathbf{0.998}\,(^{\mathbf{0.999}}_{\mathbf{0.998}})$	$0.997\,(^{0.998}_{0.997})$
Wine	$0.981\,(^{0.987}_{0.969})$	$\mathbf{0.960}\,(^{\mathbf{0.971}}_{\mathbf{0.946}})$	$0.939\,(^{0.951}_{0.922})$
Healthrisk	$0.845\,(^{0.857}_{0.831})$	$\mathbf{0.824}\,(^{\mathbf{0.836}}_{\mathbf{0.810}})$	$\mathbf{0.834}\,(^{\mathbf{0.845}}_{\mathbf{0.823}})$
Miniboone	$0.947\,(^{0.948}_{0.946})$	$\mathbf{0.870}\,(^{\mathbf{0.871}}_{\mathbf{0.869}})$	$0.863\,(^{0.865}_{0.860})$
Pen Digits	$0.967\,(^{0.967}_{0.966})$	$\mathbf{0.940}\,(^{\mathbf{0.942}}_{\mathbf{0.938}})$	$0.922\,(^{0.926}_{0.919})$
Beans	$0.924\,(^{0.927}_{0.922})$	$0.883\,(^{0.885}_{0.881})$	$\mathbf{0.908}\,(^{\mathbf{0.911}}_{\mathbf{0.906}})$

3.7.4 Ablation

An ablation study was conducted to better understand the impact of each component implemented in the proposed algorithm. By selectively removing different parts of the model, we can identify which elements have the greatest influence on the overall performance. Table 3.3 shows how excluding certain parts of the proposed methodology affects the final outcome, highlighting that the combination of the proposed changes was crucial to achieving the highest performance. The ablation study includes the proposed algorithm (using the All-action updates variation, no repeated actions, and no classification during exploation), and variations without the proposed training details, such as repeated actions allowed (RA), any actions allowed during exploration (AA), and single update (SU). Results are the average return considering all datasets mentioned in the experiments.

Table 3.3: Ablation study, showing average return values from all datasets for different algorithm configurations. Different configurations are: Proposed (All-action update, no repeated actions, no classification during exploration), SU (single update), RA (repeated actions allowed), AA (any action allowed during exploration - including classification)

Ablation	Avg. return	CI ↓	CI ↑
Proposed	**72.92**	**71.7**	**73.92**
Proposed + RA	59.73	57.08	62.25
Proposed + AA	64.53	63.17	65.69
SU + AA	48.29	46.31	50.20
SU + RA + AA	29.51	25.56	33.13

4
Using transformers to classify tabular data with missing values

This chapter will focus on the transformer approach as a classifier for tabular data with missing features during inference. This serves as an intermediate step between the previous and next chapter, where we use transformers in DQN-based RL for classification with costly features. Here, we validate the performance of a transformer used as a classifier when we do not have all features available at test time. This serves as a proof of concept that transformers are viable in modelling tabular data and dealing with missing values.

4.1
Related Work

In this section, we review papers that address the use of transformers for classification tasks. We connect this topic with the problem of missing values, a common issue in real-world environments and datasets.

Aditionally, we also point to papers advocating for tree-based methods, such as XGBoost[15], over DL and transformers models, in handling tabular data. Tree-based methods are one of the best for this type of data. However it is important to note that they are usually not assessed directly in terms of their ability to handle missing values.

Specifically for this problem, imputation methods are often used to fill in the missing data. One of the most reliable imputation methods today is the Multiple Imputation by Chained Equations (MICE) [16, 17] approach, though other methods are also regarded at the same level, such as Missforest [88, 95], Hot-deck [89], and MIDAS [90] .

4.1.1
Classifying with Transformers

It is only natural that with the excellent performance of transformers in machine translation [36], text generation [67] and other applications, this kind of model would be studied in other environments. For classification, as mentioned, there is not a consensus about the best model to approach a tabular dataset, as some papers point to tree-based methods as the better option, while

others point to transformers having better performance [53, 54, 55, 56, 57, 59]. In this section, we point to the papers advocating to transformers, mentioning their general ideas, peculiarities, strengths and weaknesses.

4.1.1.1 TabTransformer

The TabTransformer [53] is an end-to-end model based on the original transformer [36]. The overall architecture is mainly characterized by the column embedding layer, transformer layers and an MLP in the end. Architecture-wise, the main contribution is the column embedding layer, which generates the contextual embeddings of categorical features.

In practice, the contextual embeddings are the response of each transformer layer to the input feature embeddings. In other words, each categorical feature is parametrically embedded, and these embeddings are fed to the transformer layers. The output of each transformer layer is considered a contextual embedding. The calculated contextual embeddings are concatenated with the normalized continuous features, and go through the last layer of the model, the MLP. Figure 4.1 is extracted from the original paper and shows this architecture in detail.

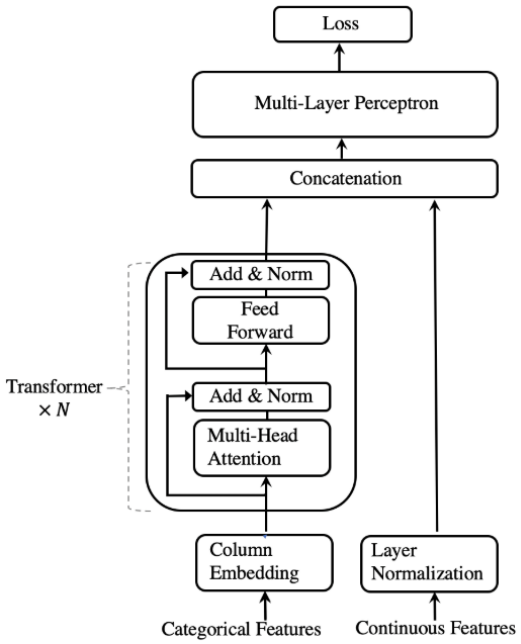


Figure 4.1: Architecture of TabTransformer, extracted from [53]. Continuous features do not go through the transformer layer

The results showed competitive performance with Gradient Boosted Decision Trees (GBDT) and better performance than the baseline MLP. This study also addresses tasks such as semi-supervised learning and robustness

for missing and noisy data. The authors provide two options to deal with missing data, either by learning a representation of the missing label for each categorical feature, or using the average learned embeddings over all labels in each categorical feature. They chose to use the second option, since there was not enough missing values so the model could learn a good representation for it. There was no proposal for dealing with missing values in continuous features.

4.1.1.2 TabNet

The TabNet model [54] is designed to incorporate the interpretability of decision trees while leveraging the power of deep learning. To achieve this, the model's encoder employs individual feature selection by applying a learnable sparse mask based on attention over the preceding decision steps. At each decision step, the model selects a subset of features to reason from. The selected features are processed by a feature transformer, which splits the output into two components, one for the current decision, and another for the subsequent step. The feature transformer comprises two layers that are shared across all decision steps, and two additional layers that are specific to each decision step. One of the objectives is to provide interpretable outputs with the learned masks, that can be aggregated to obtain a global feature importance attribution.

The decoder is simpler and aims to generate the reconstructed features. It comprises a feature transformer for each step. Visual details of the model are shown in Figure 4.2, which shows the encoder, the decoder, the feature transformer, and attentive transformer architectures. Experimental results showed performances as good or better than baselines, such as XGBoost. There was little focus on missing values, which was addressed in a specific task with the objective of predicting them, in a self-supervised learning approach.

4.1.1.3 SAINT

Like the previous approaches, SAINT [55] is an end-to-end transformer-based architecture used to classify or regress tabular data. This model is composed of a stack of identical stages, which in turn are built by one self-attention transformer block and one novel intersample attention block. The architercture is shown in Figure 4.3. Self-attention blocks follow [36] architecture. The intersample blocks are similar, only differing in the use of intersample attention layer instead of self-attention. These layers can be

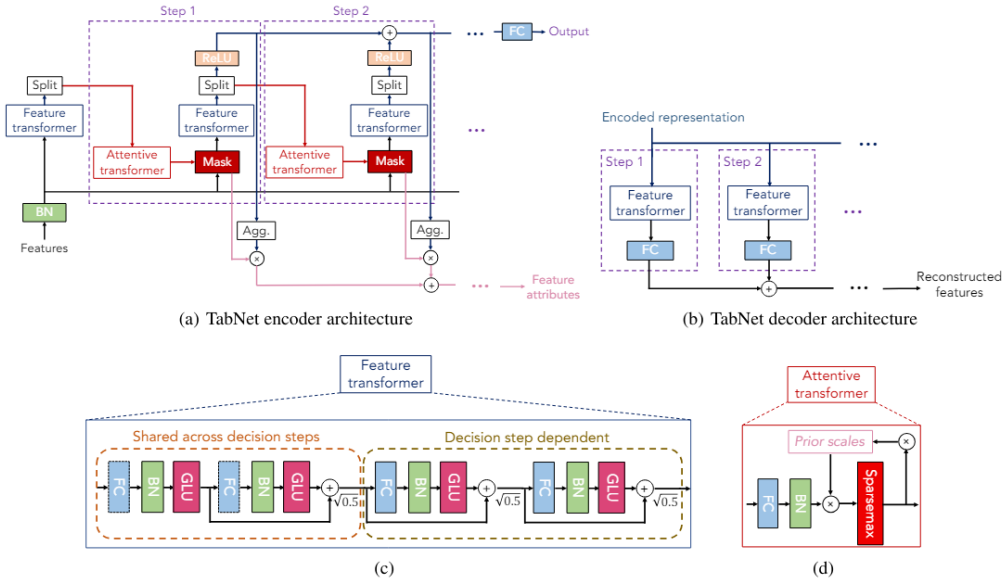


Figure 4.2: Architecture details of TabNet.

thought as row attention, as the attention is computed across all samples in a batch, rather than across features in a single sample. It allows all features from different samples to communicate with each other, which increases performance.

The intersample attention provides benefits in performance, however it can only be leveraged when a batch of inputs is fed to the model, otherwise there is no information from different sources to be shared, othet than the only inputted sample. This can go both ways, sharing similar representations when information is missing, but requiring a batch of inputs in order to make use of this approach.

Results showed overall superior performance of SAINT compared to baselines like TabNet and tree-based methods like XGBoost. The missing data problem is addressed only in the training phase, and no comparison with baselines was carried out on this aspect.

4.1.2 Are transformers better than classical methods?

Despite evidence showing that transformer models can outperform the current state-of-the-art in tabular data classification, the subject remains controversial. Some papers advocate for transformers as capable of outperforming tree-based methods, as we just saw. On the other hand, other studies argue that tree-based methods, particularly gradient-boosted ones, exhibit better performance than deep learning methods, including transformers. Next, we expose three works that point to tree-based methods as better than deep learning to

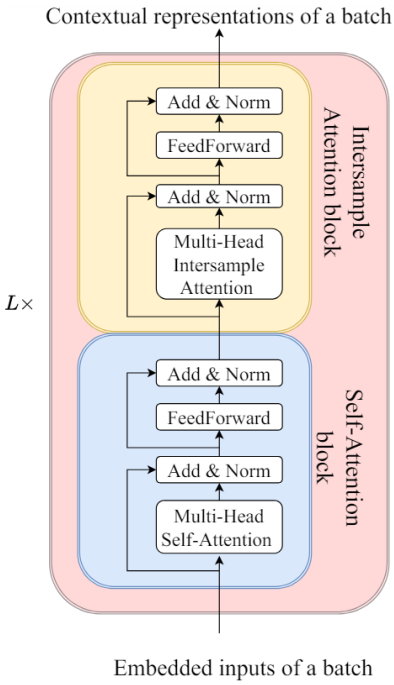


Figure 4.3: SAINT transformer block, showing how it applies the intersample attention.

classify tabular data.

4.1.2.1 Tabular data: deep learning is not all you need

The main objective of this work [56] was to evaluate recently proposed deep learning models, like TabNet, and a well established baseline, such as XGBoost, on tabular data classification and regression.

To assess the performance, several datasets were used as benchmarks, some for classification problems and others for regression problems. The overall result showed that the deep models studied did not outperform XGBoost, besides being more difficult to tune and take more time to train. Interestingly, when using an ensemble of XGBoost and deep models, the best performance was achieved.

4.1.2.2 Why do tree-based models still outperform deep learning on typical tabular data?

In this research [57], the authors provide an extensive and detailed benchmark for tabular datasets, in a variety of setups. With this benchmark they analysed different models both in the deep models realm, such as MLP

and SAINT, and in the tree-based models realm, like XGBoost and Gradient Boosting Tree.

Their findings point to tree-based models being better in the studied tasks, adding to the fact that they are easier to tune and faster to train. The authors also try to answer some questions on why tree-based models are better than deep models for this kind of data. In summary, they found that neural networks tend to overly smooth solutions, uninformative features affect MLP-like models more than tree-based ones, and deep learning models often have rotationally invariant learning procedures [58].

4.1.2.3 Deep Neural Networks and Tabular Data: A Survey

In this survey [59], the authors provide a thorough review of existing literature on deep learning models for tabular data, categorize the existing methods in groups, define state-of-the-arts approaches, as well as an extensive empirical comparison between models on multiple real-world datasets.

The empirical results showed boosted tree methods as the best options amongst the studied models (13 deep learning models and 8 classic machine learning models). This corroborates with the previous pointed papers, and shows that training efficient deep learning models suited for tabular datasets is still an open research problem.

4.1.3 Handling missing values

Missing values are a common problem for data scientists, ML practitioners, and researchers to deal with. This kind of problem can emerge from human error when imputing values, data corruption, or even unanswered questions in a survey. Depending on the type of missing data, one should approach it in different manners. So first we should understand the types of missing values.

4.1.3.1 Types of missing data

There are three types of missing values, each with its own characteristics. Determining the type of missing value in a dataset is important to choose the best approach to handle them. [60, 61, 62]

Missing Completely at Random (MCAR): when data are MCAR, the probability that it is missing does not depend on any observation of the dataset. With this type, no bias is introduced in the data because of missing data. For

example, this can happen due to data corruption in a depression survey. This is the type of missingness we mainly focus on in this chapter.

Missing at Random (MAR): when data are MAR, there is a systematic relation between the missing data and the observed data. For example, males are less inclined to fully fill a depression survey than females. In this case the missingness is directly related to a fully observed information (gender) in a dataset. Moreover, this type of missingness is also present when measured features depend on current data, similar to what is discussed in the previous chapter.

Missing Not at Random (MNAR): missing data is MNAR when there is a direct relation between the missing values and unobserved information in the dataset. For example, a patient with depression filling out a depression severity survey is less likely to completely fill out the survey as their depression is more severe. The severity itself is not observable, since it is the target of the survey.

4.1.3.2 MICE

Handling missing values can be a challenge. How you approach it depends heavily on the type of missing values, along with how much computation you are willing to use.

Missing values can be dealt with using different approaches, however, imputation is still the go-to since it does account for all available information (other approaches may leave instances with missing information out) [86]. Imputation methods replace missing information with values based on some heuristic, such as mean, mode, or more complex approaches, such as linear regression, ML models, and Hot-Deck[89]. Multiple Imputation (MI) is a derivation from the imputation approach, where many imputations are carried out on the whole dataset, and one final result is drawn from that [86]. MI has many approaches, like MIDAS [90] and Missforest [88], and compared to other methods to handle missing values, is the one that provides better performances [87, 88].

One of the most well-regarded approaches in the MI paradigm is the Multiple Imputation by Chained Equations (MICE) [16, 17]. This approach can deal with different data types, such as categorical and continuous features, as well as different missing data types, like MCAR and MAR.

As MICE is a multivariate imputation technique, each missing value is imputed taking into account one or more other features. The process follows the steps: 1. imputation is performed on missing values using a simple strategy, such as mean or median imputation. These are considered placeholders. 2. for

the first feature, these placeholders are regressed by a regression model that uses this feature as the dependent variable, while others are independent. 3. the original placeholders in the first feature are replaced with the predictions from the regression model. 4. steps 2 and 3 are repeated for every other features that has missing values. when all features are handled, one cycle is finished. 5. this process is repeated for a number of cycles. At the end of each cycle, all imputations are updated. The number of cycles is a parameter of the approach.

After this process is finished, one dataset has been created with imputed values. In practice, many imputed datasets are created by repeating the process. These datasets will have the same observed values, but different imputed values. With more datasets created, one can better estimate the error introduced by imputation, however in practice, creating a small amount of imputed datasets, such as 5, is already good enough [63].

4.1.3.3 XGBoost

When dealing with tabular data in general, many papers point to tree-based algorithms as being the best choice [56, 59, 64, 57], specifically models like Extreme Gradient Boosting (XGBoost) [15], CatBoost [91], and LightGBM [92] are well-regarded, especially over deep learning (DL) models. The now well-known XGBoost [15] has been present as one of the state-of-the-art approaches when dealing with tabular data for some time, and is the most frequent in survey works.

XGBoost is a strong machine learning model, based on decision trees, that leverages boosting, an ensemble ML method that trains sequential weak learners. XGBoost starts with a weak tree model, and iteratively, a new tree is added. This new tree is trained to predict the residual error of the model, and its predictions are added to the current predictions. The model is trained using gradient descent. One of the features of XGBoost is that it can naturally handle missing values due to the implemented sparsity-aware split finding [15]. This makes XGBoost a good baseline to assess models' performances on classification with missing values.

4.2 Methodology

This section outlines the implementation details of the proposed model for tabular data classification with missing values, defining the model design and procedures, such as training and pre-processing, used in the experiments.

Motivated by the success of transformer models across various applications, including tabular datasets, we developed a transformer-based model specifically designed to handle this type of data. Additionally, the missing data problem is a frequent reality for those who deal with real datasets, therefore this challenge is also addressed in our setup.

Consider a dataset $\mathcal{D}$, with F features, where each data sample is represented by x_f^j, with j indicating the sample row, and f the feature column. Classes are represented by y, and each sample corresponds to exactly one class, $x \mapsto y$. Feature values can be missing, where x_{miss}^j represents the set of missing features for a specific sample in row j.

The objective of the proposed approach is to correctly predict y, either provided a sample with x_{miss}^j or not. The model applies the attention mechanism to the input feature vector, establishing a communication between all features in the fed context. Missing values x_{miss}, if present, are processed in one of two different ways, either by (1) applying an attention mask to missing values, or (2) trimming the context so that missing values are no longer present, condensing the context only to present features. No approaches in the literature were found to be similar to (2) regarding missing values.

As for the model details, especially those which differ from the original transformer architecture, three main topics can be mentioned: preprocessing, embeddings, and missing values handling.

4.2.1 Preprocessing

Before training, the dataset undergoes a few preprocessing steps, to ensure that the data is better handled by the model. Since all datasets consist of continuous features, we begin by normalization the data, scaling it to the range of $[0, 1]$.

In this chapter we focus on test-time missing features. Since all datasets are originally full, versions of the validation and test data sets were prepared with missing values. In these versions, missing values were inserted by randomly selecting how many features are to be missing, and according to that randomly choosing which features would be missing. This process was applied to each sample in the validation and test data sets. These modified datasets were used to assess the model's performance in the presence of random missing data.

Additionally, to facilitate a comparison of model performance based on the quantity of missing data, versions of the test data were created with specific numbers of missing features, ranging from 0 to $F - 1$. These complement the

previously mentioned data sets and allow for a detailed evaluation of model robustness against varying levels of missing data.

4.2.2 Embeddings

In practice, the proposed model is configured to receive a vector containing all feature values available, or x^j. Missing values of x^j are set to NaN to indicate missingness. In order to extract information of both, feature indexes and their respective values are embedded through a learnable layer.

Since all features are continuous, their values are embedded with a linear layer, mapping one value ($\mathbb{R}^1$) to a vector with the embedding dimension d_{emb}. The respective feature indices are value-encoded from 1 to F, and embedded with a learnable lookup table that maps each encoded feature $\mathbb{N}^1$ to a d_{emb} dimensional vector ($\mathbb{R}^{d_{emb}}$). Both the value and feature embeddings are added together before being fed to the attention blocks.

4.2.3 Handling missing values

Two different implementations of handling missing values were applied. The first relies on attention masks to filter out values that are missing, i.e., inputs that present missing values are used to build a mask, that during attention, is applied to corresponding tokens related to the missing input. This indicates to the attention mechanism which tokens should be ignored, which in practice is zeroing their values in the attention matrix.

Formally, from a batch of inputs with b samples and F features, during the attention mechanism, Equation 2-7 generates a matrix with size $T \times T$, with T usually being the size of the context. In this case this is the full number of features present in the dataset. Therefore, for each sample in the batch, we end up with a matrix of $F \times F$, the first dimension representing the queries, while the second represents the keys. The mask points to missing elements, so each row or column from the generated matrix that corresponds to a missing value is masked out: it is the outer product of the mask vector with itself.

Algorithm 2 details this process and Figure 4.4 visually exemplifies how masking is performed.

The second way of handling missing values relies on contextual trimming, which is excluding all missing values from input context. During training, this approach actively and randomly inserts missing values inside the input context, and trims it later. For each batch, a random number of missing values is chosen and random features of each sample in the batch are replaced by missing values.

Algorithm 2 missing values with mask

Input: batch $\mathcal{B} = \{x^1, x^2, ..., x^b\}$
$\text{missing}_i^j = \begin{cases} 1, & \text{if } x_i^j = \text{NaN} \\ 0, & \text{if } x_i^j \neq \text{NaN} \end{cases}$
$\text{mask}^j = \text{missing}^j \otimes \text{missing}^j$
calculate embedded values with linear layer (input = $\mathcal{B}$)
calculate embedded features with lookup table (input = $\{1, ..., F\}$)
generate tokens adding embedded values and embedded features
perform remaining transformer forward pass with calculated tokens and mask

Algorithm 3 missing values with trimmed context

Input: batch $\mathcal{B} = \{x^1, x^2, ..., x^b\}$
number_of_missing = random(1 to F)
for x in $\mathcal{B}$ **do**
    missing_features = random(1 to F,
                              number_of_missing,
                              replace = False)
    $x_{\text{missing\_features}} = \text{NaN}$
end for
generate $\mathcal{B}_{\text{trim}}$ by trimming the context from ($\mathcal{B}$)
calculate embedded values with linear layer (input = $\mathcal{B}_{\text{trim}}$)
calculate embedded features with lookup table (input = features from$\mathcal{B}_{\text{trim}}$)
generate tokens adding embedded values and embedded features
perform remaining transformer forward pass with calculated tokens and mask

The batch is then trimmed so that only available features are left. This process is repeated for each batch of training data. The data is then fed to embedding layers and later to the attention blocks, which do not need any mask. One advantage is that no unnecessary computations are executed on values that would be masked out. The pseudo-code can be seen in Algorithm 3, and the visual representation is in Figure 4.5

This approach is feasible only because it addresses missing values specifically at test-time, rather than during training. This allows for greater flexibility in utilizing the complete training data, as features can be randomly masked during training without harming performance, and in fact, leading to performance improvements.

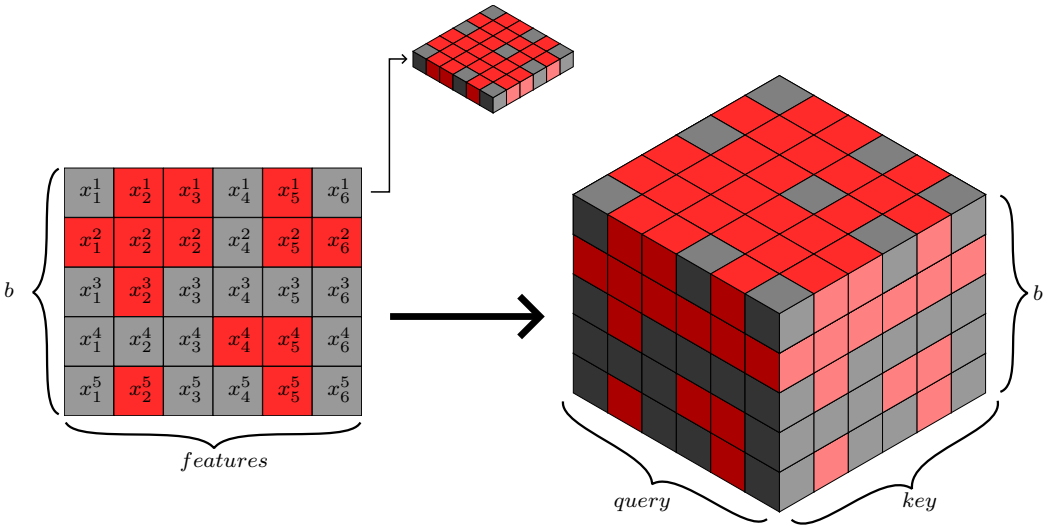


Figure 4.4: Representation of how the masking is created. On the left, we show the raw input, with missing values being masked in red. Each element is represented by a key, query, and value in the attention mechanism, and the communication between keys and queries from elements of the same input row can be represented as a matrix. An example of the matrix generated from the first row is shown in the upper right of the left image. On the right, we show the complete attention mask (one slice for each row in the left image). Note how each missing value is masked both for keys and queries.

4.3 Experiments

In this section, we focus on describing the experimental characteristics of our setup, detailing the datasets, baselines, training implementations, model architecture, and how performance was assessed.

All experimental setups were run 10 times with different seeds, achieving statistical significance across results. Confidence intervals were assessed by bootstrap confidence interval with 95% confidence level, using the bias-corrected and accelerated method [65].

The experiments were conducted on the same set of datasets mentioned throughout this work, excepting the synthetic dataset due to its simplicity.

4.3.1 Baselines

When dealing with tabular datasets, as mentioned, XGBoost is still one of the best models, either for classification or regression. Moreover, it can naturally deal with missing values based on its default direction technique [15]. Therefore, we use XGBoost as one of the baselines. The other two baselines present in this section leverage the MICE imputation method, since it is

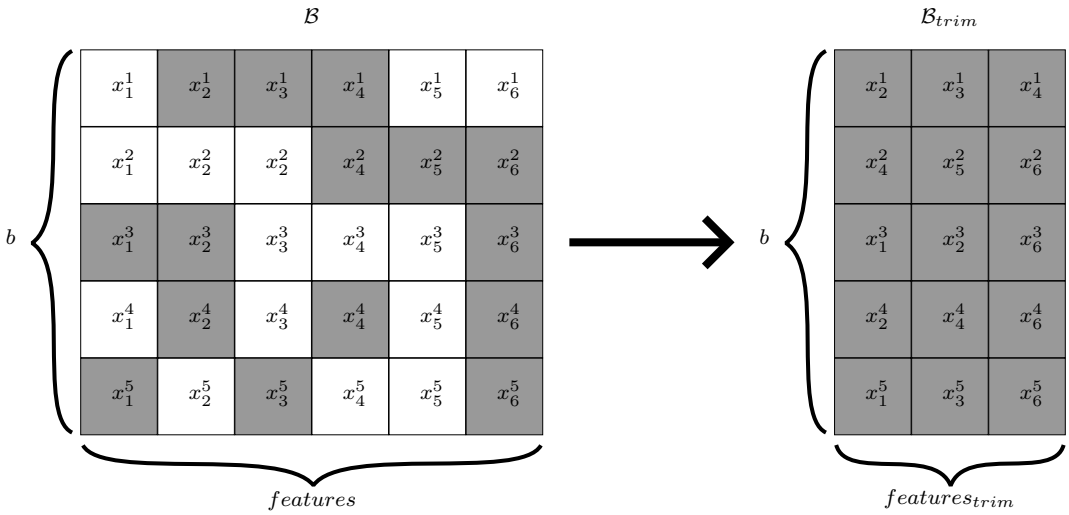


Figure 4.5: Representation of how the trimming is applied. On the left, we show the raw input, with missing values being removed in white. On the right, we show the trimmed input, which is used for embedding. Note how the amount of missing values, despite being random per batch, is fixed between samples.

considered one of the best options when dealing with missing values, and can be combined with any downstream classifier [16, 88].

We combined MICE and an MLP classifier to create a second baseline. Lastly, MICE imputation was used again, but now combined with the proposed transformer implementation, only differing in how missing values are processed, In this case, the baseline performs imputation using MICE and the transformer is trained on a full (MICE imputed) dataset.

No comparison was made against other related models [53, 54, 55], due to an inability of such models to: deal with continuous missing values, having only task-specific results for missing values, and not providing information about performance with missing values during inference, respectively.

We use accuracy as the primary metric to compare the proposed algorithm with the baselines. Also, to maintain a fair comparison, a manual tuning strategy was applied to hyperparameters, both for baselines and the proposed algorithm. Additionally, we present the accuracy of the main models while varying the number of missing features.

4.3.2 Proposed Versions

The proposed algorithm was applied considering three different training scenarios with respect to missing values, which led to three different implementations of the model. These scenarios represent how missing values are presented during training, being (1) fixed missing values, (2) different missing

values in each epoch, and (3) different missing values in each batch. Note that, as collected, all datasets are full, with no missing values, which in turn allows these scenarios to be created.

In scenario (1) *static missingness*, a random amount of missing values are inserted in the training data, only once, before training starts. The datasets do not suffer any change during training with respect to missing values. This simulates a training-time missingness scenario. Differently, in scenario (2), random missing values are inserted in the full dataset for each epoch. Because of that we call this *dynamic missingness per epoch*. Lastly, in scenario (3) *dynamic missingness per batch*, different missing values are inserted in each batch during training.

Differently from versions (1) and (2), in the last scenario, missingness is handled by the model itself. For each received batch, a random amount of missing values is chosen and that many values are randomly set to missing for each sample. Moreover, for scenario (3), missing values are handled by trimming the context only to available values, while the other scenarios use masks in order to keep missing values unnattended. This is required because their batches can have samples with different amounts of missing features.

4.3.3 Network Architecture

The architecture of the model is based on the GPT model [67], which is decoder-only. The code is based on the [66] codebase, with appropriate changes to incorporate the proposed use case. Like in the original architecture, a dropout layer is employed after embeddings, and layer normalization is applied right after the attention blocks.

The transformer architecture was chosen especially because of its ability to utilize context information, which can be a substitute for missing information. In other words, learning context information might be a way to overcome missingness. This reflects even more in cases where, during training, all information is available, since by randomly dropping out feature values, the model can learn to depend on context representation without any loss, possibly resulting in better test-time performance, even with missing values. This scenario is not usually accounted for in ML algorithms, which generally consider dropout at a neuronal level.

As mentioned, missing data is handled in two different ways, which led to slightly different architectures. The first, which handles missing values by masking, is shown in Figure 4.6. This architecture maintains most of the original architecture, while applying an attention mask to whenever tokens

that represent missing values. Differently, in the second approach the model is responsible for inserting random missing values to the input, and trimming those values, leading to a smaller context to be embedded. This approach can be seen in Figure 4.7

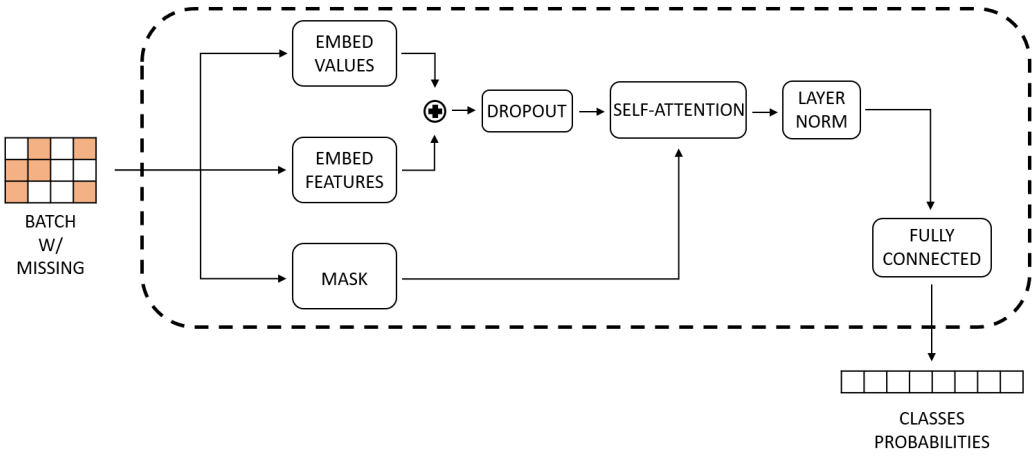


Figure 4.6: Masking architecture. The model receives a batch of data with missing values. The inputs go through two embedding layers, one embedding values, and the other embedding features. The mask is built based on the missing values int he input batch, and used during the attention mechanism.

The chosen hyperparameters, in part, are based on the same codebase, with a few changes to account for a different problem, such as less attention heads, and smaller embedding dimension. The hyperparameter list is described in Table 4.1

Table 4.1: Hyperparameters

Hyperparameter	Value
Attention layers	6
Attention heads	4
Embedding dimension	128
Batch size	128
Dropout	0.1
Initial learning rate	6e-4
Learning rate schedule	cosine decay
Early stopping patience	300 epochs

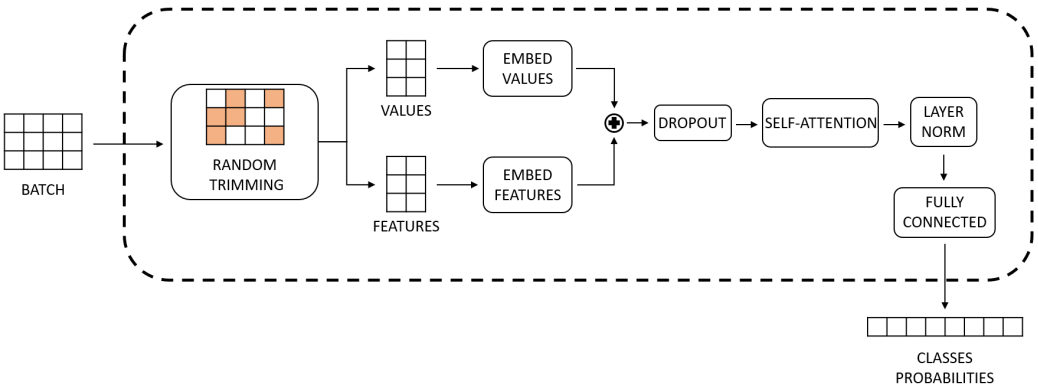


Figure 4.7: Trimming architecture. The model is fed with a full batch (no missing values), then the trimming approach is applied. The trimmed input goes through both embedding layers, similar to the masking approach.

4.4 Results

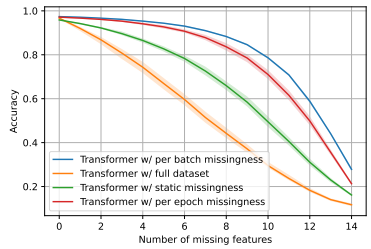
In this section we show and assess the results for the experiments described in the previous section. Moreover, we compare the proposed models and the baselines.

As mentioned before, there are three scenarios were the proposed models are assessed, and two model variations. Here is how these characteristics interact: the *masking approach* is suited both for *static missingness* and *dynamic missingness per epoch* scenarios, where the whole input can be fed to the model, with missing values included as is; the *trimming approach* is better suited for *dynamic missingness per batch*, where the full data is fed to the model, which inserts random missingness each batch itself.

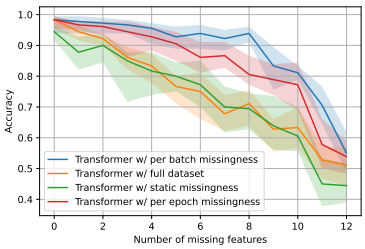
First, we compare the proposed models in all scenarios against each other, also addressing a baseline model that is trained with full data (no missing values). The results are based on versions of the test set that iteratively increased the number of missing values per sample.

In Figure 4.8, we can see how the models performed accuracy-wise, per number of missing features. In general, the trimming model, which handles missing data per batch, achieved better performance than the other approaches, even being constrained by a fixed amount of missing values for each batch. Moreover, even though the distribution of missingness is the same as in the per-epoch missingness, grouping the number of missing values per batch not only allows for a more efficient implementation through trimming, but also increases accuracy.

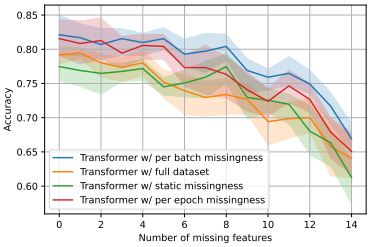
Now, we address the baselines mentioned before, comparing the best



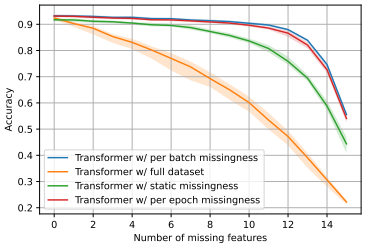
(a) Pendigits: transformer accuracy X missing features



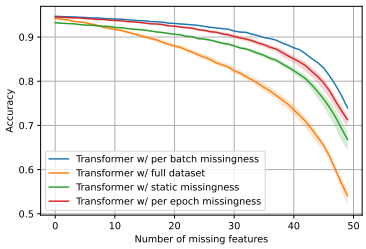
(b) Wine: transformer accuracy X missing features



(c) Healthrisk: ransformer accuracy X missing features



(d) Beans: transformer accuracy X missing features



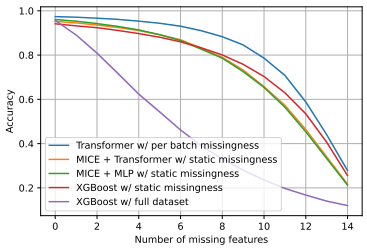
(e) Miniboone: transformer accuracy X missing features

Figure 4.8: Performances of the proposed Transformer models on test data. The shaded regions are the 95% bootstrap confidence intervals.

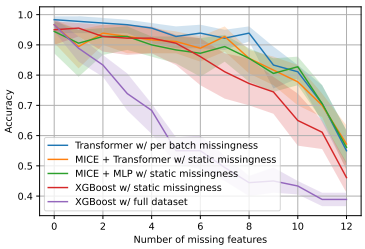
proposed model with them. To summarize, the proposed baselines are: *MICE + Transformer*, *MICE + MLP* and *XGboost*. For all baselines, we considered two scenarios, one without any missing values (full training data), and other with static missing values in the training set. Models trained on scenarios with missing values in the training data consistently achieved better performance.

Dynamic missing features were not considered, because the baseline methods do not support that use case. Since MICE is used to input missing values based on a trained model (regression in this case), all samples are necessary to better train this model. XGBoost is not specifically designed to deal with batch learning, while also not benefiting from changes in data during training.

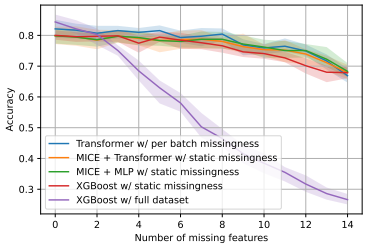
Results are shown in Figure 4.9, comparing the best proposed model



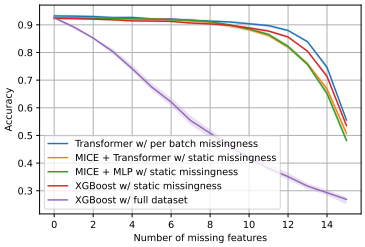
(a) Pendigits: baselines accuracy X missing features



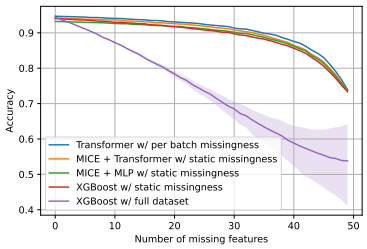
(b) Wine: baselines accuracy X missing features



(c) Healthrisk: baselines accuracy X missing features



(d) Beans: baselines accuracy X missing features



(e) Miniboone: baselines accuracy X missing features

Figure 4.9: Best variation performances on test data from the proposed Transformer and baselines.

version with the best baseline versions. Again, results are based on an iterative increasing number of missing values in the test set. In general, the proposed model achieved better results than the baselines.

In all cases, as expected, training with missing values yielded better performance than training with a full dataset, sometimes even when testing with no missing values as well. Hypothetically, this can be seen as an effect similar to dropout, where the presence of missing values in training helps the model to generalise better, forcing the learning of patterns through different features' relationships.

Lastly, to summarise the results in one experiment, models were assessed on test sets containing a random amount of missing values. In this setup, a random number of missing values is inserted on random features for each

Table 4.2: Model test set accuracies averaged over 10 training runs

Model	Pendigits	Wine	Healthrisk	Beans	Miniboone	Average
T Full	0.524	0.754	0.728	0.665	0.826	0.699
T Static	0.650	0.760	0.735	0.820	0.867	0.766
T Epoch	0.766	0.843	**0.760**	0.870	0.893	0.826
T Batch	**0.808**	**0.880**	**0.778**	**0.878**	**0.906**	**0.850**
M + T Static	0.733	**0.856**	**0.758**	0.854	0.899	0.820
M + MLP Static	0.732	0.830	**0.770**	0.852	0.892	0.815
XGB Static	0.747	0.841	0.757	0.865	0.891	0.820

sample in the test data. Obtained results, showed in Table 4.2 corroborate the previous findings, demonstrating that the proposed model with per-batch missingness consistently achieved better performance. Best results are highlighted in bold, together with values that fall within its confidence interval. To condense the table size, aliases are applied to the models following: "T Full" is "Transformer with full dataset, "T Static" is "Transformer with static missingness", "T Epoch" is "Transformer with missingness per epoch", "T Batch" is "Transformer with missingness per batch", "M + T Static" is "MICE plus Transformer with static missingness", "M + MLP Static" is "MICE plus MLP with static missingness" and "XGB Static" is "XGBoost with static missingness".

It is noteworthy that imputation baselines do perform similar to the proposed model in cases with a small amount of data and high amount of missing values, such as in Wine and Healthrisk datasets. This may indicate that the proposed model performance is limited in datasets with few samples, which is usually expected for transformers.

5
Reinforcement learning with transformers for classification with costly features

As mentioned earlier, transformer architectures are powerful models with proven groundbreaking results in NLP [36, 67, 68], computer vision [69], and many other scenarios. This chapter aims to bring together the two previous ideas proposed in this thesis: train a transformer model in a RL framework, leveraging the attention mechanism and contextual information learning, to classify data with costly features, in an auto-regressive manner.

To do so, we start by pointing to works related to this idea, and key concepts used. Later, we talk about the proposed model itself, detailing the methods, techniques, and strategies used. Then, we talk about the experimental setup, and finish with the achieved results, comparing them with baselines.

5.1
Related Work

Regarding the problem of costly features, [13, 14] are the main inspiration to the proposed model. These works use a DQN-based agent to sequentially choose between measurements and classification, balancing between more information and more cost. Authors apply state-of-the-art RL techniques, such as Double DQN [34], Dueling DQN [35], and Retrace [70]. These and other RL-based approaches were already mentioned in Chapter 3 of the thesis.

5.1.1
Transformer-based Reinforcement Learning approaches

In the last few years, the research area where RL and Transformers are used together has received a lot more attention, directly influenced by the success of transformers in NLP. We now point to some works that leverage the transformer architecture in a RL framework.

5.1.1.1
Decision Transformer

The Decision Transformer model [66] abstracts RL as a sequence modeling problem, allowing Transformer architectures to be used in an offline RL

manner. It can be seen as conditional sequence modeling, learning the best actions to take, based on expected return, previous states and actions. As an auto-regressive model, it can also generate future actions capable of achieving the desired return. The model is capable of generating an adjusted set of auto-regressive actions given a desired return. We can see the base structure of the Decision Transformer in Figure 5.1

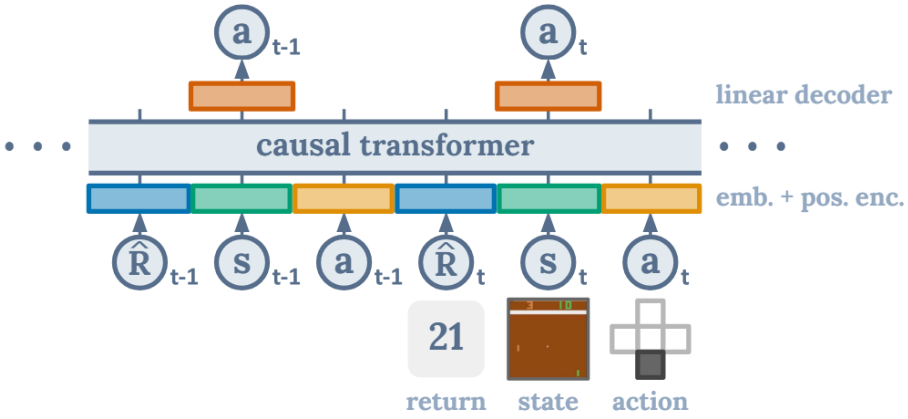


Figure 5.1: Decision Transformer base structure. Note how rewards, states, and actions are fed as a sequenced input.

In this structure, previous actions, rewards, and states are fed to the model in a sequence, while the model outputs the next action. Each input modality goes through a dedicated embedding layer and is added with an embedding for timestep, i.e. the positional encoding. More than one input state can be assigned to the same positional encoding, especially if the input is an image.

Decision Transformer (DT) is compared against different Temporal Difference (TD) RL approaches, such as Conservative Q-Learning (CQL) [80], and other well-known RL models like REM [81] and QR-DQN [82]. DT benchmarks were made in atari and gym environments and achieved matching or better performance compared to the baselines.

5.1.1.2 Trajectory Transformer

Similar to the previous model, the Trajectory transformer [71] also address the RL problem as a sequence modeling problem, aiming to produce a sequence of actions that leads to a sequence of high rewards. Unlike the Decision Transformer, this approach expects inputs as unstructured sequences of states, actions, and rewards, modeling the transition distribution, as seen in Figure 5.2.

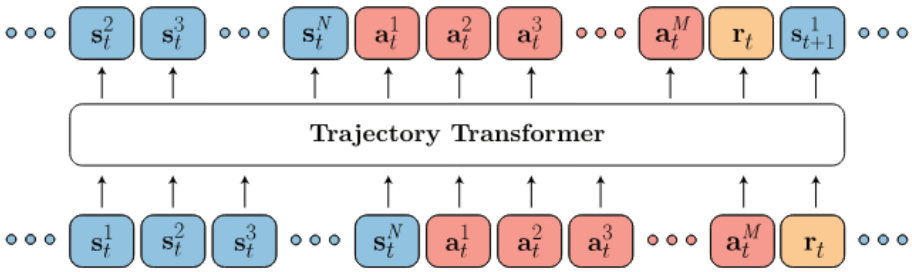


Figure 5.2: Trajectory Transformer base structure. In this model, states, actions, and reward sequences are discretized, meaning that the sequence of each type of information is processed as a subsequence.

This approach trains on auto-regressively discretized sequences of states, actions and rewards. What this means is that, for a state with dimension N, is discretized in a sequence with N elements, as seen in 5.2. Two different discretization approaches are addressed. Results show competitive or better performance than baselines, such as CQL, BRAC and DT.

5.1.1.3 Q-Transformer: Scalable Offline Reinforcement Learning via Autoregressive Q-Functions

The Q-Transformer [72] is also considered an offline RL model, with the main contribution being the per action-dimension tokenization of Q-values.

Continuous actions are discretized in bins, and during training, given a history of states, the q-values of all bins in all action dimensions are updated. Q-values of actions observed in the dataset are trained via the bellman update, while the ones not present in the dataset are minimized toward zero. Q-targets of all action dimensions are computed based on the maximization of the next action dimension within the same timestep. The Q-target of the last action dimension in a timestep is computed using the discounted maximization of the first action dimension in the next time step plus the reward. This process can be seen in Figure 5.3

To assess performance, authors used real-world offline datasets of control tasks composed with a small amount of human-demonstration episodes, all of which succeed in the proposed task, and autonomously collected data, which have a lower success ratio. Experiment results show better performance in control tasks than the compared baselines such as DT and Implicit Q-learning [83]

While the models discussed demonstrate strong performance in their respective tasks, some limitations justify opting for an approach more aligned

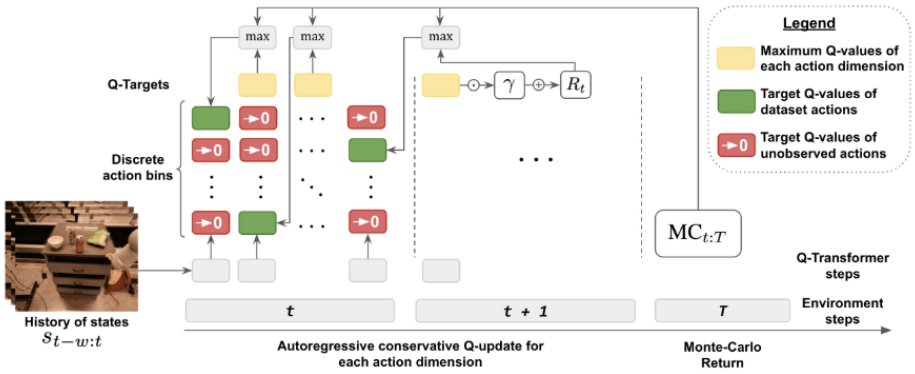


Figure 5.3: Q-Transformer base structure. Note how the target values are based on the next-dimension max Q-value.

with a vanilla Transformer architecture. The Q-Transformer, for instance, is designed to manage continuous action spaces, which are not present in CwCF problems, and its primary focus is on action dimensions rather than feature selection. Similarly, Decision Transformers tend to extract and stitch together optimal segments of trajectories from the training data, utilizing a returns-to-go strategy, which is less suited in CwCF contexts. Additionally, both the Trajectory Transformer and Q-Transformer are tailored for offline reinforcement learning, making them less suited for scenarios requiring continuous updates and dynamic feature selection, as seen in CwCF problems.

5.2 Methodology

In this section, we give details about the development and implementation of a transformer model integrated into an RL framework. As mentioned earlier, the main idea is to develop a similar model to the one in Chapter 3, leveraging the state-of-the-art characteristics of a transformer model, and enhancing policy learning and Q-value estimation. The datasets used follow previous chapters, with minor differences that will be exposed throughout the chapter.

Unlike the models previously implemented in this thesis, using a transformer model in an RL framework introduces scalability, as it is suited to deal with contextual data. With that, transformers learn richer information about the current context with the attention mechanism, which addresses the missingness from partial episodes (not seen features can be interpreted as missing values).

5.2.1 Problem setup

Since the problem we want to tackle is the same as in chapter 3, Classification with Costly Features (CwCF), we model it with a similar MDP. The objective of that problem is to train a model (policy) that, based on feature costs, is capable of balancing between measuring one extra feature or classifying with the already gathered information, at each timestep. This decision-making process is dynamic with respect to measured values and can be thought of as a type of dynamic feature selection.

5.2.2 MDP

Consider a dataset $\mathcal{D}$ with F features, the array $x = \{x_1, x_2, \ldots, x_{\mathrm{F}}\}$ represents one sample from $\mathcal{D}$, and $y \in [1, C]$ is the true class of such sample, with C being the number of different classes. The tuple (x, y) comprises one environment for the proposed agent.

In this setup, states are composed by two vectors. Vector $v = \{v_1, v_2, \ldots, v_t\}$ indicates values, and vector $f = \{f_1, f_2, \ldots, f_t\}$ indicates features. Note that the size of the vectors are dynamic, depending on which timestep t the model is currently in. Also, as initially no information is acquired, we denote v_1 and f_1 as initial states, the former being empty and the latter receiving a category that indicates "no-feature".

Regarding actions, they are divided in two groups, *measurement actions* $\mathcal{A}_{\mathrm{m}}$ and *classification actions* $\mathcal{A}_{\mathrm{c}}$. A measurement action a_{m}, with $1 \leq \mathrm{m} \leq \mathrm{F}$, is used to denote the next feature do be measured, pointing to how the state vectors are going to be updated for next timestep. For instance, if a_{m} is chosen, it points to the feature with index m as being acquired next. At the beginning of the next time step, v and f would be extended with the value and the index from that feature, respectively. As for classification actions a_{c}, $1 \leq \mathrm{c} \leq C$, they are used to denote to which class the information acquired so far is assigned. Classification actions are always terminal actions, as classification is the model's final objective. With that in mind, the full set of actions has $F + C$ possible actions.

Given an action a, the transition function maps such an action to two possible outcomes, an information addition in both state vectors if it is a measurement action, or to the terminal state if it is a classification action.

Formally, we can define it as:

$$T(v, f, a) = \begin{cases} (v', f') & , \text{if } a_{\text{m}} \\ \text{terminal} & , \text{if } a_{\text{c}} \end{cases}$$

where $v' = \text{append}(v, x_{\text{m}}|a_{\text{m}})$, and $f' = \text{append}(f, \text{m}|a_{\text{m}})$ are the next state vectors.

Following the previous setup, without lack of generality, we fix the costs of all features and set it to $\mathbf{c}_{\text{f}} = 0.03$. The rewards of correct classification and incorrect classifications are respectively set to 1 and -1. All rewards were scaled down compared to Chapter 3 to facilitate some operations during implementation. This does not affect the performance. We can formally define the reward functions as

$$R(s, a) = \begin{cases} -\mathbf{c}_{\text{f}} & , \text{if } a_{\text{m}} \\ 1 & , \text{if } a_{\text{c}} = y \\ -1 & , \text{otherwise} \end{cases} . \tag{5-1}$$

A full episode is exemplified in Figure 5.4. In the beginning, the value state vector v is empty, as there is no information available, and the feature state vector f has one element representing the "no features measured" information. From there, for each measurement action, both state vectors are appended with the respective information. Whenever a classification action is chosen, the episode terminates. For simplicity, we denote f_0 as the initial element in f.

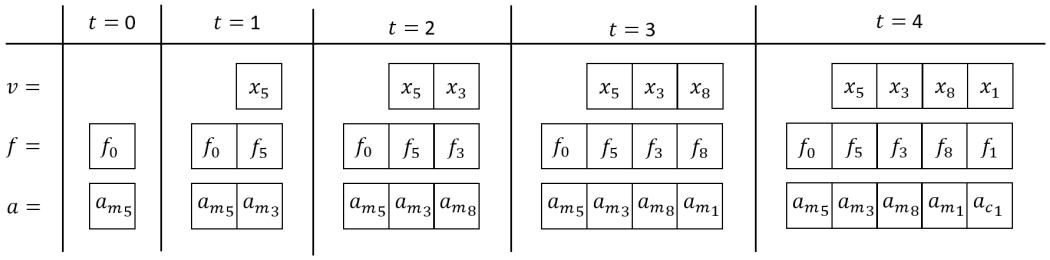


Figure 5.4: Full episode example. All vectors increase dimension with each time step. This is only allowed because transformers can deal with varying length inputs.

5.2.3 Training process

Now, the training process is described in detail. Policy strategy, memory replay, model embeddings, network architectures and training strategy are

explained in the following sections.

5.2.3.1 Epsilon greedy strategy

The ε-greedy policy strategy, explained in Equation 2-12, was applied with exponential decay during the model's training. This strategy induces more exploration at the start of the training, while exploitation takes place as training proceeds. This is suited for RL agents since at the beginning, the model does not have any reference on what actions provide higher rewards, and exploration provides different sequences of actions that will be added in the replay memory and trained on. With time, the model gradually learns sequences of actions that provide higher rewards, therefore it is better to exploit the already gathered knowledge for the most part, only having a small chance to explore aiming to avoid local minima. For that, ε starts at 1 and is decayed exponentially until $\varepsilon = 0.1$.

In this implementation, target updates are performed for classification actions along with the chosen measurement action, as opposed to Chapter 3, where all Q-values related to measurements and classification actions were updated. This change is made to avoid an unmanageable computational cost, since to achieve a similar update process, we would need to read out all transformer outputs based on combinations of current and next states. In comparison with the All-action updates, adopted in Chapter 3, this is probably less eficient way of updating targets, sample-wise.

In the proposed problem, repeating measurement actions would provide no benefit to the model, as no new information would be gathered and more cost would be added. Because of that, repeated actions are not allowed in this setup. During training, a "full episode strategy" is applied, which limits the agent to choose measurement actions as long as there is still unmeasured information, forcing episodes with full context.

5.2.3.2 Building memory

When populating memory, RL algorithms such as DQN, usually add transitions, which are changes from one timestep to another. Since the reward is based solely on action, and next state is deterministic, it is enough to store only the ground truth data array x and class y, and the sequence of actions taken. The full episode can be rebuilt with the right sequence, using the stored information.

Considering the ε-greedy strategy, the memory is initially filled with episodes generated from random actions. As the model trains, episodes gradually start to have actions based on the greedy policy. The size of the memory is set to 1M, and after that many episodes are stored, the older ones are overwritten with new ones.

5.2.3.3 Embeddings

Embeddings are important to extract information and represent input values. They convert input values into more complex learnable vectors, improving the representation and carrying information from input values.

A trainable linear layer is applied to embed values from v, while a learnable lookup table is responsible for embedding the feature values from f. Both embedding layers are configured to output a vector $\mathbb{R}^{d_{emb}}$, with embedding dimension d_{emb} elements. The embedding linear layer maps a single value ($\mathbb{R}^1$) to the mentioned vector, with a *tanh* activation function, and the embedding lookup table maps a feature index $\mathbb{N}^1$ to the mentioned vector. The embeddings are added together before the attention block.

5.2.3.4 Network Architecture

Although the network architecture is based on the GPT [67] model, which is a decoder-only transformer, we handle the inputs as an encoder-only model, since all inputs are allowed to communicate with each other during attention. The proposed model is mainly composed of the embedding blocks, multi-head self-attention blocks, and two final heads, the measurement head and the classification head. The overview of the architecture can be seen in Figure 5.5.

The process starts by feeding the value vector v and feature vector f to the model, both having $(batch, context)$ dimension. The first step is to add the "start indicator" in the feature vector. This informs the model when no information is available but the episode has already started. At this point, vector f has dimensions of $(batch, 1 + context)$.

Following, the data goes through the embedding layers, generating a $(batch, context, d_{emb})$ value embedding vector and a $(batch, 1 + context, d_{emb})$ feature embedding vector. These vectors are added together along the last dimension. The extra element in the feature embedding is left untouched. The generated embedding vector holds information about each value present in

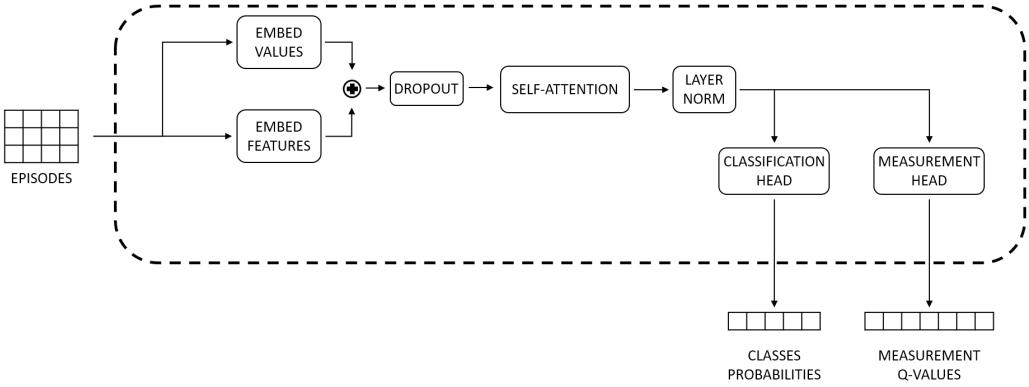


Figure 5.5: Proposed Transformer model architecture with dedicated embedding blocks for both values and features. The model includes two output heads: a classification head and a measurement head.

the context and their respective features. A visual representation is shown in Figure 5.6.

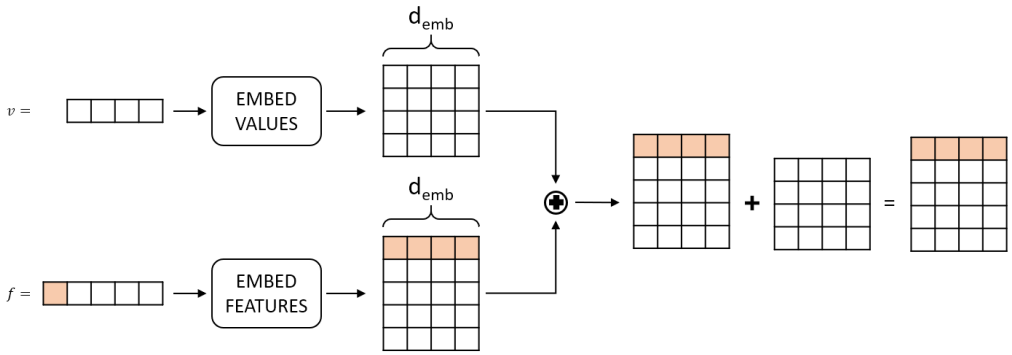


Figure 5.6: embedding blocks. The "no-information" embedding remains the same, and addition is performed over other embeddings.

The embedding vector is fed to the attention layers, which are based on [66]. Attention values are extracted and go through a layer normalization. At this point, the calculated values feed a classification head and a measurement head. Both heads are fully connected layers, with input size of d_{emb} and output size of C and F respectively.

The classification head is responsible for outputting the logits for each class based on the available information, while the measurement head is responsible for outputting the Q-value for each possible action.

5.2.3.5 Training strategy

Given the mentioned architecture, the training process applied, in general, follows a basic DQN training strategy. We can divide the training process in three steps: memory populating, memory sampling, and training step.

To populate memory, the ε-greedy exploration strategy was applied. At each training step, the ε value is decayed by a factor dependent on how many episodes the model will be training, reaching the lowest value of 0.1 at $\frac{2}{3}$ of maximum training episodes. The decaying factor is a function of how many training steps are set to be processed, such that

$$\text{decay} = 0.1^{\frac{3}{2\text{Tr}}},$$

with Tr being the total training steps.

This translates to full exploration at the beginning of the training, gradually moving to exploitation. Exploration is still maintained at 10% in the final half of the training, allowing the model to move from local minima if those occur.

Once enough samples are stored in memory, one training step is executed after each episode rollout is added to the memory. In this training step, a batch of random episodes is sampled from memory. Each sample s stores the features indexes vector f, the true value vector v_{true}, and the true class of that sample y_{true}. With that, the sample value vector v is rebuilt based on f. The batch is truncated with a random amount of information, similar to what is done in Chapter 4.

The model is then trained over this batch. The attention mechanism does not apply any masks, meaning that all queries and keys are allowed to communicate with each other. This ensures that context information is learned in all scenarios, from small to full contexts.

It is noteworthy how the test-time missing values problem from chapter 4 is incorporated in this chapter in the form of unread features. The lack of information during test time has the same characteristics as unread features in the current modeling. In practice, the model ignores the unseen values, as they are not part of the input, similar to the trimming approach.

Results based on the full context coming from the attention blocks go through two heads, the measurement and the classification head. The classification head is formed by a linear layer and outputs the logits for each class. As in Chapter 3, the measurement head is formed by two fully connected layers and outputs the Q-values for all measurement actions.

To train the model, a loss function that covers both heads is needed, therefore Mean Squared Error (MSE) is used to deal with measurement Q-values, while Cross Entropy (CE) is used to deal with classification. Q-value updates follows the usual approach in RL frameworks, such as the one exposed in Chapter 3. With that, the loss

$$\mathcal{L}(Y, \hat{Q}, y, \hat{y}) = \text{MSE}_{\text{measurement}} + \text{CE}_{\text{classification}} \tag{5-2}$$

, is applied with

$$\text{MSE}_{\text{measurement}} = (\hat{Q} - Y)^2 \tag{5-3}$$

, where $\hat{Q}$ is the model predicted Q-value and Y is the updated target Q-value, and

$$\text{CE}_{\text{classification}} = -\sum_{c=1}^{C} \mathbb{I}_y \log(P(\hat{y})) \tag{5-4}$$

, where $\mathbb{I}_y$ is the indicator vector specifying if the class element c is the correct class, and $P(\hat{y})$ is the predicted probability distribution between classes.

5.2.4 Validation and test

During validation and testing, the model is assessed by auto-regressive generation of actions. Each measurement action returns the respective cost of the chosen feature, which is fixed in these experiments, and classification actions return a reward of 1 or -1 for a correct or incorrect classification, respectively.

In an RL approach, the greedy action is chosen based on the highest Q-value in that state. To do that, the logits from the classification head need to be converted to Q-values. That is achieved by first applying a softmax function to the classification head logits, generating a probability distribution, and converting from the distribution to equivalent Q-values for each classification action, such as in

$$\begin{aligned} Q_{\text{equivalent}} &= P(\hat{y_c}) - 1(1 - P(\hat{y_c})) \\ &= 2P(\hat{y_c}) - 1 \end{aligned} . \tag{5-5}$$

The model generates a sequence of actions, while state values and features are updated between one action and another. If there is a case where the model is more certain about to which class the measured information pertains, it will finish the episode earlier by classifying the data.

This approach allows the model to interpret each episode as a totally different environment. Moreover, the initial state is the same for all episodes, since the starting point is the absence of any information.

5.3 Experiments

The set of experiments executed to assess the model's performance follows similar standards to the ones defined in Chapter 3. Exposed results are an average of ten runs with different seeds. Feature costs are fixed at $\mathbf{c}_f = 0.03$, and as noted, this is equivalent to the previously used costs, only scaled down along with classification rewards of 1 and -1 for correct and incorrect classifications, respectively.

Experiments were carried out in the same six datasets: 'Synthetic', 'Wine'[40], 'Healthrisk'[45], 'Miniboone'[43], 'Pen Digits'[41] and 'Beans'[44].

Performance is assessed based on accuracy and return. The results are evaluated against the baselines and the proposed model in Chapter 3. Transformer results are scaled up to be at the same level as those from other models, both in Table 5.1 results and graphs in Figure 5.9. The hyperparameters of the proposed algorithm are initially based on the Decision Transformer implementation and are manually tuned from there to ensure a fair comparison with the baselines, which are also manually tuned.

5.4 Results

Now, experimental results are shown and discussed for every dataset considered. Graphs of validation performance during training are also exposed. Note that the transformer model was trained on ten different seeds, with 50000 training steps for Synthetic, Wine, Healtrisk, Pen Digits and Beans datasets, and with 300000 steps for miniboone.

5.4.1 Verification

In the synthetic dataset used for verification, the transformer model achieves statistically equivalent test returns and accuracies as the DRL model proposed in Chapter 3. As noted there, this dataset is not meant to be a challenge or to assess the model's performance, but rather to understand how it behaves in all possible states.

In Figure 5.7, we can see the trained model's policy. The graph is divided in four plots, each representing the (un)measured features. Upper left plot represents states with measured information about feature 1 and no information about feature 0. Upper right plot represent states with both features measured. Bottom left plot represent states with no measured features. And bottom right plot represent states with only feature 0 measured. The

model was capable of learning a good classification boundary, even though it seems more strict regarding class 0 if compared to the model in Chapter 3. Also, the latter was able to learn an upper limit for feature 1 so that samples beyond that are classified as class 1, which does not seem to be the case in the transformer model.

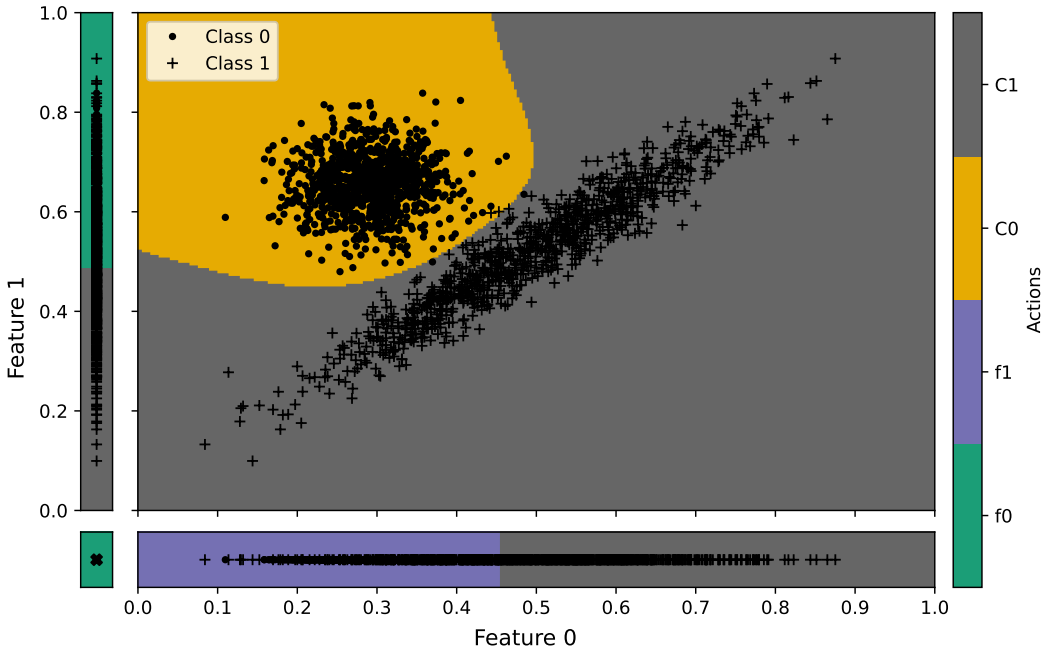


Figure 5.7: Transformer policy on 'Synthetic' dataset. Actions M0 and M1 are measurement actions for the respective features. C0 and C1 are classification actions for the respective classes. This model achieves the same behavior as the one from chapter 3, as it can classify early when there is enough information, and it measures another feature when there is overlapping between classes.

Figure 5.8 shows the maximum value of Q learned by the model for every possible state. The model is very confident about all actions taken. Again, we see a clear decision boundary between the classes distributions. Even in regions with mixed classes, e.g. between 0.2 and 0.3 in bottom right plot, the model seems to have a high "confidence" when compared to the one in Chapter 3.

In terms of performance, it achieved a slightly better return for validation data than the other models. Validation performance during training can be seen in Figure 5.9a. As noted before, this did not translate into significantly better test performance (see also Table 5.1)

5.4.2 Learning curves

During validation the transformer model achieved different levels of return and accuracy depending on the specific dataset. Return-wise, for

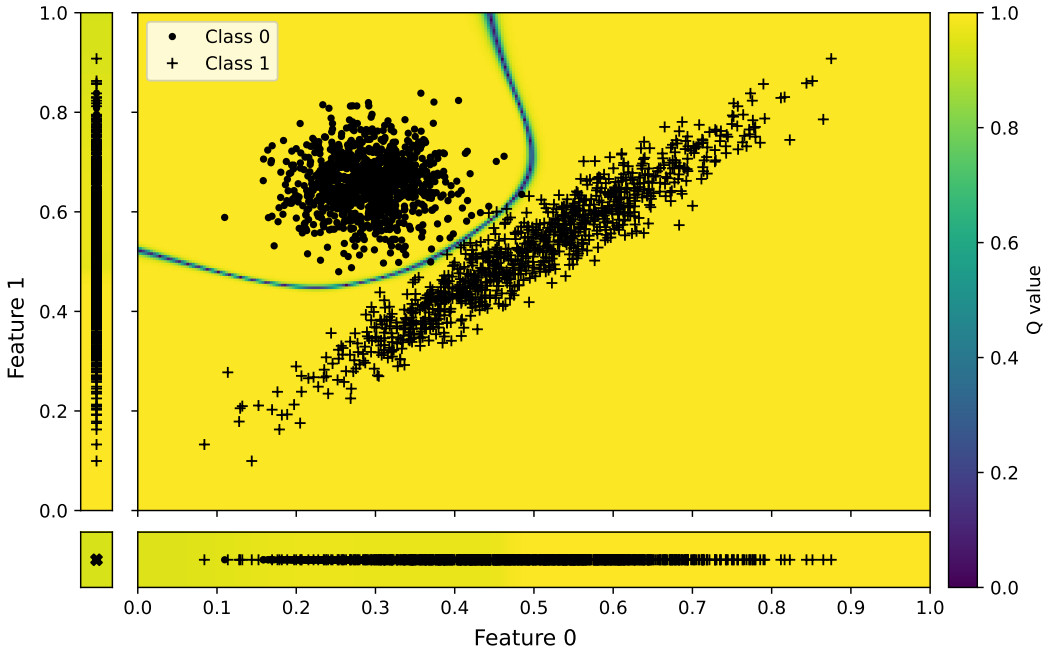


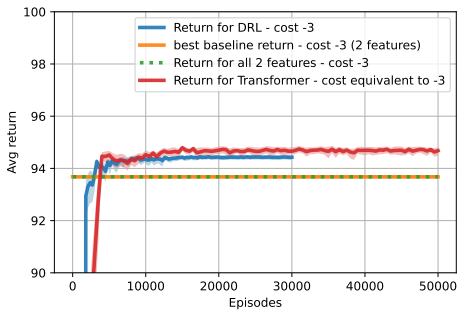
Figure 5.8: Transformer agent maximum Q values. The decision boundary is thinner, and apparently, this model has more "confidence" if compared to the one from chapter 3.

Synthetic, Wine and Pen Digits, the best return performance achieved was slightly higher or on par with the compared models. For Healthrisk, Miniboone and Beans, returns achieve a lower performance. On the other hand, accuracy-wise in general the transformer model achieved a better performance, with lower performance only on Healthrisk data. Return results are shown in Figure 5.9.

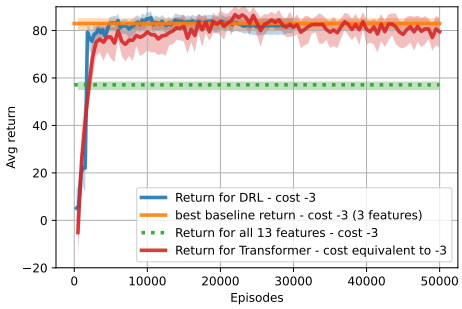
The difference in training episodes between proposed models is due to the slower convergence of the transformer model. Figure 5.9 depicts how the transformer architecture took more training steps to achieve convergence during training. Therefore, in order to show the best performance, more episodes were necessary when training the proposed transformer model.

5.4.3 Performances on test data

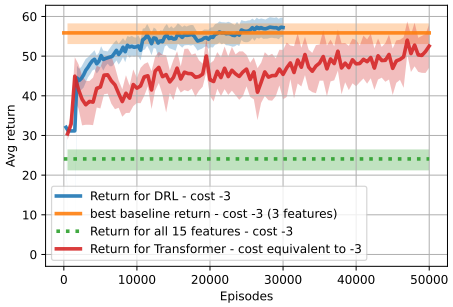
The performances on test data are shown in Tables 5.1 and 5.2, respectively describing return and accuracy values achieved. Return-wise, the proposed transformer model achieved similar or better performance on three datasets, Synthetic, Wine, and Pen Digits, while accuracy-wise it achieved better performance than the baselines in four datasets, Wine, Miniboone, Pen Digits, and Beans.



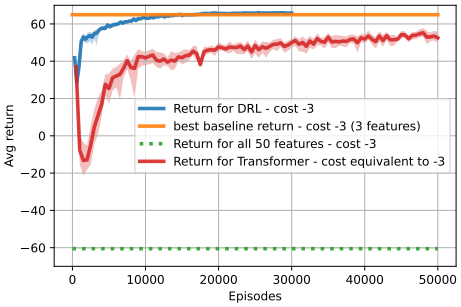
(a) Average returns on Synthetic dataset



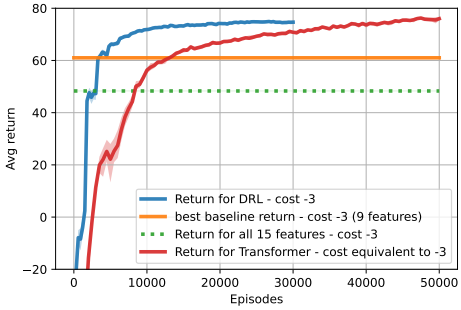
(b) Average returns on 'Wine' dataset



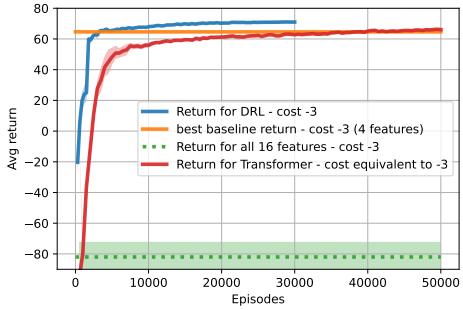
(c) Average returns on 'Healthrisk' dataset



(d) Average returns on 'Miniboone' dataset



(e) Average cumulative returns on 'Pen Digits' dataset



(f) Average returns on 'Beans' dataset

Figure 5.9: Comparison between proposed models performances on validation data

Considering the results, we can argue that the transformer model is somewhat more cautious about classification since it achieves better accuracy while giving up on return. Perhaps adjusting the loss to give more importance to the measurement actions can improve the performance in terms of final return. Another hypothesis is that changing the costs and gamma could bring these results closer to the DRL model. A lower cost could influence the model to measure less, however, it is still unclear why the models achieved different outcomes since the problem configuration was the same.

For Healthrisk dataset, the proposed model achieved a lower performance both in return and accuracy. This might indicate that the model could not learn good context representations, or that there is some overlooked characteristic in this dataset, which negatively affects the attention mechanism more than other algorithms. On the other hand, for Pen Digits dataset, the proposed Transformer model achieved better results on both performances, showing that it was capable of achieving improved results.

It is important to clarify that the proposed DRL model was tested using 40 different seeds, while the Transformer model was tested with 10 different seeds. As a result, the latter may produce wider confidence intervals, but this does not necessarily imply that it is less stable than the former.

Table 5.1: Cumulative Return Performances on test data

	Cumulative Average Returns			
	All features Baseline	Baseline	DRL	Transformer
Synthetic	$93.68\,\binom{93.8}{93.6}$	$93.68\,\binom{93.72}{93.62}$	$\mathbf{94.52}\,\binom{\mathbf{94.58}}{\mathbf{94.44}}$	$\mathbf{94.50}\,\binom{\mathbf{94.60}}{\mathbf{94.26}}$
Wine	$57.11\,\binom{58.4}{54.8}$	$\mathbf{82.94}\,\binom{\mathbf{85.17}}{\mathbf{80.17}}$	$\mathbf{81.51}\,\binom{\mathbf{83.93}}{\mathbf{78.73}}$	$\mathbf{83.23}\,\binom{\mathbf{88.29}}{\mathbf{77.56}}$
Healthrisk	$24.09\,\binom{26.4}{21.2}$	$\mathbf{55.86}\,\binom{\mathbf{58.25}}{\mathbf{53.04}}$	$\mathbf{57.05}\,\binom{\mathbf{59.26}}{\mathbf{54.87}}$	$47.96\,\binom{52.07}{43.07}$
Miniboone	$-60.55\,\binom{-60.4}{-60.8}$	$64.95\,\binom{65.12}{64.77}$	$\mathbf{66.38}\,\binom{\mathbf{66.70}}{\mathbf{66.05}}$	$58.79\,\binom{60.72}{57.30}$
Pen Digits	$48.31\,\binom{48.4}{48.2}$	$61.02\,\binom{61.33}{60.64}$	$66.94\,\binom{67.51}{66.36}$	$\mathbf{69.99}\,\binom{\mathbf{70.66}}{\mathbf{69.15}}$
Beans	$36.85\,\binom{37.3}{36.4}$	$64.63\,\binom{65.07}{64.18}$	$\mathbf{71.12}\,\binom{\mathbf{71.56}}{\mathbf{70.75}}$	$67.19\,\binom{68.10}{66.32}$

Table 5.2: Accuracy Performances on test data

	Average Accuracy			
	All features Baseline	Baseline	DRL	Transformer
Synthetic	$0.998\,(^{0.999}_{0.998})$	$\mathbf{0.998}\,(^{\mathbf{0.999}}_{\mathbf{0.998}})$	$0.997\,(^{0.998}_{0.997})$	$0.998\,(^{0.998}_{0.996})$
Wine	$0.981\,(^{0.987}_{0.969})$	$\mathbf{0.960}\,(^{\mathbf{0.971}}_{\mathbf{0.946}})$	$0.939\,(^{0.951}_{0.922})$	$\mathbf{0.958}\,(^{\mathbf{0.972}}_{\mathbf{0.917}})$
Healthrisk	$0.845\,(^{0.857}_{0.831})$	$\mathbf{0.824}\,(^{\mathbf{0.836}}_{\mathbf{0.810}})$	$\mathbf{0.834}\,(^{\mathbf{0.845}}_{\mathbf{0.823}})$	$0.811\,(^{0.830}_{0.783})$
Miniboone	$0.947\,(^{0.948}_{0.946})$	$0.870\,(^{0.871}_{0.869})$	$0.863\,(^{0.865}_{0.860})$	$\mathbf{0.928}\,(^{\mathbf{0.929}}_{\mathbf{0.926}})$
Pen Digits	$0.967\,(^{0.967}_{0.966})$	$0.940\,(^{0.942}_{0.938})$	$0.922\,(^{0.926}_{0.919})$	$\mathbf{0.951}\,(^{\mathbf{0.955}}_{\mathbf{0.948}})$
Beans	$0.924\,(^{0.927}_{0.922})$	$0.883\,(^{0.885}_{0.881})$	$0.908\,(^{0.911}_{0.906})$	$\mathbf{0.925}\,(^{\mathbf{0.928}}_{\mathbf{0.921}})$

6
Conclusions

This thesis tackles the challenge of Classification with Costly Features (CwCF), a significant issue in real-world applications, as data retrieval often incurs costs that are frequently overlooked. This work aims to develop novel state-of-the-art models for CwCF problems, utilizing the paradigm of reinforcement learning (RL) to achieve this goal. The contributions are structured around three main models: two specifically designed for the CwCF problem and one focused on classification tasks.

First, we note that the Chapter 3 proposed DRL model achieves better or similar results when compared to the baselines. Reward-wise, the proposed RL model achieves better performance in four datasets, while achieving similar performance in the other two datasets. The noted downside in accuracy is expected as the objective is indeed balancing between more information and lower classification cost. In practice, the model is being trained to maximize classification accuracy, while minimizing cost, which translates to less information.

The 'synthetic' dataset provided a way of demonstrating the model choices in an easy-to-visualize scenario. With this simplified data, it was possible to show how the proposed method behaves in different states, with different levels of information. A possible reason why the performance was not as good as desired in two datasets is that the model might be data-dependent, i.e. it suits datasets with specific characteristics, for instance, data that can be classified with less information, while falling behind in performance on datasets without that characteristic.

Only a few parameters (number of training episodes, exploration decay and learning rate schedule) were adjusted between experiments, yet the model performed consistently well across all datasets and scenarios. Moreover, this approach maintains the RL framework, facilitating the incorporation of new improvements suited for DQN.

Second, the transformer-based classifier showed better performance compared to the baseline methods when handling datasets with missing values. This is likely an effect of the model's attention mechanism, which allows learning the input context effectively. By attending to different parts of the input,

the model can leverage the learned relationship between observed features to better predict the class. This resulted in better overall performance in scenarios with incomplete data.

Additionally, transformer models are flexible and scalable, offering advantages over more classic DL algorithms. It can naturally manage incomplete data efficiently, simplifying data preprocessing, and avoiding imputation of missing values. Integrating contextual information during learning is a key factor that allows the transformer classifier to generalize well in different domains.

Third, the transformer-based agent trained in an RL framework achieved slightly lower performance regarding returns when compared to the DRL model, while achieving better accuracy. This indicates a more cautious classification process, acquiring more information before classifying the data. In principle, this behavior can be influenced by weights for each loss factor (MSE and CE).

6.1 Future Work

Regarding the first contribution, the DRL model is sufficiently flexible to adapt to various contexts and leverage ongoing advancements in the field, allowing for the incorporation of recent developments in RL to boost performance. Additionally, different misclassification rewards could be implemented to address scenarios where certain classes carry more significant consequences than others, such as certain diseases. Furthermore, a natural extension would be to consider actions that acquires multiple features simultaneously, which poses a challenge in avoiding an impractical number of actions for datasets with numerous variables. Lastly, enabling modifications to costs without the need for retraining the agent would greatly enhance the model's usability, making it adaptable to dynamic cost behavior in real problems.

For the second model, the Transformer-based classifier, leveraging available pre-trained models could significantly improve feature representation by addressing the words that represents each feature (feature title instead of feature index). This enhancement could enable the model to reason more effectively about features, thereby allowing it to interpret key terms within the task domain more accurately.

Finally, as previously mentioned, the Transformer agent trained within a RL framework can be influenced by loss weights. Therefore, implementing loss normalization with weighted loss, ensuring that all factors are on the same scale, could be a natural progression. Similar strategies to those used for the Transformer-based classifier could be applied here, utilizing the vocabulary of

each feature. Additionally, an auto-regressive approach, employing a decoder-based architecture, presents a promising alternative. Lastly, considering that the current work does not address missing data during training, employing masking techniques to handle unavailable features is a viable strategy.

Bibliography

[1] KAO, H.-C.; TANG, K.-F. ; CHANG, E.. **Context-aware symptom checking for disease diagnosis using hierarchical reinforcement learning**. In: PROCEEDINGS OF THE AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE, volumen 32, 2018.

[2] MANSOURYPOOR, F.; ASADI, S.. **Development of a reinforcement learning-based evolutionary fuzzy rule-based system for diabetes diagnosis**. Computers in Biology and Medicine, 91:337–352, 2017.

[3] PARVEZ, M. R.; BOLUKBASI, T.; CHANG, K.-W. ; SALIGRAMA, V.. **Robust text classifier on test-time budgets**. In: PROCEEDINGS OF THE 2019 CONFERENCE ON EMPIRICAL METHODS IN NATURAL LANGUAGE PROCESSING AND THE 9TH INTERNATIONAL JOINT CONFERENCE ON NATURAL LANGUAGE PROCESSING (EMNLP-IJCNLP), p. 1167–1172, Hong Kong, China, Nov. 2019. Association for Computational Linguistics.

[4] BOLUKBASI, TOLGA AND CHANG, KAI-WEI AND WANG, JOSEPH AND SALIGRAMA, VENKATESH. **Resource constrained structured prediction**. In: PROCEEDINGS OF THE AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE, volumen 31, 2017.

[5] NAN, F.; WANG, J. ; SALIGRAMA, V.. **Feature-budgeted random forest**. In: INTERNATIONAL CONFERENCE ON MACHINE LEARNING, p. 1983–1991. PMLR, 2015.

[6] NAN, F.; WANG, J. ; SALIGRAMA, V.. **Pruning random forests for prediction on a budget**. Advances in neural information processing systems, 29, 2016.

[7] XU, Z.; WEINBERGER, K. ; CHAPELLE, O.. **The greedy miser: Learning under test-time budgets**. arXiv preprint arXiv:1206.6451, 2012.

[8] ZHU, H.; NAN, F.; PASCHALIDIS, I. ; SALIGRAMA, V.. **Sequential dynamic decision making with deep neural nets on a test-time budget**. arXiv preprint arXiv:1705.10924, 2017.

[9] SHIM, H.; HWANG, S. J. ; YANG, E.. **Why pay more when you can pay less: A joint learning framework for active feature acquisition and classification**. arXiv preprint arXiv:1709.05964, 2017.

[10] KACHUEE, M.; KARKKAINEN, K.; GOLDSTEIN, O.; ZAMANZADEH, D. ; SARRAFZADEH, M.. **Cost-sensitive diagnosis and learning leveraging public health data**. arXiv preprint arXiv:1902.07102, 2019.

[11] WANG, K.; ZHANG, D.; LI, Y.; ZHANG, R. ; LIN, L.. **Cost-effective active learning for deep image classification**. IEEE Transactions on Circuits and Systems for Video Technology, 27(12):2591–2600, 2016.

[12] ZHU, M.; ZHU, H.. **Learning a diagnostic strategy on medical data with deep reinforcement learning**. IEEE Access, 9:84122–84133, 2021.

[13] JANISCH, J.; PEVNỲ, T. ; LISỲ, V.. **Classification with costly features using deep reinforcement learning**. In: PROCEEDINGS OF THE AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE, volumen 33, p. 3959–3966, 2019.

[14] JANISCH, J.; PEVNỲ, T. ; LISỲ, V.. **Classification with costly features as a sequential decision-making problem**. Machine Learning, 109(8):1587–1615, 2020.

[15] CHEN, T.; GUESTRIN, C.. **Xgboost: A scalable tree boosting system**. In: PROCEEDINGS OF THE 22ND ACM SIGKDD INTERNATIONAL CONFERENCE ON KNOWLEDGE DISCOVERY AND DATA MINING, p. 785–794, 2016.

[16] VAN BUUREN, S.. **Multiple imputation of discrete and continuous data by fully conditional specification**. Statistical methods in medical research, 16(3):219–242, 2007.

[17] AZUR, M. J.; STUART, E. A.; FRANGAKIS, C. ; LEAF, P. J.. **Multiple imputation by chained equations: what is it and how does it work?** International journal of methods in psychiatric research, 20(1):40–49, 2011.

[18] RUMELHART, D. E.; HINTON, G. E. ; WILLIAMS, R. J.. **Learning internal representations by error propagation**. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[19] LECUN, Y.; BENGIO, Y. ; HINTON, G.. **Deep learning**. nature, 521(7553):436–444, 2015.

[20] MASI, I.; WU, Y.; HASSNER, T. ; NATARAJAN, P.. **Deep face recognition: A survey**. In: 2018 31ST SIBGRAPI CONFERENCE ON GRAPHICS, PATTERNS AND IMAGES (SIBGRAPI), p. 471–478. IEEE, 2018.

[21] CHENG, D.; XIANG, S.; SHANG, C.; ZHANG, Y.; YANG, F. ; ZHANG, L.. **Spatio-temporal attention-based neural network for credit card fraud detection**. In: PROCEEDINGS OF THE AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE, volumen 34, p. 362–369, 2020.

[22] KOHONEN, T.. **The self-organizing map**. Proceedings of the IEEE, 78(9):1464–1480, 1990.

[23] ZHANG, W.; WANG, J.; JIN, D.; OREOPOULOS, L. ; ZHANG, Z.. **A deterministic self-organizing map approach and its application on satellite data based cloud type classification**. In: 2018 IEEE INTERNATIONAL CONFERENCE ON BIG DATA (BIG DATA), p. 2027–2034. IEEE, 2018.

[24] TEALAB, A.. **Time series forecasting using artificial neural networks methodologies: A systematic review**. Future Computing and Informatics Journal, 3(2):334–340, 2018.

[25] MAMMADLI, S.. **Financial time series prediction using artificial neural network based on levenberg-marquardt algorithm**. Procedia computer science, 120:602–607, 2017.

[26] MINSKY, M.; PAPERT, S.. **An introduction to computational geometry**. Cambridge tiass., HIT, 479:480, 1969.

[27] CYBENKO, G.. **Continuous valued neural networks with two hidden layers are sufficient**. 1988.

[28] LIPPMANN, R.. **An introduction to computing with neural nets**. IEEE Assp magazine, 4(2):4–22, 1987.

[29] RUMELHART, D. E.; HINTON, G. E. ; WILLIAMS, R. J.. **Learning representations by back-propagating errors**. nature, 323(6088):533–536, 1986.

[30] KINGMA, D. P.; BA, J.. **Adam: A method for stochastic optimization**. arXiv preprint arXiv:1412.6980, 2014.

[31] SUTTON, RICHARD S AND BARTO, ANDREW G. **Reinforcement learning: An introduction**. MIT press, 2018.

[32] MNIH, V.; KAVUKCUOGLU, K.; SILVER, D.; GRAVES, A.; ANTONOGLOU, I.; WIERSTRA, D. ; RIEDMILLER, M.. **Playing atari with deep reinforcement learning**. arXiv preprint arXiv:1312.5602, 2013.

[33] MNIH, V.; KAVUKCUOGLU, K.; SILVER, D.; RUSU, A. A.; VENESS, J.; BELLEMARE, M. G.; GRAVES, A.; RIEDMILLER, M.; FIDJELAND, A. K.; OSTROVSKI, G. ; OTHERS. **Human-level control through deep reinforcement learning**. nature, 518(7540):529–533, 2015.

[34] VAN HASSELT, H.; GUEZ, A. ; SILVER, D.. **Deep reinforcement learning with double q-learning**. In: PROCEEDINGS OF THE AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE, volumen 30, 2016.

[35] WANG, ZIYU AND SCHAUL, TOM AND HESSEL, MATTEO AND HASSELT, HADO AND LANCTOT, MARC AND FREITAS, NANDO. **Dueling network architectures for deep reinforcement learning**. In: Balcan, M. F.; Weinberger, K. Q., editors, PROCEEDINGS OF THE 33RD INTERNATIONAL CONFERENCE ON MACHINE LEARNING, volumen 48 de **Proceedings of Machine Learning Research**, p. 1995–2003, New York, New York, USA, 20–22 Jun 2016. PMLR.

[36] VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L.; GOMEZ, A. N.; KAISER, Ł. ; POLOSUKHIN, I.. **Attention is all you need**. Advances in neural information processing systems, 30, 2017.

[37] BAHDANAU, D.; CHO, K. ; BENGIO, Y.. **Neural machine translation by jointly learning to align and translate**. arXiv preprint arXiv:1409.0473, 2014.

[38] LUONG, M.-T.; PHAM, H. ; MANNING, C. D.. **Effective approaches to attention-based neural machine translation**. arXiv preprint arXiv:1508.04025, 2015.

[39] GEVA, M.; SCHUSTER, R.; BERANT, J. ; LEVY, O.. **Transformer feed-forward layers are key-value memories**. arXiv preprint arXiv:2012.14913, 2020.

[40] DUA, DHEERU AND GRAFF, CASEY. **UCI machine learning repository**, 2017.

[41] ALIMOGLU, F.; ALPAYDIN, E.. **Methods of combining multiple classifiers based on different representations for pen-based handwriting recognition**. In: PROCEEDINGS OF THE FIFTH TURKISH AR-

TIFICIAL INTELLIGENCE AND ARTIFICIAL NEURAL NETWORKS SYMPOSIUM (TAINN 96), 1996.

[42] AEBERHARD, S.; COOMANS, D. ; VEL, O.. **Comparison of classifiers in high dimensional settings (tech. rep. no. 92-02)**. North Queensland, Australia: James Cook University of North Queensland, 1992.

[43] ROE, B. P.; YANG, H.-J.; ZHU, J.; LIU, Y.; STANCU, I. ; MCGREGOR, G.. **Boosted decision trees as an alternative to artificial neural networks for particle identification**. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 543(2-3):577–584, may 2005.

[44] KOKLU, M.; OZKAN, I. A.. **Multiclass classification of dry beans using computer vision and machine learning techniques**. Computers and Electronics in Agriculture, 174:105507, 2020.

[45] SINGH, AMRIK. **Healthriskdata_ver_3**, 2019.

[46] DULAC-ARNOLD, G.; DENOYER, L.; PREUX, P. ; GALLINARI, P.. **Datum-wise classification: a sequential approach to sparsity**. In: JOINT EUROPEAN CONFERENCE ON MACHINE LEARNING AND KNOWLEDGE DISCOVERY IN DATABASES, p. 375–390. Springer, 2011.

[47] WIERING, M. A.; VAN HASSELT, H.; PIETERSMA, A.-D. ; SCHOMAKER, L.. **Reinforcement learning algorithms for solving classification problems**. In: 2011 IEEE SYMPOSIUM ON ADAPTIVE DYNAMIC PROGRAMMING AND REINFORCEMENT LEARNING (ADPRL), p. 91–96. IEEE, 2011.

[48] VAN HASSELT, H.; WIERING, M. A.. **Using continuous action spaces to solve discrete problems**. In: 2009 INTERNATIONAL JOINT CONFERENCE ON NEURAL NETWORKS, p. 1149–1156. IEEE, 2009.

[49] SILVER, D.; HUANG, A.; MADDISON, C. J.; GUEZ, A.; SIFRE, L.; VAN DEN DRIESSCHE, G.; SCHRITTWIESER, J.; ANTONOGLOU, I.; PANNEERSHELVAM, V.; LANCTOT, M. ; ET. AL. **Mastering the game of go with deep neural networks and tree search**. nature, 529(7587):484–489, 2016.

[50] SILVER, D.; SCHRITTWIESER, J.; SIMONYAN, K.; ANTONOGLOU, I.; HUANG, A.; GUEZ, A.; HUBERT, T.; BAKER, L.; LAI, M.; BOLTON, A. ; ET. AL. **Mastering the game of go without human knowledge**. nature, 550(7676):354–359, 2017.

[51] GUYON, I.; WESTON, J.; BARNHILL, S. ; VAPNIK, V.. **Gene selection for cancer classification using support vector machines**. Machine learning, 46(1):389–422, 2002.

[52] BISHOP, CHRISTOPHER M AND NASRABADI, NASSER M. **Pattern recognition and machine learning**, volumen 4. Springer, 2006.

[53] HUANG, X.; KHETAN, A.; CVITKOVIC, M. ; KARNIN, Z.. **Tabtransformer: Tabular data modeling using contextual embeddings**. arXiv preprint arXiv:2012.06678, 2020.

[54] ARIK, S. Ö.; PFISTER, T.. **Tabnet: Attentive interpretable tabular learning**. In: PROCEEDINGS OF THE AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE, volumen 35, p. 6679–6687, 2021.

[55] SOMEPALLI, G.; GOLDBLUM, M.; SCHWARZSCHILD, A.; BRUSS, C. B. ; GOLDSTEIN, T.. **Saint: Improved neural networks for tabular data via row attention and contrastive pre-training**. arXiv preprint arXiv:2106.01342, 2021.

[56] SHWARTZ-ZIV, R.; ARMON, A.. **Tabular data: Deep learning is not all you need**. Information Fusion, 81:84–90, 2022.

[57] GRINSZTAJN, L.; OYALLON, E. ; VAROQUAUX, G.. **Why do tree-based models still outperform deep learning on typical tabular data?** Advances in neural information processing systems, 35:507–520, 2022.

[58] NG, A. Y.. **Feature selection, l 1 vs. l 2 regularization, and rotational invariance**. In: PROCEEDINGS OF THE TWENTY-FIRST INTERNATIONAL CONFERENCE ON MACHINE LEARNING, p. 78, 2004.

[59] BORISOV, V.; LEEMANN, T.; SESSLER, K.; HAUG, J.; PAWELCZYK, M. ; KASNECI, G.. **Deep neural networks and tabular data: A survey**. IEEE transactions on neural networks and learning systems, 2022.

[60] RUBIN, D. B.. **Multiple imputation for nonresponse in surveys**, 1987.

[61] HEYMANS, M. W.; TWISK, J. W.. **Handling missing data in clinical research**. Journal of clinical epidemiology, 151:185–188, 2022.

[62] C, M.; Z, S. ; D, W.. **Types of missing data**. Technical report, Rockville (MD): Agency for Healthcare Research and Quality (US), February 2018.

[63] VAN BUUREN, S.. **Flexible imputation of missing data**. CRC press, 2018.

[64] GORISHNIY, Y.; RUBACHEV, I.; KHRULKOV, V. ; BABENKO, A.. **Revisiting deep learning models for tabular data**. Advances in Neural Information Processing Systems, 34:18932–18943, 2021.

[65] EFRON, B.; TIBSHIRANI, R. J.. **An introduction to the bootstrap**. Chapman and Hall/CRC, 1994.

[66] CHEN, L.; LU, K.; RAJESWARAN, A.; LEE, K.; GROVER, A.; LASKIN, M.; ABBEEL, P.; SRINIVAS, A. ; MORDATCH, I.. **Decision transformer: Reinforcement learning via sequence modeling**. Advances in neural information processing systems, 34:15084–15097, 2021.

[67] RADFORD, A.; NARASIMHAN, K.; SALIMANS, T.; SUTSKEVER, I. ; OTHERS. **Improving language understanding by generative pre-training**. 2018.

[68] DEVLIN, J.; CHANG, M.-W.; LEE, K. ; TOUTANOVA, K.. **Bert: Pre-training of deep bidirectional transformers for language understanding**. arXiv preprint arXiv:1810.04805, 2018.

[69] JAMIL, S.; JALIL PIRAN, M. ; KWON, O.-J.. **A comprehensive survey of transformers for computer vision**. Drones, 7(5):287, 2023.

[70] MUNOS, R.; STEPLETON, T.; HARUTYUNYAN, A. ; BELLEMARE, M.. **Safe and efficient off-policy reinforcement learning**. Advances in neural information processing systems, 29, 2016.

[71] JANNER, M.; LI, Q. ; LEVINE, S.. **Offline reinforcement learning as one big sequence modeling problem**. Advances in neural information processing systems, 34:1273–1286, 2021.

[72] CHEBOTAR, Y.; VUONG, Q.; HAUSMAN, K.; XIA, F.; LU, Y.; IRPAN, A.; KUMAR, A.; YU, T.; HERZOG, A.; PERTSCH, K. ; OTHERS. **Q-transformer: Scalable offline reinforcement learning via autoregressive q-functions**. In: CONFERENCE ON ROBOT LEARNING, p. 3909–3928. PMLR, 2023.

[73] ALAMMAR, J.. **The illustrated transformer [blog post]**. `https://jalammar.github.io/illustrated-transformer/`, 2018. Accessed: 2024-09-08.

[74] SCHULMAN, J.; WOLSKI, F.; DHARIWAL, P.; RADFORD, A. ; KLIMOV, O.. **Proximal policy optimization algorithms**. arXiv preprint arXiv:1707.06347, 2017.

[75] HESSEL, M.; MODAYIL, J.; VAN HASSELT, H.; SCHAUL, T.; OSTROVSKI, G.; DABNEY, W.; HORGAN, D.; PIOT, B.; AZAR, M. ; SILVER, D.. **Rainbow: Combining improvements in deep reinforcement learning**. In: PROCEEDINGS OF THE AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE, volumen 32, 2018.

[76] FANG, M.; LI, Y. ; COHN, T.. **Learning how to active learn: A deep reinforcement learning approach**. arXiv preprint arXiv:1708.02383, 2017.

[77] LIU, K.; FU, Y.; WU, L.; LI, X.; AGGARWAL, C. ; XIONG, H.. **Automated feature selection: A reinforcement learning perspective**. IEEE Transactions on Knowledge and Data Engineering, 35(3):2272–2284, 2023.

[78] KACHUEE, M.; GOLDSTEIN, O.; KARKKAINEN, K.; DARABI, S. ; SARRAFZADEH, M.. **Opportunistic learning: Budgeted cost-sensitive learning from data streams**. arXiv preprint arXiv:1901.00243, 2019.

[79] KARAYEV, S.; FRITZ, M. J. ; DARRELL, T.. **Dynamic feature selection for classification on a budget**. In: INTERNATIONAL CONFERENCE ON MACHINE LEARNING (ICML): WORKSHOP ON PREDICTION WITH SEQUENTIAL MODELS, volumen 95, 2013.

[80] KUMAR, A.; ZHOU, A.; TUCKER, G. ; LEVINE, S.. **Conservative q-learning for offline reinforcement learning**. Advances in Neural Information Processing Systems, 33:1179–1191, 2020.

[81] AGARWAL, R.; SCHUURMANS, D. ; NOROUZI, M.. **An optimistic perspective on offline reinforcement learning**. In: INTERNATIONAL CONFERENCE ON MACHINE LEARNING, p. 104–114. PMLR, 2020.

[82] DABNEY, W.; ROWLAND, M.; BELLEMARE, M. ; MUNOS, R.. **Distributional reinforcement learning with quantile regression**. In: PROCEEDINGS OF THE AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE, volumen 32, 2018.

[83] KOSTRIKOV, I.; NAIR, A. ; LEVINE, S.. **Offline reinforcement learning with implicit q-learning**. arXiv preprint arXiv:2110.06169, 2021.

[84] ERTIN, E.; PRIDDY, K. L.. **Reinforcement learning and design of nonparametric sequential decision networks**. In: APPLICATIONS AND SCIENCE OF COMPUTATIONAL INTELLIGENCE V, volumen 4739, p. 40–47. SPIE, 2002.

[85] JI, S.; CARIN, L.. **Cost-sensitive feature acquisition and classification**. Pattern Recognition, 40(5):1474–1485, 2007.

[86] NUNES, L. N.; KLUECK, M. M. ; FACHEL, J. M. G.. **Multiple imputations for missing data: a simulation with epidemiological data**. Cadernos de saude publica, 25:268–278, 2009.

[87] PRATAMA, I.; PERMANASARI, A. E.; ARDIYANTO, I. ; INDRAYANI, R.. **A review of missing values handling methods on time-series data**. In: 2016 INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY SYSTEMS AND INNOVATION (ICITSI), p. 1–6, 2016.

[88] JOEL, L. O.; DOORSAMY, W. ; PAUL, B. S.. **On the performance of imputation techniques for missing values on healthcare datasets**. arXiv preprint arXiv:2403.14687, 2024.

[89] ANDRIDGE, R. R.; LITTLE, R. J.. **A review of hot deck imputation for survey non-response**. International statistical review, 78(1):40–64, 2010.

[90] LALL, R.; ROBINSON, T.. **The midas touch: accurate and scalable missing-data imputation with deep learning**. Political Analysis, 30(2):179–196, 2022.

[91] PROKHORENKOVA, L.; GUSEV, G.; VOROBEV, A.; DOROGUSH, A. V. ; GULIN, A.. **Catboost: unbiased boosting with categorical features**. Advances in neural information processing systems, 31, 2018.

[92] KE, G.; MENG, Q.; FINLEY, T.; WANG, T.; CHEN, W.; MA, W.; YE, Q. ; LIU, T.-Y.. **Lightgbm: A highly efficient gradient boosting decision tree**. Advances in neural information processing systems, 30, 2017.

[93] NAN, F.; SALIGRAMA, V.. **Adaptive classification for prediction under a budget**. Advances in neural information processing systems, 30, 2017.

[94] ACAR, D. A. E.; GANGRADE, A. ; SALIGRAMA, V.. **Budget learning via bracketing**. In: INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE AND STATISTICS, p. 4109–4119. PMLR, 2020.

[95] STEKHOVEN, D. J.; BÜHLMANN, P.. **Missforest—non-parametric missing value imputation for mixed-type data**. Bioinformatics, 28(1):112–118, 2012.