Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

Transformer Approaches on Hyper-Parameter Optimization and Anomaly Detection With Applications in Stream Tuning

Nuno Filipe Marques Carriço

Orientador(es) | Paulo Miguel Quaresma

Évora 2022

Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

Transformer Approaches on Hyper-Parameter Optimization and Anomaly Detection With Applications in Stream Tuning

Nuno Filipe Marques Carriço

Orientador(es) | Paulo Miguel Quaresma

Évora 2022

A dissertação foi objeto de apreciação e discussão pública pelo seguinte júri nomeado pelo Diretor da Escola de Ciências e Tecnologia:

Presidente	|	Lígia Maria Ferreira (Universidade de Évora)
Vogais	|	Miguel José Barão (Universidade de Évora) (Arguente) Paulo Miguel Quaresma (Universidade de Évora) (Orientador)

Évora 2022

To all the dancing stars.

Contents

List of Figures

List of Tables

Acronyms

HPO Hyper-parameter Optimisation

MDP Markov Decision Process

RL Reinforcement Learning

TCN Temporal Convolution Network

RNN Recurrent Neural Network

CNN Convolutional Neural Network

LSTM Long Short-Term Memory

TGB-PRE Transformer Graph Base-Param Reward Encoding

TGR-PRE Transformer Graph RNN-Param Reward Encoding

TGB-PRC Transformer Graph Base-Param Reward Concat

TGR-PRC Transformer Graph RNN-Param Reward Concat

VAE Variational Auto Encoder

ECM Evolving Cluster Method

SC Silhouette Coefficient

PPO Proximal Policy Optimisation

ELBO Evidence Lower Bound

Abstract

Hyper-parameter Optimisation consists of finding the parameters that maximise a model's performance. However, this mainly concerns processes in which the model shouldn't change over time. Hence, how should an online model be optimised? For this, we pose the following research question: How and when should the model be optimised? For the optimisation part, we explore the transformer architecture as a function mapping data statistics into model parameters, by means of graph attention layers, together with reinforcement learning approaches, achieving state of the art results. On the other hand, in order to detect when the model should be optimised, we use the transformer architecture to empower already existing anomaly detection methods, in this case, the Variational Auto Encoder. Finally, we join these developed methods in a framework capable of deciding when an optimisation should take part and how to do it, aiding the stream tuning process.

Keywords: Transformer, Reinforcement Learning, Hyper-parameter Optimisation, Anomaly Detection, Variational Auto Encoder

Sumário

Abordagens de *Transformer* em Optimização de Hiper-Parâmetros e Deteção de Anomalias com Aplicações em *Stream Tuning*

Optimização de hiper parâmetros consiste em encontrar os parâmetros que maximizam a performance de um modelo. Contudo, maioritariamente, isto diz respeito a processos em que o modelo não muda ao longo do tempo. Assim, como deve um modelo *online* ser optimizado? Para este fim, colocamos a seguinte pergunta: Como e quando deve ser o modelo optimizado? Para a fase de optimização, exploramos a arquitectura de transformador, como uma função que mapeia estatísticas sobre dados para parâmetros de modelos, utilizando atenção de grafos junto de abordagens de aprendizagem por reforço, alcançando resultados de estado da arte. Por outro lado, para detectar quando o modelo deve ser optimizado, utilizamos a arquitectura de transformador, reforçando abordagens de detecção de anomalias já existentes, o *Variational Auto Encoder*. Finalmente, juntamos os métodos desenvolvidos numa *framework* capaz de decidir quando se deve realizar uma optimização e como o fazer, auxiliando o processo de *tuning* em *stream*.

Palavras chave: Transformador, Aprendizagem por Reforço, Optimização de Hiper parâmetros, Detecção de Anomalias, Variational Auto Encoder

1

Introduction

1.1 Objectives and Motivation

Hyper-parameters are a set of values that control the learning process, contrary to parameters which are derived from the data during training. That is, parameters are learnt, while hyper-parameters must be configured prior to training. Following this definition, Hyper-parameter Optimisation (HPO) is the problem of finding the set of hyper-parameters that provide the best performance for a model, given a dataset and a performance measure. However, this optimisation techniques maximize the performance over only the given dataset, and does not take into consideration future data shifts that may affect the model's performance. In such paradigms, where data is constantly changing its aspects, the previous found hyper-parameters rapidly lose their value, and we find ourselves in need of finding another set of hyper-parameters that better suite the current data. For instance, suppose we have an online matching system that matches users with job openings. Given that the job openings are mutable data, meaning they can be deleted, added or modified frequently, the model is likely to lose performance over-time. That is, the quality of the matches declines. Moreover, as new different openings are introduced in the system, the model is exposed to data never seen before, introducing concept drift problems. The same may happen with the users of the system, whose profiles keep being modified as users add new information, such as new courses, new interests or new skills.

In face of such problem, which we call stream tuning, we ask three research questions, that will, in turn, guide us trough this work:

- How should we optimise such processes?
- When should we optimise the process?
- How should we integrate the answers from the previous questions?

The first question poses itself naturally, since this is still an HPO problem. The second question comes into play to avoid unnecessary optimisations if the model is performing well with the current data, and the third appears in order to produce a framework capable of approaching such problems.

For the first problem, we use reinforcement learning methods assisted by a transformer architecture. Reinforcement Learning (RL) appears in this architecture for the following reasons: first and foremost, HPO is a sequential decision problem, which suggests a Markov Decision Process (MDP); secondly, we want to maximize the performance values over-time, which falls exactly on the problem RL tries to solve. Furthermore, we will obtain a policy that based on the dataset statistics, outputs the hyper-parameter set that is most likely to maximize the performance of the model over-time.

The transformer is useful to learn a mapping between dataset statistics and hyper-parameters, easing the navigation of the hyper-parameter space. The architecture in use, makes advantaged of graph attention mechanisms, to model the compatibility between variables as a dependency graph. We also experiment with different approaches to include the reward information: the first includes the reward value in the hyper-parameter set, intrinsically; the second processes the reward information individually, extrinsically. This method is then compared with state of the art HPO techniques, including BOHB, Hyperband, Random Search and TPE. For this purpose, we use a meta-dataset containing classification datasets ready to use and compare approaches using the average rank. Additionally, we compare the obtained results to the baseline and test our method with unseen datasets, proving it can generalize to similar datasets.

For the second problem, we propose to use unsupervised anomaly detection methods. For that we propose the already known normality learning methods, such as the Auto Encoder, and follow a reconstructive approach. That is, we use the reconstruction loss to determine if some sequence is an anomaly or not. Specifically, we will be using a variation of the auto encoder, the Variational Auto Encoder (VAE), that enables us to adopt probabilistic reconstruction measures, which follows an unsupervised setting. Moreover, we plan to enhance an already existing architecture to take more advantage of the available information and follow a more transformer like architecture, namely in the use of multi-head attention, rather than single head attention. Furthermore, we can reuse the transformer architecture developed in the HPO problem to address multivariate anomaly detection. This includes the use of a full transformer architecture that processes the sequence's features and time separately, and then combines this information before the encoding-decoding phase characteristic of the Auto Encoder.

In order to evaluate the proposed architectures, we train our model using a solar dataset, consisting of different meteorological variables, to assess the latent space representations and attention mechanisms plots. Here we show that the latent space exhibits a time structure and discuss how the anomaly presents itself in the attention plots. In addition, we discuss the quality of the anomalies detected based on the number of attention heads.

Finally, we intend to combine the previously developed methods to propose a framework that is able to detect anomalies in the model's performance, and, when said anomalies are detected, runs the trained policy to find a set of hyper-parameters that best suit the dataset seen by the model. In this case, we need to reformulate the Markov Decision Process differently from the normal optimisation case and define what

a normal sequence is to a sequence of model's performance values. Furthermore, we test our proposed framework in a clustering problem, against the baseline. Since we are dealing with an online approach, instead of using the average rank as an evaluation measure, we introduce the rolling average rank.

The development and training of the proposed models was devised using tensorflow [AAB+15] and tf-agents [HDV18].

1.2 Related Work

Grid search [LL19] represents the traditional method of HPO, making a throughout search over all possible hyper-parameter configurations of a given algorithm to optimise. Nonetheless, when considering a high dimensional search space, grid search does not present itself as the best option. This dimensionality problem can be atoned with a degree of parallelization.

On the other hand, random search [BB12] operates differently from grid search, in the sense that random search does not explore the hyper-parameter search space exhaustively. Instead, it randomly chooses hyper-parameter configurations drawn from a relevant hyper-parameter subspace. This method proved to have less efficiency in low dimensional spaces, yet, it achieves a much greater efficiency improvement in high dimensional spaces. In addition to discrete hyper-parameter spaces, this approach can be extended to continuous spaces or even mixed spaces [LL19].

Hyperband [LJD+18] is a bandit-based strategy capable of dynamically allocating resources to a set of random hyper-parameter configurations. This procedure relies heavily on early stopping strategies that avoid wasting resources on low performance configurations, allowing this approach to focus the resources in promising configurations. However, since it only samples configurations randomly, it does not learn from previous configuration evaluations, which, in turn, can have an impact on its performance [FKH18].

Contrary to Hyperband, TPE [BBKB11] is a bayesian optimisation method that uses kernel density estimators to model densities over the hyper-parameter input space, choosing, as the next configuration to evaluate, the configuration that maximizes the densities ratio. Additionally, by the use of kernel density estimators, TPE is capable of processing both discrete and mixed continuous hyper-parameter spaces [FKH18].

Combining both bayesian and bandit approaches, BOHB [FKH18] follows the main idea behind Hyperband to determine the number of configurations to evaluate with a given budget, yet it differs on how to select the next configuration. In this case, instead of randomly select the next configuration, BOHB introduces a model-based selection built with previously evaluated configurations information. The bayesian optimisation part of BOHB resembles TPE, using kernel density estimators to determine the next configuration and model the hyper-parameter space, with the exception that BOHB uses a single multidimensional kernel density estimator, while TPE opts for a hierarchy of one dimensional kernel density estimators [FKH18].

HypRL [JGST19] approaches the HPO problem as a RL problem. Here the agent learns to navigate the hyper-parameter space of a given model in a manner that maximizes the expected return over a given budget. The agent hyper-space traversal is based on the sequence of previous tested configurations and respective performances, modelled by a LSTM network, and making use of Q-learning techniques to determine which action to take in a given state.

[WCL20] follows the same principles of tackling the HPO problem though a RL framework. Specifically, they opt for a policy gradient approach, which enables their approach to consider continuous and mixed-continuous hyper-parameter spaces. At each episode the devised agent produces n normal distributions, one for each parameter, from where the parameter values are sampled. Furthermore, to avoid running the

algorithm to optimise, it is introduced the use of an auxiliary model to predict the algorithm's performance given a configuration. Such predictive model is trained with data gathered from the algorithm to optimise, however, its use introduces some noise into the policy training that should be taken into account as the training evolves.

RL approaches have also debuted in neural network architectures search. MetaQNN [BGNR17] succeeded in applying a Q-learning agent to discover Convolutional Neural Network (CNN) architectures that have good performance on a machine learning task, choosing the layers that compose the architecture in a sequential manner. Here the agent learns through random exploration, at the beginning, and eventually learns how to exploit to find higher performant models, following an ϵ-greedy strategy. The validation accuracy is used as the reward signal for the agent. Going further, ReMAADE [KGN+20] was successful in both the neural architecture search and HPO tasks. This approach combines the REINFORCE algorithm with masked attention auto-regressive estimators, using a transformer like architecture to specify the policy network.

Evolutionary algorithms also play a role in HPO. For instance, [YRK+15] used a genetic algorithm to optimise hyper-parameters of a CNN. Additionally, [RMS+17], proposed an evolutionary algorithm based only in mutations that builds the neural network bit by bit trying to produce a good hyper-parameter configuration. Finally, [XYB+20], addresses the HPO problem with a variable length genetic algorithm capable of finding a parameter configuration that optimizes a CNN. In addition, the proposed approach incorporates cross-over operations as a way to achieve better efficiency when compared to the previous approaches.

1.3 Thesis Outline

We begin by providing the necessary concepts for the thesis in chapter 2. Namely, the basics of RL, attention mechanisms and Auto Encoders. Furthermore, we introduce the auxiliary networks used in this thesis, such as the Temporal Convolution Network (TCN).

Regarding the first question posed, we address it in chapter 3 by first providing a MDP formulation of the problem and then describe the tested architectures and training methods. We finalize the chapter by presenting and discussing the obtained results.

Chapter 4 concerns the development of the tools necessary for the anomaly detection. Here it is presented a discussion of the developed architectures, the univariate and multivariate transformer like architectures. In addition, we discuss how the anomalies are detected and proceed to describe the network training. The chapter ends with a discussion of the latent space representations, attention mechanisms plots and the anomalies detected in both the univariate case and multivariate case.

Finally, in chapter 5, we combine the previous developed methods to provide a stream tuning framework. The chapter starts with a re-formulation of the initial MDP, following an anlysis of the data statistics attention plots and ending with the comparison to the baseline model.

The thesis ends in chapter 6, where we provide a review of the main aspects of the work. Moreover, we include a discussion of some of the main problems left to solve and possible improvements and extensions of this work. Additionally, we include some questions that appeared through the development of this work and we consider pertinent to answer in the future.

2

Background

2.1 Reinforcement Learning

Throughout this section we will be referring to [SB18] as our basis to provide background on the RL task.

2.1.1 Markov Decision Processes

Agent-Environment Interaction

A MDP provides us with a framework for the problem of learning through interaction with the environment in order to achieve a goal. The one that interacts with the environment and learns to make decisions it's called the agent. The object the agent interacts with, is called the environment. These interact in turn, continually: the agent picks an action and the environment responds to it by presenting a different setting to the agent. Also, the environment sends reward signals to agent, which the latter tries to maximize, over time, through its choice of actions.

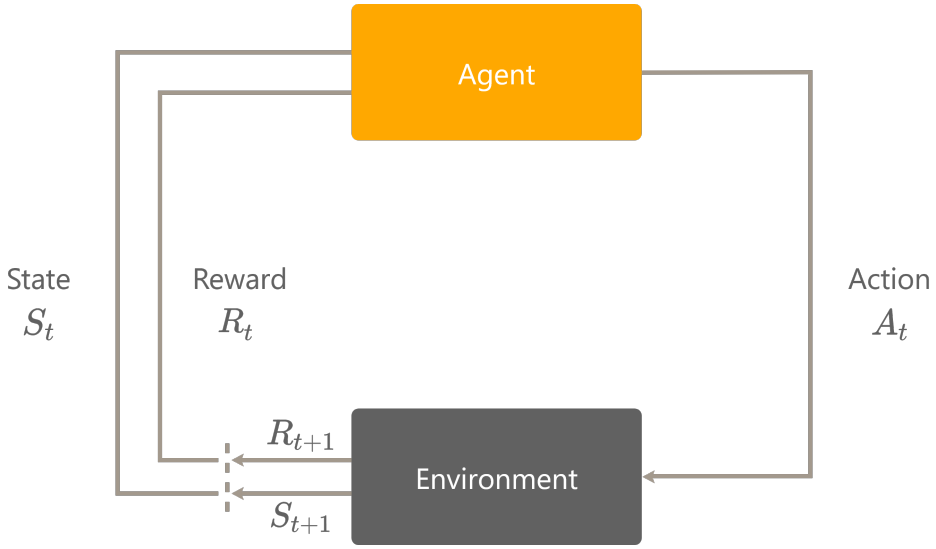


Figure 2.1: Agent Environment Interaction

Formally, the agent and the environment interact in turn during a sequence of discrete time steps, $t = 0, 1, 2, 3, \ldots$, and, at each time step t, the agents receives a representations of the environment's state, $S_t \in \mathcal{S}$. Based on this representation, the agent chooses an action, $A_t \in \mathcal{A}(s)$, and, one time step later, $t+1$, as a result of its action, a reward is emitted, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$. Moreover, a new state S_{t+1} is provided to the agent. These interactions produce a sequence or trajectory as follows:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$$

In case of working with a finite MDP, the set of states, actions and rewards, denoted $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, has a finite number of elements. Also, the probabilities of the random variables R_t and S_t have well defined discrete probability distributions, which are only dependent of the previous state and action. In other words, for given values of reward and state, $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, the probability of these occurring at time t, based on the previous state and action is:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.1}$$

for all $s', s \in \mathcal{S}, r \in \mathcal{R}$ and $a \in \mathcal{A}(s)$.

The function p describes the dynamics of the MDP. The dynamics function p: $\mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0; 1]$ is a deterministic function with four arguments. Also, it satisfies a probability distribution for each choice of s and a, that is:

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1, \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{2.2}$$

It should be noted that the state must contain all the relevant information of past interactions that are of importance for future decisions. If this is true, then the state is said to have the Markov property.

Using the dynamics function, p, we can derive more information about the environment, as an example,

the state-transition probabilities, which are given by:

$$p(s'|s,a) = Pr\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{r\in\mathcal{R}} p(s', r|s, a) \tag{2.3}$$

Furthermore, we can also obtain the expected rewards for the state-action pairs

$$r(s,a) \doteq \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p(s', r|sa) \tag{2.4}$$

and the expected rewards for the state-action-next-state triples

$$r(s,a,s') \doteq \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r\in\mathcal{R}} r\frac{p(s', r|s, a)}{p(s'|s, a)} \tag{2.5}$$

Rewards and Episodes

Generally, it is in our interest to maximize the expected return, denoted as G_t. The return is defined as function of the reward sequence. Its simplest case takes form as a sum of rewards:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T,$$

where T is a final time step.

This method can be of use in applications where there is a notion of final time step, strictly speaking, when the agent's interaction with the environment can be separated into sub-sequences, which are called episodes. The state in which each episode ends is called the terminal state. After the terminal state is reached, the episode starts anew from a standard starting state or from a starting state sampled from a distribution of starting states. For this reason, we can consider that the episodes all end in the same terminal state but with different rewards for different outcomes. Tasks with this kind of episodes are called episodic tasks. The time at each episode terminates, T, is a random variable that can vary from episode to episode.

On the other hand, there are cases that cannot be broken down into episodes, and go on continually without limit. These are called continuing tasks. Following the previous formulation of return, we can face problems of infinite returns, since the final time step would be $T = \infty$. As an alternative, we should consider the concept of discounted return. In this case, the agent tries to select actions that maximizes the sum of the discounted rewards received over time. Formally, the agent chooses A_t to maximize the expect discount return:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.6}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the discount rate.

The discount rate determines the present value of future rewards: a reward that is to be received k time steps into the future is worth y^{k-1} times what it would be worth if it were to be received immediately. If

$\gamma < 1$, the infinite sum has a finite value given the reward sequence $\{R_k\}$ is bounded. If $\gamma = 0$, the agent is said to be myopic, since it is only concerned with maximizing the immediate rewards. As γ approaches 1, the return objective takes future rewards into more consideration. Successive returns can be related in the following manner:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots \tag{2.7}$$
$$= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots) \tag{2.8}$$
$$= R_{t+1} + \gamma G_{t+1} \tag{2.9}$$

Policies and Value Functions

The majority of reinforcement learning algorithms involve estimating value functions - functions of states that estimate how good it is for the agent to be in a given state. These functions are defined with respect to particular ways of acting, called policies.

Formally, a policy maps states to the probabilities of selecting each possible action. If the agent is following a policy π at time t, then $\pi(a|s)$ is the probability of taking $A_t = a$ given $S_t = s$. Reinforcement learning methods describe how the agent's policy should change based on its experience.

The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π afterwards. For MDPs, we can define v_π by:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] \tag{2.10}$$

for all $s \in \mathcal{S}$, where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agents follows policy π, and t is any time step. The function v_π is called the state-value function for policy π.

In a similar manner, we can define the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from state s, taking the action a, and thereafter following policy π :

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{2.11}$$

The q_π is called the action-value function for policy π. These value functions v_π and q_π can be estimated from experience.

Given any policy π and any state s, the following condition holds between the value of s and the value of its possible successor states :

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s] && (2.12)\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] && (2.13)\\
&= \sum_{a\in\mathcal{A}(s)} \pi(a|s) \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\right] && (2.14)\\
&= \sum_{a\in\mathcal{A}(s)} \pi(a|s) \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] && (2.15)
\end{aligned}
$$

for all $s \in \mathcal{S}$. Equation 2.15 is the Bellman equation for v_π. It expresses the relationship between the value of a state and the values of its successor states.

Optimal Policies and Value Function

Considering the case of finite MDPs, we can define an optimal policy in the following manner. Value functions can be viewed as a partial ordering over the policies, furthermore, we can say that a policy π is better than a policy π' if its expected return is greater than or equal to the expected return of π' for all states. That is, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other polices, the optimal policy, denoted by π_*. There can be more than one optimal policy, which share the same value function, called the optimal state-value function, denoted by v_*, and defined as:

$$v_*(s) \doteq \max_\pi v_\pi(s) \tag{2.16}$$

for all $s \in \mathcal{S}$. In addition, optimal policies also share the same optimal action-value function, denoted q_*, and defined as:

$$q_*(s, a) \doteq \max_\pi q_\pi(s, a) \tag{2.17}$$

for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$. For the state action pair (s, a), this function gives the expected return for taking action a in state s and thereafter following an optimal policy. We can then write q_* in terms of v_*:

$$q_*(s, a) = \mathbb{E}\left[R_{t+1} + \gamma v_*(s_{t+1})|S_t = s, A_t = a\right] \tag{2.18}$$

As v_* is the value function for a policy, it must satisfy the Bellman equation for state values 2.15. Moreover, since it is the optimal value function, the equation 2.15 can be written without any specific policy, as such:

$$v_*(s) = \max_{a \in \mathcal{A}(s)} q_*(s,a) \tag{2.19}$$
$$= \max_a \mathbb{E}_{\pi_*} \left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a \right] \tag{2.20}$$
$$= \max_a \mathbb{E}_{\pi_*} \left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a \right] \tag{2.21}$$
$$= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{2.22}$$

The last two equations 2.22 are two forms of the Bellman optimality equation for v_*. The Bellman optimality equation for q_* is:

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a \right] \tag{2.23}$$
$$= \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{2.24}$$

The Bellman optimality equation for v_* has a unique solution in the case of finite MDPs. From the moment we have v_*, we can easily determine an optimal policy: for each state s there will one or more actions at which the maximum is obtained in the Bellman optimality equation, thus, any policy that attributes a nonzero probability only to these actions is an optimal policy. Trough v_*, the optimal expected long-term return can be obtained locally and in each state. Alternatively, using q_* to choose optimal actions, for any state s we can simply find the action that maximizes $q_*(s,a)$, without the need to search one step ahead for the optimal actions.

Explicitly solving the Bellman optimality equation provides a way to find an optimal policy. However, this solution is rarely useful. This solution is based on at least three assumptions that are not usually true in practice :

1. the dynamics of the environment is accurately known;
2. computational resources are sufficient to complete calculations;
3. the states have the Markov property.

2.1.2 On Policy and Off Policy Methods

On-policy approaches essentially try to improve the same policy that is used to generate data, whereas off-policy methods separate this two tasks.

Regarding on-policy methods, the policy tends to be soft. That is, $\pi(a|s) > 0$ for all $s \in \mathcal{S}$ and all $a \in \mathcal{A}$, but gradually shifts to a deterministic policy. This leads to a problem in control methods: they try to learn action values conditional on subsequent optimal behaviour, but they need to behave non-optimally to explore all actions. For this, one can use two policies, one that is improved to reach optimal results, and one to generate behaviour. The first is called the target policy, while the latter is called the behaviour policy. In such setting, we are learning from data "off" the target policy, thus overall process is termed off-policy learning.

2.1.3 Policy Gradient Methods

Regarding policy gradient methods, we learn a parameterized policy that is able to select actions without referring a value function. However, a value function can be still useful to learn the policy parameter, but it is not required for action selection. The policy's parameter vector is denoted by $\theta \in \mathbb{R}^{d'}$. Thus we write $\pi(a|s,\theta) = Pr\{A_t = a|S_t = s, \theta_t = \theta\}$ for the probability that action a is taken at time t given that the environment is in state s at time t with parameter θ. In case of a method using a learned value function, we denote the value function's weight vector by $w \in \mathbb{R}^d$, as in $\hat{v}(s,w)$.

Here we consider methods for learning the policy parameter based on the gradient of some scalar performance measure $J(\theta)$ with respect to the policy parameter. These methods seek to maximize performance, so their updated approximate gradient ascent in J :

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)} \tag{2.25}$$

where $\widehat{\nabla J(\theta_t)} \in \mathbb{R}^{d'}$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument θ.

The policy in policy gradient methods can be parametrized in any way we want. Nonetheless, we have to guarantee that $\pi(a|s,\theta)$ is differentiable with respect to its parameters, that is, as long as $\nabla\pi(a|s,\theta)$ exists and is finite for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, and $\theta \in \mathbb{R}^{d'}$. Additionally, to favor exploration, it is required that the policy never becomes deterministic.

Parameterizing the policy can have some advantages, namely, the policy may be simpler to approximate and the choice of parameterization can be useful to inject prior knowledge about the desired form of the policy.

2.1.4 REINFORCE

Classic REINFORCE

The policy gradient theorem establishes a way to approximate the gradient of the performance with respect to the policy parameter. Additionally, this does not take into account the derivative of the state distribution, which, in episodic cases, resolves itself to the following expression.

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a)\nabla\pi(a|s,\theta) \tag{2.26}$$

where $\mu(s)$ is the state distribution, π is the policy correspondent to parameter vector θ and q_π is the action-value function under π. Note that we just obtained an expression directly proportional to the gradient. Thus, it remains to find a way of sampling whose expectation equals or approximates this expression. Looking at the right side of the policy gradient theorem, we have a weighted sum over states under a target policy π; if π is followed, then states will be encountered in these proportions. Thus

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) \tag{2.27}$$

$$= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\theta) \right] \tag{2.28}$$

We can now apply the stochastic gradient-ascent as

$$\theta_{t+1} \doteq \theta_t + \alpha \sum_a \hat{q}(S_t,a,\mathsf{w}) \nabla \pi(a|S_t,\theta) \tag{2.29}$$

where $\hat{q}$ is some learned approximation to q_π. This algorithm updates involves all actions, however, the classic REINFORCE algorithm only considers the action taken at time t, A_t. Modifying the above equation to consider only the action at time t, we have

$$\nabla J(\theta) \propto \mathbb{E}_\pi \left[\sum_a \pi(a|S_t,\theta) q_\pi(S_t,a) \frac{\nabla \pi(a|S_t,\theta)}{\pi(a|S_t,\theta)} \right] \tag{2.30}$$

$$= \mathbb{E}_\pi \left[q_\pi(S_t,a) \frac{\nabla \pi(a|S_t,\theta)}{\pi(a|S_t,\theta)} \right] \text{ considering only the action at time } t \tag{2.31}$$

$$= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(a|S_t,\theta)}{\pi(a|S_t,\theta)} \right] \tag{2.32}$$

where G_t is the usual return. The expression in brackets is a quantity that can be sampled on each time step and whose expectation is proportional to the gradient. Again, applying stochastic gradient-ascent, yields the REINFORCE update rule :

$$\theta_{t+1} \doteq \theta_t + \alpha G_t \frac{\nabla \pi(a|S_t,\theta)}{\pi(a|S_t,\theta)}$$

Each increment is proportional to the product of a return G_t and a vector, the gradient of the probability of taking the action actually taken divided by the probability of taking action. The vector points in the direction of the parameter space in which the probability of repeating action A_t on future visits increases the most. The parameter vector is increased in this direction proportionally to the return, and inversely proportional to the action probability. The proportional increase makes the parameter to move most in the direction that benefits actions which result in the highest return. The inversely proportional portion of it penalizes actions that are selected more frequently. We are now in condition to provide the classic

REINFORCE algorithm 1.

Algorithm 1: REINFORCE

Input : A differentiable policy parametrization $\pi(a|s,\theta)$
Algorithm parameter: Step size $\alpha > 0$

Initialize policy parameter $\theta \in \mathbb{R}^{d'}$

Loop *forever (for each episode)*
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\theta)$
  Loop *for each step of the episode* $t = 0, 1, \ldots, T-1$
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$;
    $\theta \leftarrow \theta + \alpha \gamma^t G \nabla \log \pi(A_t|S_t,\theta)$;

Baseline REINFORCE

We can incorporate in the policy gradient theorem, equation 2.26, a comparison of the action value to an arbitrary baseline $b(s)$:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a (q_\pi(s,a) - b(s)) \nabla \pi(a|s,\theta) \tag{2.33}$$

We can use any function as the baseline, if it does not depend on a. The equation continues to make sense since the subtracted quantity is zero:

$$\sum_a b(s) \nabla \pi(a|s,\theta) = b(s) \nabla \sum_a \pi(a|s,\theta) = b(s) \nabla 1 = 0 \tag{2.34}$$

Following the procedure in the previous section, we can use the policy gradient theorem to derive an update rule. Including the baseline we have:

$$\theta_{t+1} \doteq \theta_t + \alpha (G_t - b(S_t)) \frac{\nabla \pi(a|S_t,\theta)}{\pi(a|S_t,\theta)} \tag{2.35}$$

The baseline does not affect the expected value of the update, but it can largely impact the variance.

A standard way to chose the baseline is as an estimate of the state value, $\hat{v}(S_t, w)$, where $w \in \mathbb{R}^d$ is a

learnable weight vector. The algorithm is as follows :

Algorithm 2: REINFORCE with Baseline

Input : A differentiable policy parameterization $\pi(a|s,\theta)$
Input : A differentiable state-value function parameterization $\hat{v}(s,w)$
Algorithm parameter: Step size $\alpha^{\theta} > 0,\ \alpha^{w} > 0$

Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^{d}$

Loop *forever (for each episode)*
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\theta)$
  Loop *for each step of the episode* $t = 0, 1, \ldots, T-1$
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$;
    $\delta \leftarrow G - \hat{v}(S_t, w)$;
    $w \leftarrow w + \alpha^{w} \delta \nabla \hat{v}(S_t, w)$;
    $\theta \leftarrow \theta + \alpha^{\theta} \gamma^{t} \delta \nabla \log \pi(A_t|S_t,\theta)$;

2.1.5 Actor Critic Methods

Regarding actor-critic [BSGL09] methods, we also apply the state-value function to the second state of the transition. Thus, the estimated value of second state, when properly discounted and added to the reward, is the one-step return, $G_{t:\,t+1}$. This estimate of the actual return is useful to access the action which lead to the second state. In such a case, the state-value function is called a critic, and the overall policy-gradient method is termed an actor-critic method. Thus, the return of one-step actor-critic methods is obtained as follows:

$$\theta_{t+1} \doteq \theta_t + \alpha(G_{t:\,t+1} - \hat{v}(S_t, w)) \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \tag{2.36}$$

$$= \theta_t + \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)) \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \tag{2.37}$$

$$= \theta_t + \alpha \delta_t \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \tag{2.38}$$

Giving rise to the algorithm 3.

Algorithm 3: Actor-Critic

Input : A differentiable policy parameterization $\pi(a|s,\theta)$
Input : A differentiable state-value function parameterization $\hat{v}(s,w)$
Algorithm parameter: Step size $\alpha^{\theta} > 0,\ \alpha^{w} > 0$

Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^{d}$

Loop *forever (for each episode)*
  Initialize S (first state episode) ;
  $I \leftarrow 1$;
  Loop *while S is not terminal (for each time step)*
    $A \sim \pi(\cdot|S,\theta)$;
    Take action A, observe $S',\ R$;
    $\delta \leftarrow R + \gamma\hat{v}(S',w) - \hat{v}(S,w)$;
    $w \leftarrow w + \alpha^{w}\delta\nabla\hat{v}(S,w)$;
    $\theta \leftarrow \theta + \alpha^{\theta} I\delta\nabla \log\pi(A|S,\theta)$;
    $I \leftarrow \gamma I$;
    $S \leftarrow S'$

2.1.6 Proximal Policy Optimisation

Proximal Policy Optimisation (PPO) optimizes the main objective [SWD+17]:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right] \tag{2.39}$$

where epsilon is a hyper parameter. This comes with the following benefits: we can ignore the change in probability ratio, if the objective were to improve, and, include the change in the probability ratio if it makes the objective worse [SWD+17].

A majority of the techniques which compute variance-reduced advantage-functions estimators uses the learned state-value function, $V(s)$, to do so [SWD+17]. Additionally, in case of using a neural network that shares parameters between the policy and the value function, we must take these factors into account and adapt the loss function to consider the policy surrogate and a value function error term [SWD+17]. Therefore, we consider the following objective to maximize:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right] \tag{2.40}$$

where c_1, c_2 are coefficients, S denotes an entropy bonus, and L_t^{VF} is a squared-error loss $(V_\theta(s_t) - V_t^{\text{targ}})^2$. One style of policy gradient implementation, runs the policy for T timesteps, and uses the collected samples for an update. This style requires an advantage estimator that does not look beyond timestep T [SWD+17]. A generalized advantage estimation is given by:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{2.41}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$.

In each iteration, each of the N actors collect T timesteps of data. After that, we compute the surrogate loss on these NT timesteps of data, and we optimized it using minibatch SGD [Rud17] for K epochs [SWD+17]. An algorithm is provided in 4 [SWD+17].

Algorithm 4: PPO Algorithm

Input : A differentiable policy parameterization $\pi(a|s,\theta)$
Input : Number of parallel actors N
Input : Minibatch size M
Input : optimisation epochs K
For *iteration* $= 1, 2, \ldots$
  For *actor* $= 1, 2, N$
    Run policy $\pi(\theta_{\text{old}})$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$

2.2 Anomaly Detection

2.2.1 Normality Feature Learning

This type of methods aims to learn representations of data by means of optimizing a feature learning objective function, which will, by turn, force these methods to capture key data regularities that can take part in anomaly detection [PSCH21]. Formally,

$$\{\Theta^*, \mathsf{W}^*\} = \text{argmin}_{\Theta,\mathsf{W}} \sum_{\mathsf{x}\in\mathcal{X}} \ell(\psi(\phi(\mathsf{x};\Theta);\mathsf{W})) \tag{2.42}$$

$$s_x = f(\mathsf{x}, \phi_{\Theta^*}, \psi_{\mathsf{W}^*}) \tag{2.43}$$

where ϕ, parameterized by Θ, maps the original x onto the representation space $\mathcal{Z}$, ψ, parameterized by W, is a surrogate learning task that operates on the $\mathcal{Z}$ space and enforces the learning of underlying data regularities, ℓ is a loss function and f is a scoring function that utilizes ϕ and ψ to compute the anomaly score s [PSCH21].

2.2.2 Autoencoders

Autoencoders learn a low-dimensional feature representation space from which the data instances can be well reconstructed [PSCH21]. Considering the anomaly detection task, this approach, by learning the important features of data, will have problems reconstructing anomalies from the learned representations. That is, normal data instances will have accurate low-dimensional representations, producing low reconstruction errors, while anomalous data representations won't be as accurate and, thus, producing high reconstruction errors.

Autoencoders are composed of an encoding network, which maps the data instances to a low dimensional representation, and a decoding network, responsible for reconstructing the data from the representation space instance. These networks are trained by minimizing a reconstruction loss function between the

original data and the reconstructed data [PSCH21]. Formally,

$$\mathsf{z} = \phi_e(\mathsf{x}; \Theta_e),\ \hat{\mathsf{x}} = \phi_d(\mathsf{z}; \Theta_d) \tag{2.44}$$

$$\{\Theta_e^*, \Theta_d^*\} = \mathsf{argmin}_{\Theta_e, \Theta_d} \sum_{\mathsf{x} \in \mathcal{X}} \|\mathsf{x} - \phi_d(\phi_e(\mathsf{x}; \Theta_e); \Theta_d)\|^2 \tag{2.45}$$

$$s_{\mathsf{x}} = \|\mathsf{x} - \phi_d(\phi_e(\mathsf{x}; \Theta_e^*); \Theta_d^*)\|^2 \tag{2.46}$$

where ϕ_e is the encoding network with parameters Θ_e and ϕ_d is the decoding network with the parameters Θ_d, s_{x} is the reconstruction error-based anomaly score of x [PSCH21].

2.2.3 Variational Autoencoder

The VAE [KW14] follows the same architecture of the typical Autoencoder, being composed of an encoder and decoder network. This approach differs from the traditional Autoencoder by restricting the low-dimensional representation to a random variable. Such random variable follows a prior distribution $p_\theta(z)$, commonly, the standard Normal distribution, $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$. However, the problem of learning the true posterior distribution, $p_\theta(z|x)$, is most of the time impossible [KW19]. Thus, we instead use an approximation of such distribution, $q_\phi(z|x)$, using the parameters produced by the encoder network. In case of considering a Normal distribution, said parameter are the mean μ_z and variance σ_z^2. As such, the training objective of the VAE is to maximize the evidence lower bound (ELBO), lower bounded on the training data log-likelihood [KW19]. For a training instance x, the ELBO is obtained by the following expression:

$$\mathcal{L}(\theta, \phi; x) = -D_{KL}(q_\phi(z|x) \| p_\theta(z)) + \mathsf{E}_{q_\phi(z|x)} \left[\log p_\theta(x, |z)\right] \tag{2.47}$$

where θ and ϕ are the parameters of the encoder and decoder, respectively.

2.3 Attention

An attention function can be thought as a mapping between a query and a set of key-value pairs to an output, where the query, key, value and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key [VSP+17]. We will discuss some of the approaches taken to compute attention weights and output vectors.

2.3.1 Bahdanau Attention

Following the work of [BCB16], the output vector, c_i is computed by:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{2.48}$$

where T_x corresponds to the length of the input sequence. Here the weights α_{ij} of each input representation h_j can be obtained using the following expression:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ij})} \tag{2.49}$$

and

$$e_{ij} = a(s_{i-1}, h_j) \tag{2.50}$$

where s_{i-1} is the previous hidden state. Considering equation 2.50 the e_{ij}, also called the associated energy of the weights α_{ij}, scores how compatible the inputs around the j-th position and the output at the i-th position are.

From another perspective, the weights α_{ij} can be thought to describe how important the representation h_j is with respect to the previous hidden state s_{i-1}, when it comes to obtain the next state s_i and generating the output y_i [BCB16]. This behaviour is captured by the alignment function, or compatibility function, a. In this case, the compatibilty function is computed with a single-layer multilayer perceptron, such that:

$$a(s_{i-1}, h_j) = v_a^\top \tanh(W_a s_{i-1} + U_a hj) \tag{2.51}$$

where $W_a \in \mathbb{R}^{n \times n}, U_a \in \mathbb{R}^{n \times 2n}$ and $v_a \in \mathbb{R}^n$ are weight matrices. Considering how the compatibility function is computed, Bahdanau's attention is also denoted by additive attention.

2.3.2 Scaled Dot Product Attention

A different approach to attention, provided by [VSP+17], focus on the dot-product operation. The input is split into queries, keys and values, where the queries and keys have dimension d_k and the values dimension d_w. The procedure to obtain the weights works as follows. First the dot product between the queries with all keys is taken, followed by a division by $\sqrt{d_k}$. It remains to apply a softmax function, in order to obtain the weights on the values.

Practically, the queries, keys and values are packed into matrices Q, K and V, respectively. The output matrix is then obtained by:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{2.52}$$

Note that, this method of computing attention is identical to dot-product attention, if the scaling factor is removed. However, the scaling factor $\frac{1}{\sqrt{d_k}}$ comes into play when the dot-product grows larger in magnitude, which will, in turn, pull the softmax function back into regions where the gradient is not small. In contrast, if the scaling factor is not taken into account, the softmax function gradient becomes small, which causes additive attention to outperform dot-product attention for larger values of d_k [VSP+17].

A major benefit of this type of attention, when compared to additive attention, comes from an implementation perspective. The dot-product can be implemented using highly optimized matrix multiplication code, proving to be faster and more space-efficient than additive attention.

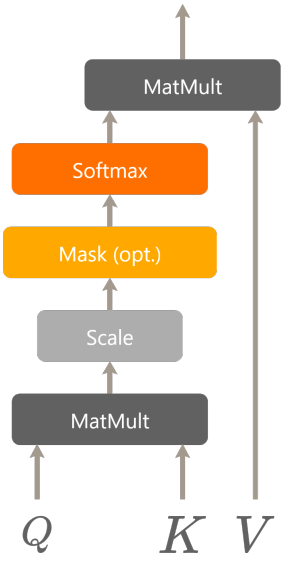


Figure 2.2: Scaled Dot-Product Attention, adapted from [VSP+17]

2.3.3 Multi-Head Attention

The previous section concerns only a single attention function. Nonetheless, linearly projecting the queries, keys and values h times with different linear projections d_k, d_k, and d_v dimensions, respectively, can prove itself beneficial. The attention function is applied individually to each of these projected versions, yielding a d_v-dimensional output values [VSP+17]. The latter is concatenated and further projected, providing us with the final values.

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions, which is inhibited by a single attention head [VSP+17]. Finally, we end up with

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{2.53}$$

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2.54}$$

The projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.

2.3.4 Graph Attention

Graph attention mechanisms are well suited for graph-structured data, offering an efficient solution, since it its parallelizable across node-neighbor pairs [VCC+18]. For the input we consider a set of node features,

$$\text{h} = \{\vec{h_1}, \vec{h_2}, \ldots, \vec{h_N}\}, \vec{h_i} \in \mathbb{R}^F$$

where N is the number of nodes, and F is the number of output features in each node. The attention layer will then produce a new set of node features, that may be different in number,

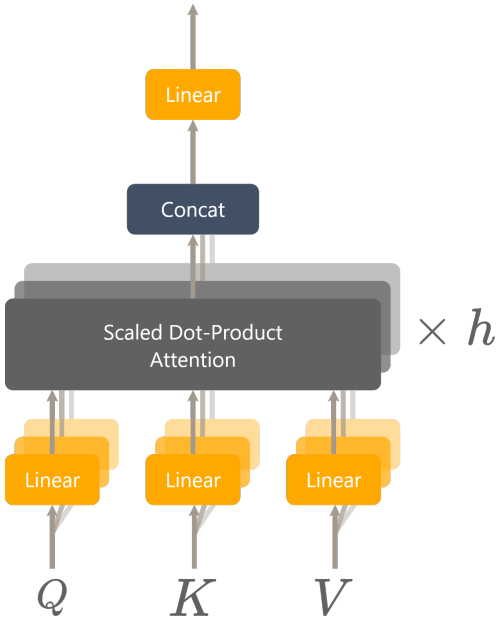


Figure 2.3: Multi-Head Attention, adapted from [VSP+17]

$$\mathsf{h}' = \{\vec{h'_1}, \vec{h'_2}, \ldots, \vec{h'_N}\}, \vec{h'_i} \in \mathbb{R}^{F'}.$$

The first step is to transform the input features into higher-level features. As a way to achieve this, a shared linear transformation, $\mathsf{W} \in \mathbb{R}^{F' \times F}$ is applied to all nodes, followed by a self-attention operation to produce the attention coefficients.

$$e_{ij} = a(\mathsf{W}\vec{h_i}, \mathsf{W}\vec{h_j}) \tag{2.55}$$

Here the values e_{ij} relate the importance of node's j features with respect to node i. Furthermore, since the purpose of this mechanism is to be applied over graphs, a masked attention is taken into consideration, computing the values e_{ij} only for nodes in the neighborhood of node i, denoted $\mathcal{N}_i$, in the graph. The weighs are finally obtained via a softmax function on the values e_{ij}:

$$\alpha_{ij} = \mathsf{softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})} \tag{2.56}$$

In this case, the compatibility function, a, is a single-layer feed-forward neural network, $\vec{\mathsf{a}} \in \mathbb{R}^{2F'}$ followed by a LeakyReLU function. Expanding equation 2.56, we are left with:

$$\alpha_{ij} = \mathsf{softmax}(e_{ij}) = \frac{\mathsf{LeakyReLU}\left(\vec{\mathsf{a}}^\top \left[\mathsf{W}\vec{h_i} \| \mathsf{W}\vec{h_j}\right]\right)}{\sum_{k \in \mathcal{N}_i} \mathsf{LeakyReLU}\left(\vec{\mathsf{a}}^\top \left[\mathsf{W}\vec{h_i} \| \mathsf{W}\vec{h_k}\right]\right)} \tag{2.57}$$

where $\|$ denotes the concatenate operation. Having the attention weights, the final output features for

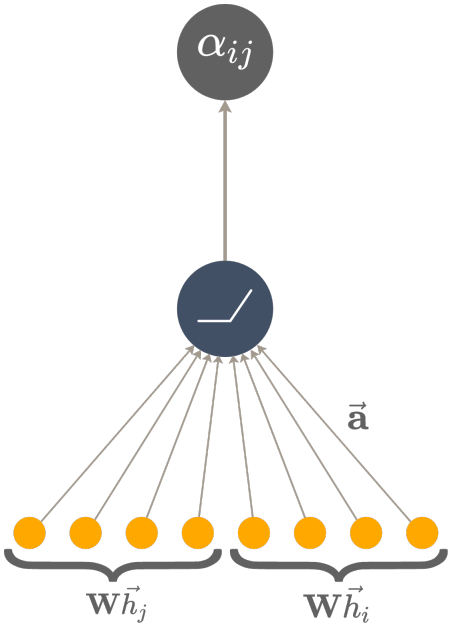


Figure 2.4: Graph Attention Mechanism, adapted from [VCC+18]

every node is computed by the expression:

$$\vec{h_i} = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathsf{W}\vec{h_j}\right) \tag{2.58}$$

Following [VSP+17], the mechanism can be extended to multi-head attention. Having K independent attention mechanisms executing the equation 2.58 and concatenating the resulting features, the output is given by:

$$\vec{h_i} = \Big\|_{k=1}^{K} \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^{k} \mathsf{W}^{k}\vec{h_j}\right) \tag{2.59}$$

where $\|$ represents concatenation, α_{ij}^{k} are the attention weights computed by the k-th attention mechanism (a^k), and W^k is the corresponding input linear transformation's weight matrix [VCC+18]. The output will then consist of KF' features, rather than F', for each node.

2.3.5 Variational Attention

Variational Attention builds itself on top of the deterministic attention (scaled dot product attention, for instance), from which we obtain the context vectors. We define the prior distribution over the context vectors to be the standard Normal, $p(\mathsf{c}_t) = \mathcal{N}(0, \mathsf{I})$ [BMVP18]. The variational parameters of the approximate posterior of the context vectors, $\tilde{q}_\phi^a(\mathsf{c}_t|\mathsf{x})$, mean μ_{c_t}, and standard deviation σ_{c_t} are obtained by applying two fully connected layers. Thereafter, we sample the final context vectors from the approximate posterior, $\mathsf{c}_t \sim \mathcal{N}(\mu_{\mathsf{c}_t}, \sigma_{\mathsf{c}_t})$ [BMVP18].

2.4 Auxliary Networks

2.4.1 LSTM

Long Short-Term Memory (LSTM) networks [HS97] come into play to solve the vanishing and exploding gradient problems by introducing memory cells and gates. These gates are responsible to control how the memory is affected by the current input, i_t, and previous memory, f_t , and control the output of the current memory cell, o_t. The memory is updated at each time step using the following rules:

$$\mathsf{i}_t = \sigma(\mathsf{W}_i\mathsf{h}_{t-1} + \mathsf{U}_i\mathsf{x}_t) \tag{2.60}$$
$$\mathsf{f}_t = \sigma(\mathsf{W}_f\mathsf{h}_{t-1} + \mathsf{U}_f\mathsf{x}_t) \tag{2.61}$$
$$\mathsf{o}_t = \sigma(\mathsf{W}_o\mathsf{h}_{t-1} + \mathsf{U}_o\mathsf{x}_t) \tag{2.62}$$
$$\mathsf{c}_t = \mathsf{f}_t \odot \mathsf{c}_{t-1} + \mathsf{i}_t \odot \tanh(\mathsf{W}_c\mathsf{h}_{t-1} + \mathsf{U}_c\mathsf{x}_t) \tag{2.63}$$
$$\mathsf{h}_t = \mathsf{o}_t \odot \tanh(\mathsf{c}_t) \tag{2.64}$$

where i_t is the input gate, f_t the forget date, o_t the output gate, c_t the memory cell and h_t the hidden state. $\mathsf{W}_i, \mathsf{W}_f, \mathsf{W}_o, \mathsf{U}_i, \mathsf{U}_f, \mathsf{U}_o$ represent weight matrices to be learned that are shared across all time steps.

2.4.2 Temporal Convolution Network

The Temporal Convolution Network (TCN) architecture differs from the traditional convolution by the following reasons: the convolutions in the architecture are causal, meaning that future information is not used in past time steps; the architecture can receive a sequence of any length and map it to a sequence of the same length [BKK18]. The causal nature of the architecture is achieved by causal convolutions, convolutions where an output at time t is convolved only with elements from time t and earlier in the previous layer [BKK18]. To produce an output of the same length of the input, it is used a 1D fully-convolutional network architecture, where each hidden layer is the same length as the input layer, and zero padding of length (kernel size - 1) is added to keep subsequent layers the same length as previous ones [BKK18].

Causal Convolution

A causal convolution can only take into account a past history with size linear in the depth of the network. With this in mind, the use of dilated convolutions comes handy, since it enables an exponentially large receptive field. That is, for a 1-D sequence input $\mathsf{x} \in \mathbb{R}^n$ and a filter $f\colon \{0, \ldots, k-1\} \mapsto \mathbb{R}$, the dilated convolution operation F on element s of the sequence is defined as

$$F(s) = (\mathsf{x} *_d f)(s) = \sum_{i=0}^{k-1} f(i) \cdot \mathsf{x}_{s-d\cdot i} \tag{2.65}$$

where d is the dilation factor, k is the filter size, and $s - d\cdot i$ accounts for the direction of the past [BKK18].

Residual Blocks

A residual block applies a series of transformations $\mathcal{F}$ to an input. Said input x is then added to the output of the residual block [BKK18]:

$$o = \text{Activation}(\text{x} + \mathcal{F}(\text{x})) \tag{2.66}$$

Such concept allows the layers to learn modifications to the identity mapping rather than the entire transformation [BKK18].

The TCN's receptive field depends not only on the network depth n, but also on the filter size k and dilation factor d. For that reason, the stabilization of deeper and larger TCNs is a topic of importance [BKK18]. Each layer consists of multiple filters for feature extraction. The generic TCN model employs generic residual modules in place of a convolution layer [BKK18]. A residual block consists of two layers of dilated causal convolution and non-linearity, for which is used the ReLU activation function. Additionally, to introduce normalization, the convolutional filters are normalized through weight normalization. Furthermore, dropout is also employed after each dilated convolutions for regularization, zeroing a whole channel at each training step [BKK18].

Note that in the TCN architecture, we may encounter different shapes between the input and output. To handle such cases, it is used and additional 1x1 convolution to ensure that element wise addition receives tensors of the same shape [BKK18].

2.4.3 Transformer

The Transformer [VSP+17] follows an encoder-decoder structure, where the encoder maps an input sequence of symbol representations $(x_1, \ldots, x_n)$ to a sequence of continuous representations $\text{z} = (z_1, \ldots, z_n)$. Based on the representation z, the decoder generates an output sequence $(y_1, \ldots, y_n)$ of symbols one element at a time. In each time step the model takes an auto-regressive behaviour, consuming the previously generated symbols as additional input [VSP+17]. Additionally, the Transformer takes advantage of self-attention and point-wise, fully connected layers. These layers are present in both the encoder and decoder.

The encoder can be composed of multiple stacks, where each layer has two sub-layers: the first being a multi-head self attention mechanism, and the second a position-wise fully connected feed-forward network [VSP+17].

The decoder is similar to the encoder, however, it can have an additional sub-layer which features a multi-head attention layer over the encoder's output [VSP+17].

An example of such architecture is pictured in figure 3.1.

3

Hyper Parameter Optimisation

3.1 Context

HPO is a challenging problem that aims to find the parameters which produce the best results for a given task and a given performance measure. Formally, following [JGST19], let Λ be a n-dimensional hyper-parameter space, such that $\Lambda = \Lambda_1 \times \Lambda_2 \times \cdots \times \Lambda_n$ where Λ_i is the hyper-parameter set of the i-th hyper-parameter. Denote the space of all possible models by $\mathcal{M}$, and the model $M \in \mathcal{M}$ equipped with hyper-parameters λ, by M_λ. Let $\mathcal{D}$ be the set of all datasets and $\mathcal{P}$ be a performance measure, that is, a function that measures how well the model performed given a task and a dataset of $\mathcal{D}$. We want to estimate a function $\mathcal{T}\colon \mathcal{D} \times \Lambda \mapsto \mathcal{M}$ such that:

$$\mathcal{T}(D, \lambda) = \underset{M_\lambda \in \mathcal{M}}{\arg\max}(\mathcal{P}(M_\lambda, D)) \tag{3.1}$$

In other words, given a set of hyper-parameter spaces, discrete or continuous, we want to retrieve the hyper-parameter configuration that maximises the performance $\mathcal{P}$ of our model M with the current data set D.

From this formalism, we can provide a configuration for a MDP as follows: the state, S contains statistical information about the dataset D, which will be elaborated next, and the sequence of tested hyper-parameters, that is,

$$S = (D_S, (\lambda^1, \lambda^2, \ldots, \lambda^n))$$

where D_S is the statistical information about D, λ^i is the i-th hyper-parameter configuration and n is the episode length; the action set, A, represents the configurations we are able to select, in this case its Λ itself; π is the transition probabilities we want to estimate, that is, a policy for state transition; and, finally, the reward, r, corresponds to the model's performance for a given hyper-parameter configuration.

To estimate π, we train a RL model that helps navigate the hyper-parameter space by sequentially choosing new hyper-parameters based on the previous information.

3.2 Dataset Statistics

Contrary to approaches like BOHB, Hyperband and [WCL20] that tune for a specific dataset, our goal is to learn a policy that generalizes well to the same class of problems. That is, for instance, for classification problems we aim to learn a policy that can aid the hyper-parameter space search on various classification datasets, avoiding unnecessary iterations and model training. As a way to accomplish this, we first need to define dataset measures. We follow the procedure of [JGST19]. For a given dataset $D \in \mathcal{D}$ we consider the following information:

- Mean kurtosis.
- Max kurtosis.
- Min kurtosis.
- Kurtosis standard deviation.
- Mean skewness.
- Max skewness.
- Min skewness.
- Skewness standard deviation.
- Number of instances.
- Logarithm of number of instances.
- Data dimension.
- Logarithm of data dimension.
- Inverse data dimension.
- Logarithm of inverse data dimension.

This information is then represented by D_S. Concretely, $D_S \in \mathbb{R}^{1\times 14}$ is a vector in which each entry corresponds to a statistic measure referenced above.

$$D_S = \begin{bmatrix} \mu_k & \max_k & \min_k & \sigma_k & \mu_s & \max_s & \min_s & \sigma_s & N & \log(N) & d & \log(d) & 1/d & \log(1/d) \end{bmatrix}$$

The vector is posteriorly included in the state S.

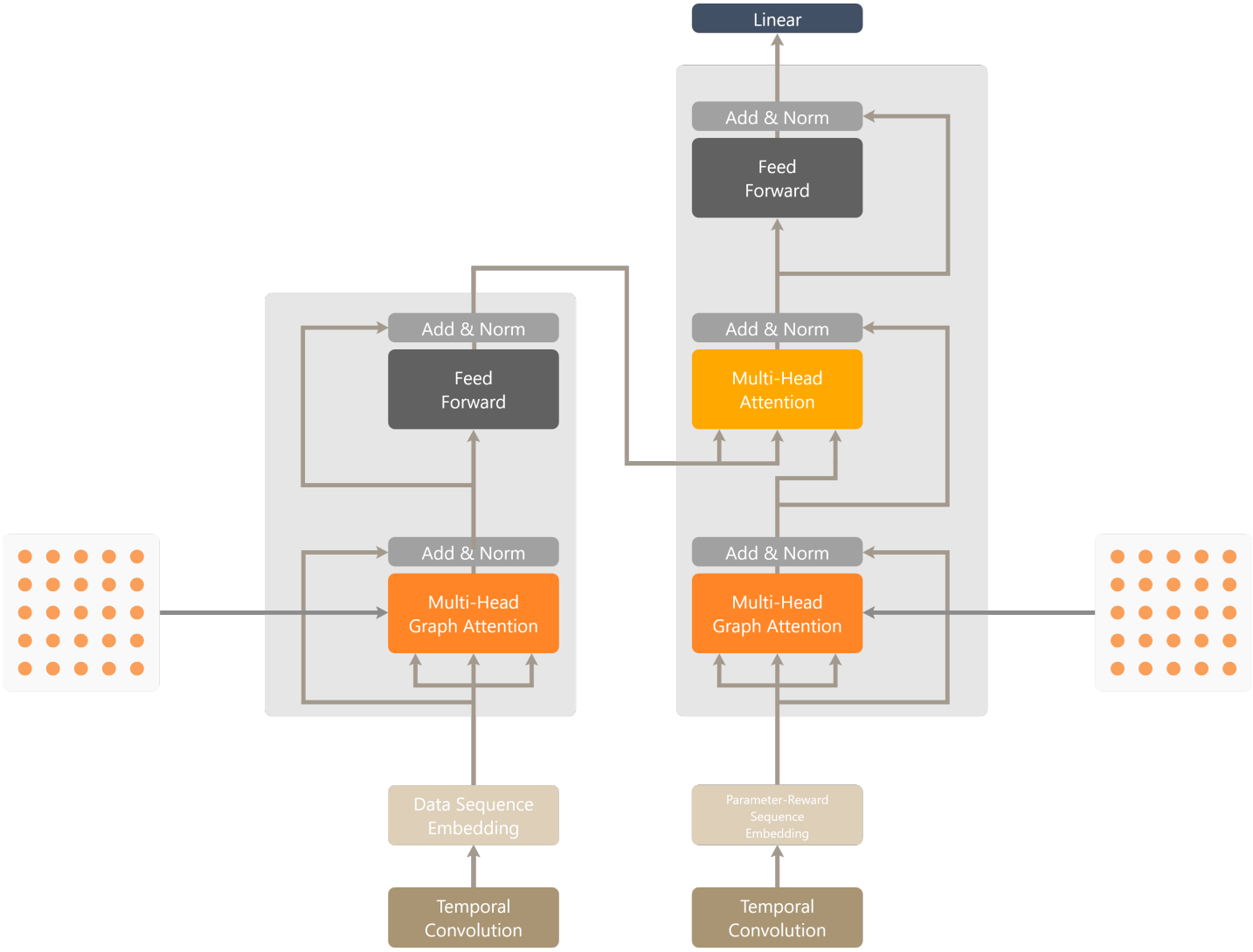


Figure 3.1: TGraph Base: Param-Reward Encoding

3.3 Proposed Approach

3.3.1 Graph Attention Transformer

Since our main objective is to train a policy that generalizes across datasets for a given task, algorithm and performance measure, we mainly use the capabilities of transfer learning. Namely, the popular approach of the transformer [VSP+17]. Here, the transformer is though of as a function that takes as arguments the current dataset statistics, to the encoder, and the sequence of observed hyper-parameters, to the decoder, in order to produce an embedding that helps the next hyper-parameter decision. The base architecture is displayed in figure 3.1.

There are some differences to the original transformer that we will now discuss. Firstly, we encode both the data statistics sequence and the parameter sequence with temporal convolution networks. The use of this networks comes from the fact that recurrent networks are known for gradient issues, gradient vanishing or exploding gradients. Also, we can use the capabilities of convolution to locally analyse the sequences without referring to future time steps. Moreover, since temporal convolution is in use, we can already

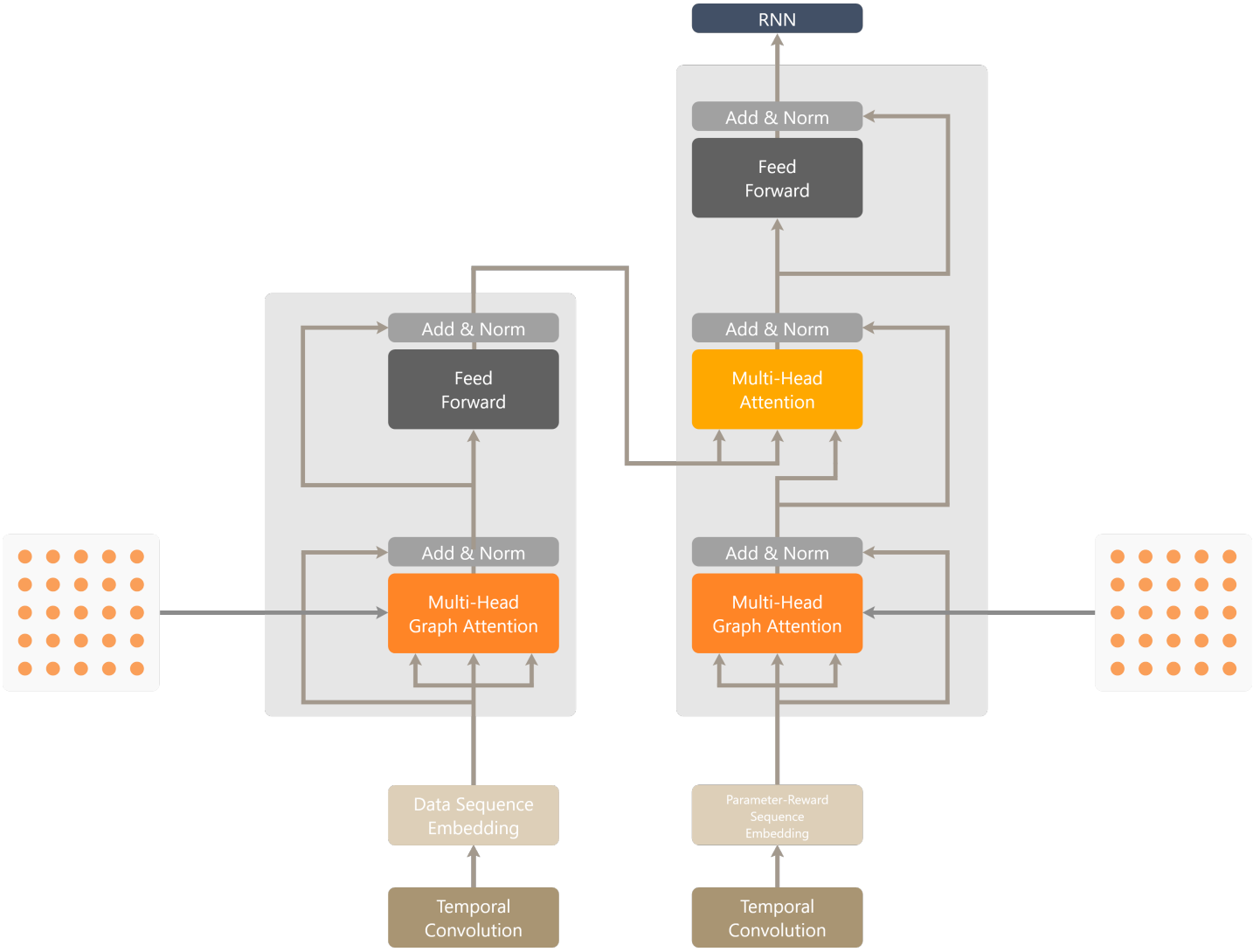


Figure 3.2: TGraph RNN: Param-Reward Encoding

extract causality at the encoding level. Secondly, we replaced the usual masked multi-head attention with multi-head graph attention, with the corresponding adjacency matrices represented in the picture. The main reason behind this is as follows: the decision of one parameter, influences the decision of the remaining parameters, that is, there may be parameters that are dependent of each other. Using this attention mechanism, we try to capture these dependencies and enrich each sequence encoding. The same is done with the data statistics sequence.

Notice that in figure 3.1, we include the reward sequence in the parameter embedding as a way to early enclose possible correlation between the reward and parameters sequence. We provide three more variants of the base architecture. The first variant, presented in figure 3.2, applies a RNN layer, in our case a LSTM network, at the output of the transformer, instead of the traditional dense layers. We introduce this change in order to impose sequential decision over the hyper-parameters, in other words, given the embeddings of the hyper-parameters, by passing trough a RNN, the previous hyper-parameters embeddings will influence the decision of the next hyper-parameter. The remaining variants separate the parameter and reward embeddings, concatenating them before the final layer, figures 3.3, 3.4.

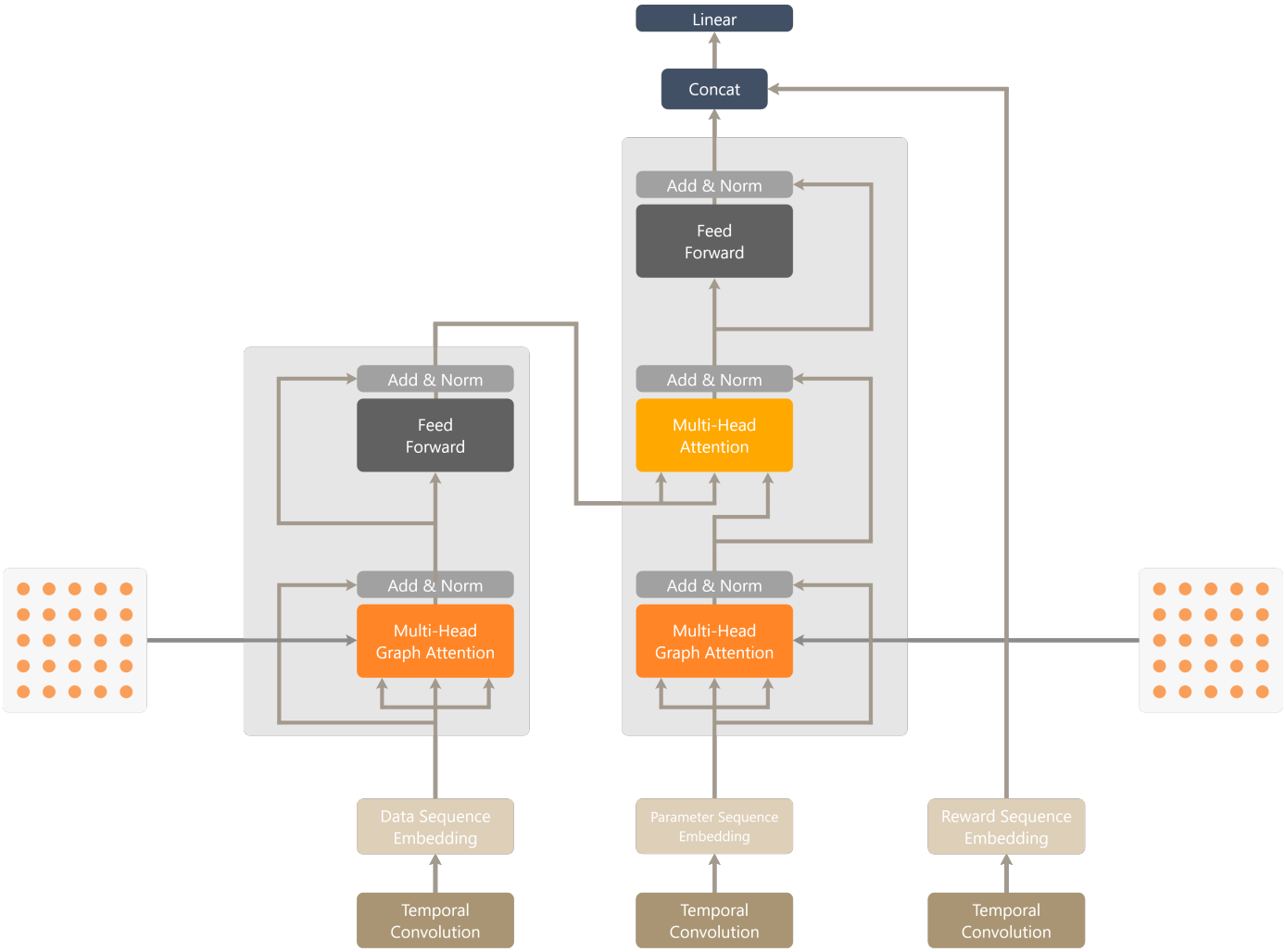


Figure 3.3: TGraph Base: Param-Reward Concat

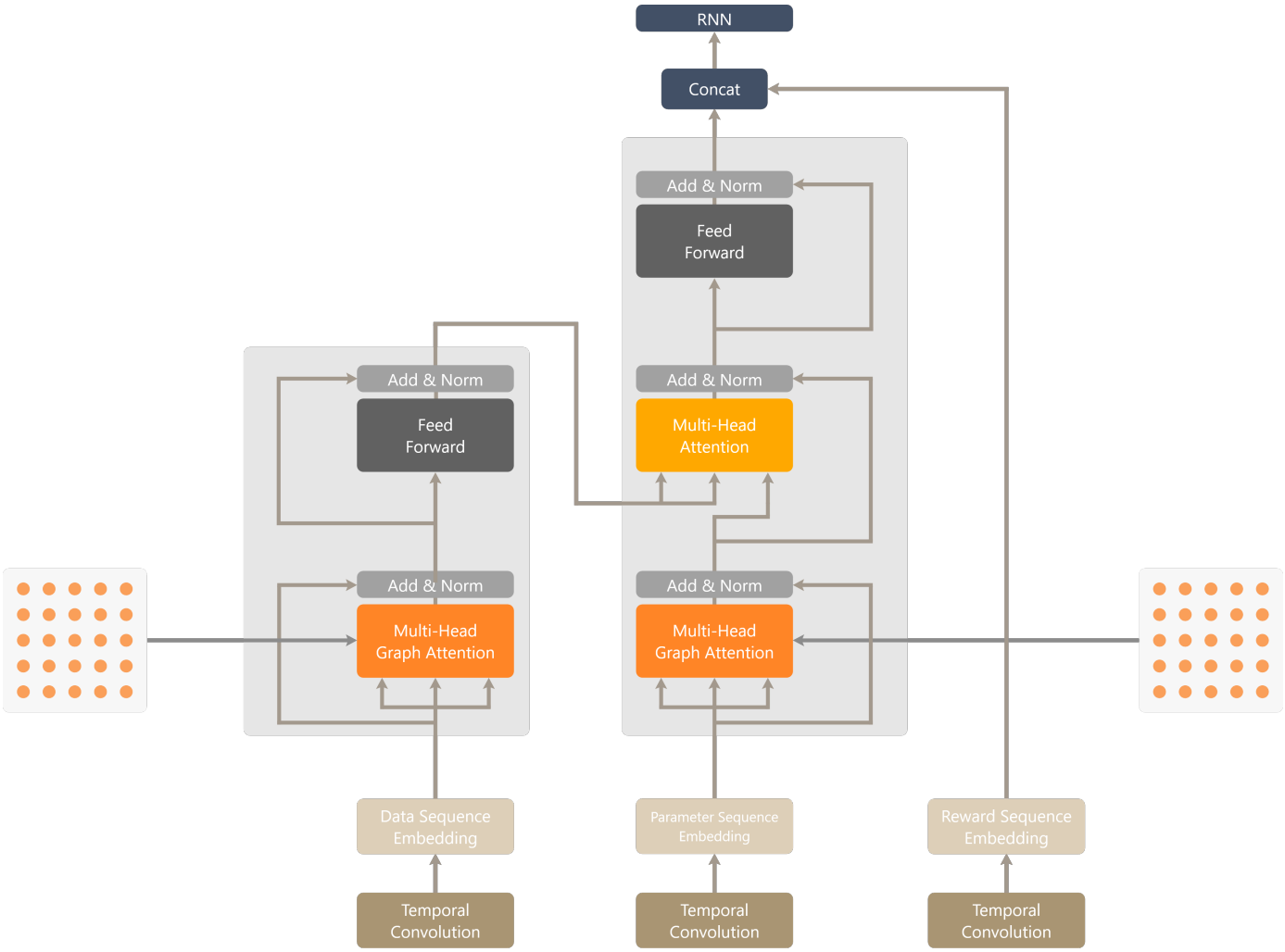


Figure 3.4: TGraph RNN: Param-Reward Concat

3.3.2 Action Distribution Networks

Action distribution networks receive the state embeddings and produce distributions that are further used to produce an action vector. In this case, the state embedding is the transformer's output, and for each parameter, the action network will produce a normal distribution, in case of a continuous parameter, and a categorical distribution, when concerning discrete parameters. The action distribution network can be visualized as the transformer plus projection layers in figure 3.5.

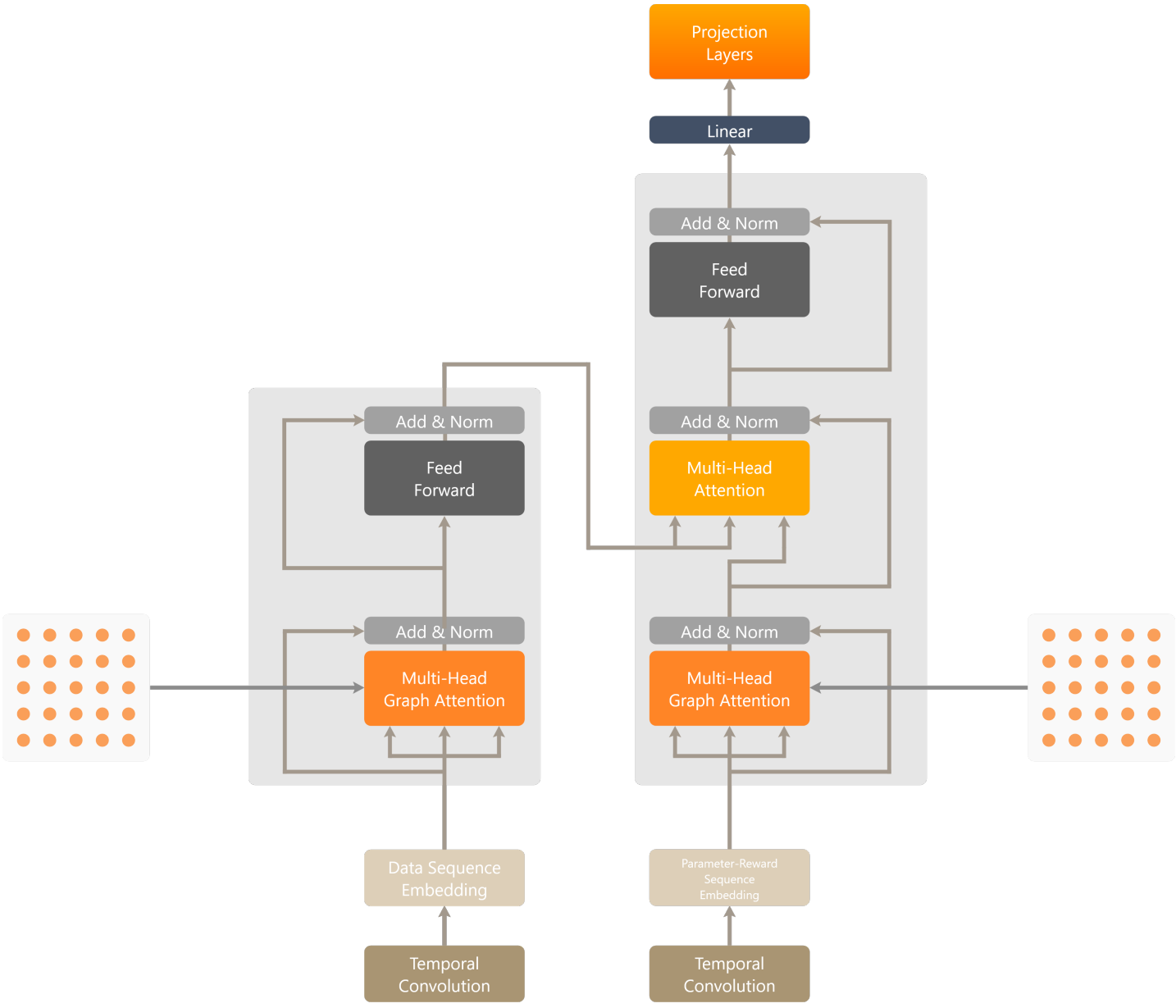


Figure 3.5: Action TGraph Example

3.3.3 Value Networks

Value networks receive the state embedding and produce a scalar describing how good the state is. Once again, the state embedding is produced by the transformer. The state embedding is posteriorly processed

through a dense layer that outputs said scalar. Visually, we just have to replace the projection layers with a dense layer in figure 3.6.

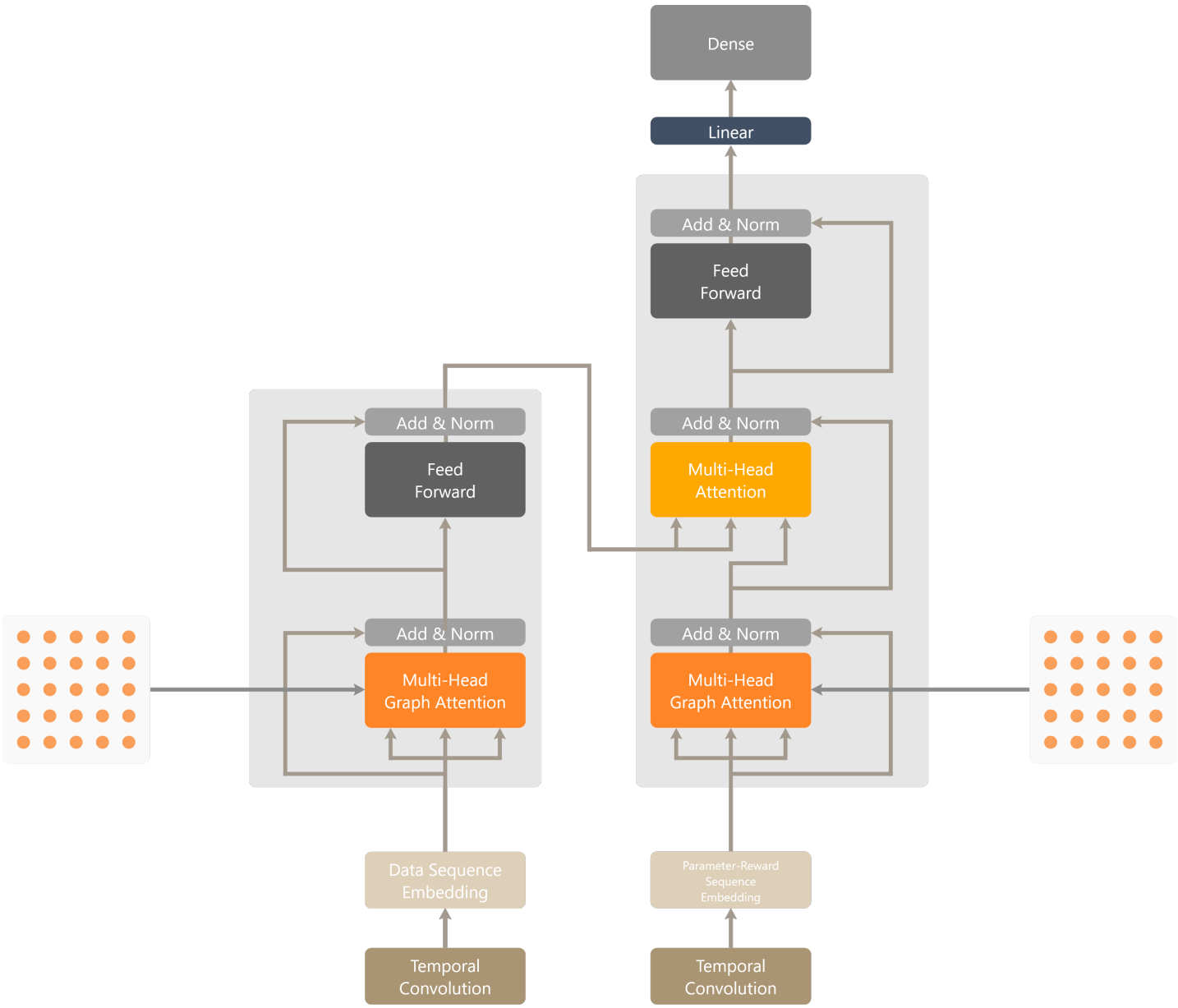


Figure 3.6: Value TGraph Example

3.4 Training

To experiment with the proposed approach, we begin by selecting a meta-dataset for classification tasks, that is, a dataset that contains classification datasets ready to use for our purposes. We found the datasets in `https://github.com/uestc-chensp/datasets` of particular interest, since they are ready to use and provide a good base for us to test on. However, we did not use all of the available sets. For speed purposes we selected the smaller datasets, concretely, datasets with less than 1500 samples, leaving 67 datasets for us to use. The larger datasets will serve their purpose later. Each dataset is further split into train and test sets, with 80% of the instances for training and 20% for testing.

Given that the chosen datasets are directed towards classification tasks, we selected XGBoost [CG16] to be our model to tune. This decision was mainly motivated by XGBoost's popularity, simplicity and powerfulness. The hyper-parameters considered for tuning are those described in table 3.1, together with the respective minimum and maximum allowed values.

Hyper-Parameter	Min Value	Max Value	Type
max_depth	1	25	Discrete
learning_rate	0.001	0.1	Continuous
n_estimators	50	1200	Discrete
gamma	0.05	0.9	Continuous
min_child_weight	1	9	Discrete
subsample	0.5	1	Continuous
colsample_bytree	0.5	1	Continuous
colsample_bylevel	0.5	1	Continuous
reg_alpha	0.1	0.9	Continuous
reg_lambda	0.01	0.1	Continuous

Table 3.1: XGBoost Hyper-parameters

In total, we have a set of 10 hyper-parameters, with different amplitudes and different natures, discrete and continuous.

For training our policy, we will be using the PPO algorithm, which is an episodic algorithm. In other words, PPO makes use of full episodes to train the underlying policy. Moreover, PPO being an off-policy algorithm, uses a different policy to generate the episode. The collect-policy in this approach is a greedy ϵ-soft policy with $\epsilon = 0.1$ as a way to promote different parameter exploration. Specifically, we run PPO for 1000 iterations, collecting 5 episodes per iteration and optimizing the policy for 5 epochs. For performance reasons we maintain the episodes short, in our case, the episodes have length 20. These parameters are summarized in the following table.

Parameter Name	Value
Number of Iterations	1000
Collected Episodes per Iteration	5
Policy Update Epochs	5
Episode Length	20
Replay Buffer Size	1000

Table 3.2: Policy Training Parameters

The action and value transformers also come with their own set of parameters. In our experiment they share the same parameters and values, as they are described in table 3.3.

Parameter Name	Value
Number of Layers	2
Model Dimensions	128
Number of Heads	4
Feed-Forward Size	128
Output Size	128

Table 3.3: Transformer Training Parameters

We consider two different modes of training:

- Static: the dataset to optimise is chosen at random at the beginning of the episode;

- Dynamic: the dataset to optimise is chosen at random at the start of each episode iteration.

The static training procedure works as follows. At the start of the episode, we pick at random a dataset, D, from the meta-dataset. During the episode the XGBoost model is trained with the predicted parameters based on the dataset statistics. The model's training is done via 5-fold cross validation over D's training set. The reward is considered to be the mean of the cross validation accuracy (note that during training, D's test set is never seen neither used). The state is then updated with the observed reward and test parameters, as it will be used to obtain the next set of parameters. The episode ends when 20 parameters are tested. Here the data statistics remains the same through the episode, thus we do not need to further update the data state. The static training process is summarized in algorithm 5.

Algorithm 5: Static Training Algorithm

Input : A meta-dataset $\mathcal{D}$, Model to tune M
Input : A greedy policy π
Input : A ϵ-soft collect policy b
Algorithm parameters: Number of iterations N, Collect episodes per iteration C, Episode length T, Policy update epochs P_u, Replay buffer size R_s

for N *iterations* **do**
  $\mathcal{R} = \varnothing$
  for C *iterations* **do**
    $S_D = \text{statistics}(D)$ $D \sim \text{Unif}(\mathcal{D})$
    $S_P = \{(\lambda_0, 0)\}$ $\lambda_0 \sim \text{Unif}(\Lambda)$
    $S_0 = (S_D, S_P)$
    for $t \in 1, \ldots, T$ **do**
      Get next action from collect policy $\lambda_t = b(\lambda_{t-1}|S_{t-1})$
      Train model with parameters λ_t with 5-fold cross-validation on D's training set.
      $M_\lambda = M(\lambda_t, D_{\text{train}})$
      Get reward as the mean accuracy of cross-validation $r_t = R(M_\lambda)$
      Update parameter state $S_P \leftarrow S_P \cup \{(\lambda_t, r)\}$
      Update state $S_t = (S_D, S_P)$
      Add trajectory to replay buffer $\mathcal{R} \leftarrow \mathcal{R} \cup \{(S_t, S_{t+1}, \lambda_t, r_t)\}$ and replace the oldest tuple if $|\mathcal{R}| > R_s$
  Apply PPO to π for P_u iterations using $\mathcal{R}$ data.
return π

The dynamic training is similar to the static one, with the following differences: the training set D is updated at random at the start of each episode iteration; the data state includes the sequence of observed data statistics, and thus, the overall state contains both the dynamic data state and parameter-reward

state. This training procedure is summarized in algorithm 6.

Algorithm 6: Dynamic Training Algorithm

Input : A meta-dataset $\mathcal{D}$, Model to tune M
Input : A greedy policy π
Input : A ϵ-soft collect policy b
Algorithm parameters: Number of iterations N, Collect episodes per iteration C, Episode length T, Policy update epochs P_u, Replay buffer size R_s

for N *iterations* **do**
  $\mathcal{R} = \varnothing$
  for C *iterations* **do**
    $S_D = \{\text{statistics}(D_0)\}\ D_0 \sim \text{Unif}(\mathcal{D})$
    $S_P = \{(\lambda_0, 0)\}\ \lambda_0 \sim \text{Unif}(\Lambda)$
    $S_0 = (S_D, S_P)$
    for $t \in 1, \ldots, T$ **do**
      Get next action from collect policy $\lambda_t = b(\lambda_{t-1}|S_{t-1})$
      Train model with parameters λ_t with 5-fold cross-validation on D_{t-1}'s training set. $M_\lambda = M(\lambda_t, D_{t_{\text{train}}})$
      Get reward as the mean accuracy of cross-validation $r_t = R(M_\lambda)$
      Update parameter state $S_P \leftarrow S_P \cup \{(\lambda_t, r)\}$
      Draw new dataset from set $D_t \sim \text{Unif}(\mathcal{D})$
      Update data state $S_D \leftarrow S_D \cup \{\text{statistics}(D_t)\}$
      Update state $S_t = (S_D, S_P)$
      Add trajectory to replay buffer $\mathcal{R} \leftarrow \mathcal{R} \cup \{(S_t, S_{t+1}, \lambda_t, r_t)\}$ and replace the oldest tuple if $|\mathcal{R}| > R_s$
  Apply PPO to π for P_u iterations using $\mathcal{R}$ data.
return π

Both S_D and S_P are modeled with matrices of dimensions $T \times 14$ and $T \times (\dim \Lambda + 1)$ respectively.

3.5 Evaluation

The evaluation procedure works as follows: since our states are modeled with matrices of said dimensions, to extract the best of our approach we first fill the state matrices with T runs; secondly, having the matrices full of run information, we now take 1, 2, 3 and 4 trials. Note that the policy is still optimizing over cross-validation accuracy, however, since we are testing, we extract the test accuracy on each dataset.

We compare our approach with the standard Random algorithm, Hyperband, BOHB and TPE, using the average rank of the test set accuracy over the datasets as a comparison measure:

$$\text{Average Rank} = \frac{1}{|\mathcal{D}|} \sum_{D \in \mathcal{D}} \text{rank}_D \tag{3.2}$$

We also compare each algorithm to the baseline at each iteration. The baseline consists of the XGBoost

with the default parameters.

3.6 Discussion

3.6.1 Training Results

We begin by discussing the training results, namely the observed average return at evaluation episodes and training loss at model checkpoints.

Based on the plot in figure 3.7, we can observe that the RNN based models, TGR-PRC and TGR-PRE, have an upper advantage in the earlier stages of training, surpassing the base models. However, this tendency is not carried out to the late stages of the training phase, where the base models have a clear increasing in average return, when compared with the RNN models. In fact, the RNN models seem to be decreasing performance wise as the training progresses.

Figure 3.7: Average Return At Eval Episodes

As we can observe from the loss plot, figure 3.8, the model TGR-PRC loss is very high and maintains this tendency during training. The same happens with the model TGB-PRC, starting with higher loss values at the beginning of training, but steadily decreasing as training progresses. These major differences in loss values, when compared with the remaining models comes from the way the reward information is treated. From this plot, the concatenation of the reward information before the final layer of processing introduces noise, impacting the model's convergence. In fact, the model TGR-PRC is never able to recover from it. Furthermore, since the model TGR-PRC uses a RNN as the final layer, this could have also worsen the loss problem, which would justify why the model TGB-PRC could converge despite its initial loss values.

Moreover, looking at figure 3.8 we may find an explanation to the decay in performance in the late stages of training regarding the RNN models. In what concerns the TGR-PRC model, the loss plot suggests that the non convergence of the model is responsible for the decay in performance. Looking now at the remaining models, we note that the model TGR-PRE loss steadily decreases with no major loss spikes when compared

to the models TGB-PRE and TGB-PRC. A good explanation for this is that the base models drive the policy to explore more, leading to loss spikes, whereas the RNN model leads to safer parameter choices, which promotes smooth convergence. This extra exploration made by the base models, justify the increase in average return in the final stages of training. As they explored more, they were more likely to find better hyper-parameters, increasing the average return.

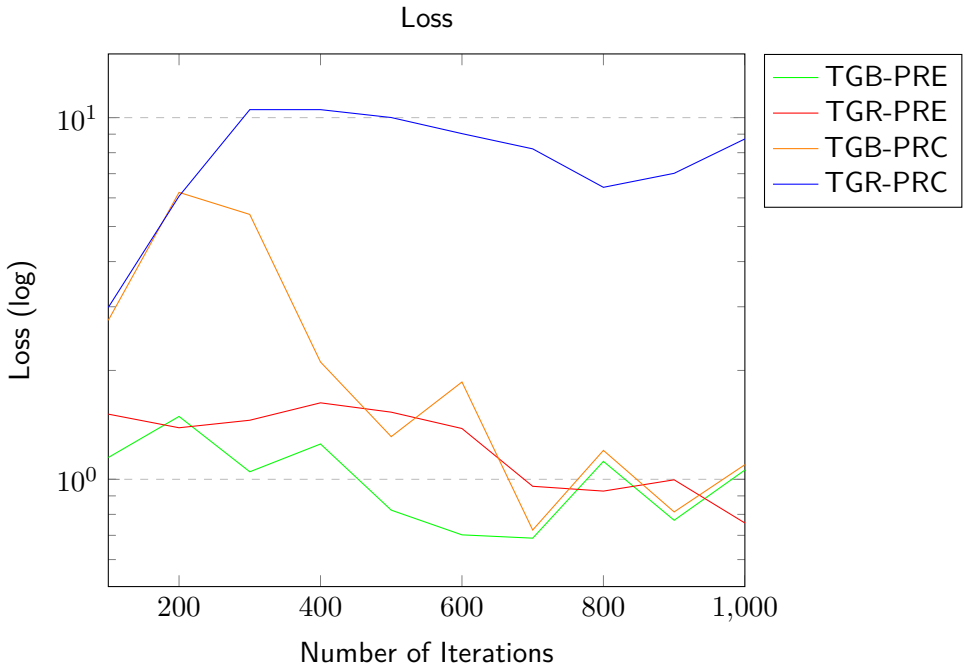


Figure 3.8: Loss at checkpoint in logarithmic scale

3.6.2 Test Set Comparison

Having discussed the training phase, we can now turn our attention over to the concrete results. We will now discuss each trained model performance in turn, based on figure 3.9.

As we already discussed, the model TGR-PRC had very high loss values, thus its performance was not expected to be among the best.

Regarding the TGB-PRC model, at 1 trials it has one of the best average ranks, but quickly diluting at the remaining trials. This was also one of the models with higher loss values. From this we can already draw some conclusions. First, the concatenation of the reward information maybe be beneficial in early runs, nonetheless, in the long term, it proved not to improve the model's hyper-parameter selection.

Looking now at TGB-PRE, it maintains its rank across trials, fairly keeping up the state of the art methods in the earlier and mid trials. However, at the final trial, it distances itself from the rest.

Finally, the TGR-PRE model achieved the best results of all the trained models. It starts with the lowest average rank and keeps this tendency across the four trials. As we would expect, the remaining methods catch up in the later iterations, but is worth noting that this model can still keep up, staying in range of state of the art methods.

We shall now focus our attention to the comparison with the baseline. Below, in figures 3.10, 3.11, 3.12 and 3.13, we provide a visual comparison to baseline at 1, 2, 3 and 4 trials, respectively.

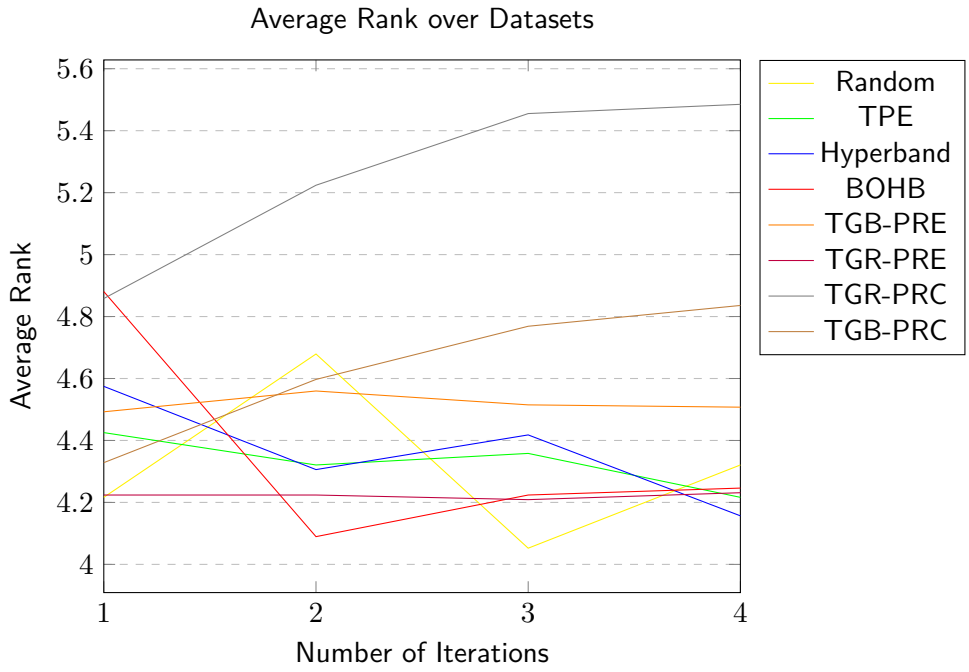


Figure 3.9: Average Rank on Test set accuracy

Looking at the first trial, both TGR-PRE and TGB-PRE have achieved better accuracy than the baseline in almost 40 datasets and have better results when compared to the remaining methods. This dominance reverts at the second iteration, since the remaining models are starting to find better parameters. Nonetheless, we observe a little improvement in TGR-PRE, which dominates again at 3 trials. At 4 trials the majority of the methods achieved better accuracy than the baseline in almost 40 datasets. We shall note that the models TGR-PRC and TGB-PRC remained constant over the four trials, which implies that the found hyper-parameters were not sufficient to improve past the baseline. Moreover, TGR-PRC was not able to get at least 50% better results when compared to the baseline. Table 3.4 summarizes the comparison to baseline at different iterations.

Method	1 Trial	2 Trial	3 Trial	4 Trial
Random	33	35	37	41
TPE	35	35	39	42
Hyperband	33	40	37	41
BOHB	34	43	38	39
TGB-PRE	36	35	36	36
TGR-PRE	38	39	40	39
TGR-PRC	31	31	31	31
TGB-PRC	35	35	35	35

Table 3.4: Number of Datasets Better than Baseline

3.6.3 Attention Analysis

The transformer architecture provides us with a form of understanding the decision making and which variables the model is looking at by the analysis of the attention plots. For this analysis we will be using the architecture TGR-PRE. We start by the encoder graph attention, figure 3.14. This attention mechanism

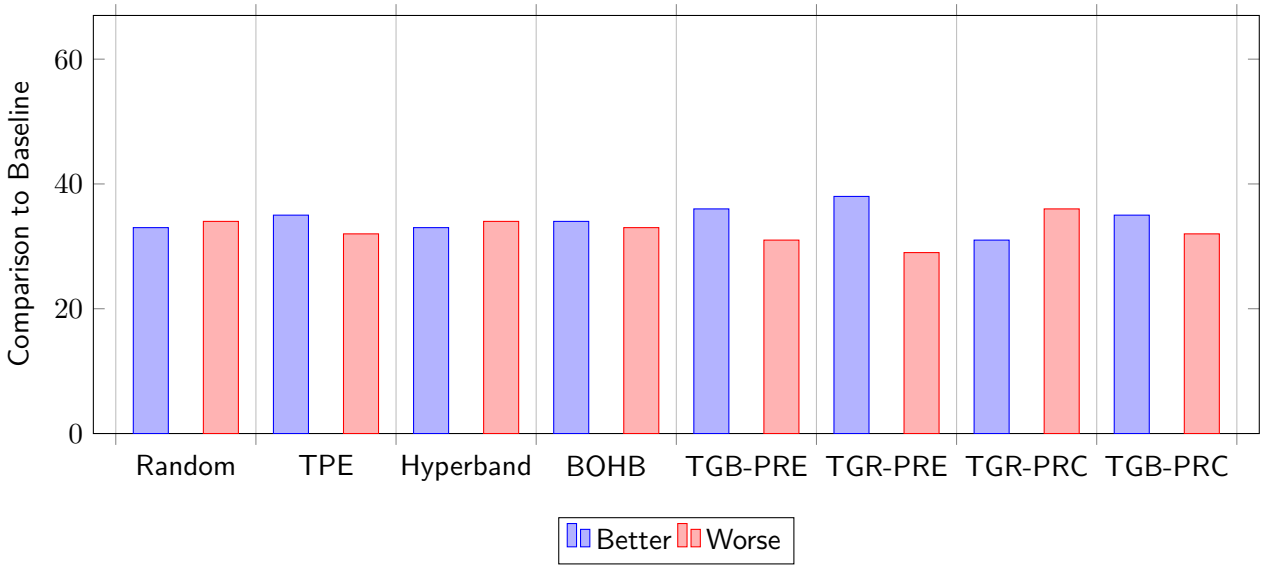


Figure 3.10: Comparison to Baseline at 1 trial

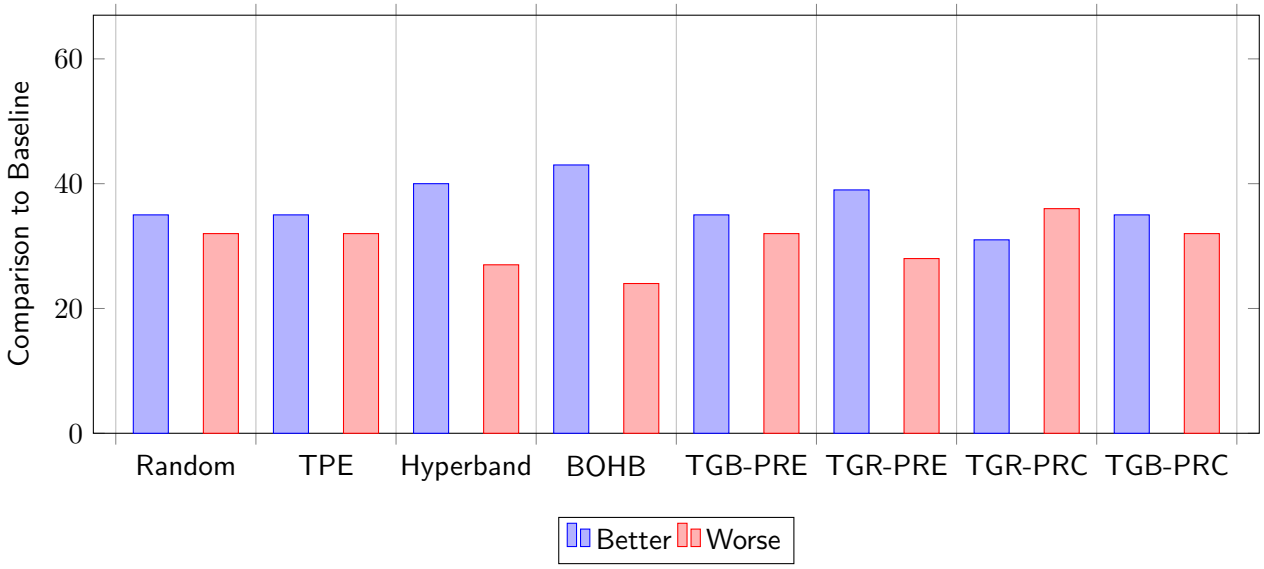


Figure 3.11: Comparison to Baseline at 2 trials

encaptures the compatibility of the various data statistics extracted from the dataset. From the plot we can conclude that the feature which has more importance to the remaining statistics is the number of instances. We also note that the logarithm of the dimensions and the kurtosis mean pay no such role, since there is little attention over this values. As a consequence of the static training used in this phase, this attention plot remains static across iterations.

Looking now at the decoder graph attention, figure 3.15, at this first iteration, the subsample, reg lambda and reward parameters are targeted with the highest attention scores, followed by the max depth and learning rate. It also follows that the number of estimators is not taken in so much consideration in this case. Contrary to the encoder graph attention mechanism, this mechanism does not remain static over iterations, as we can see in figure 3.16, since the parameters matrix contains a new sequence at each iteration. At the next iteration the model got information about the previously predicted sequence, which is now inserted into the parameter matrix, provoking a shit in attention to different parameters, in this

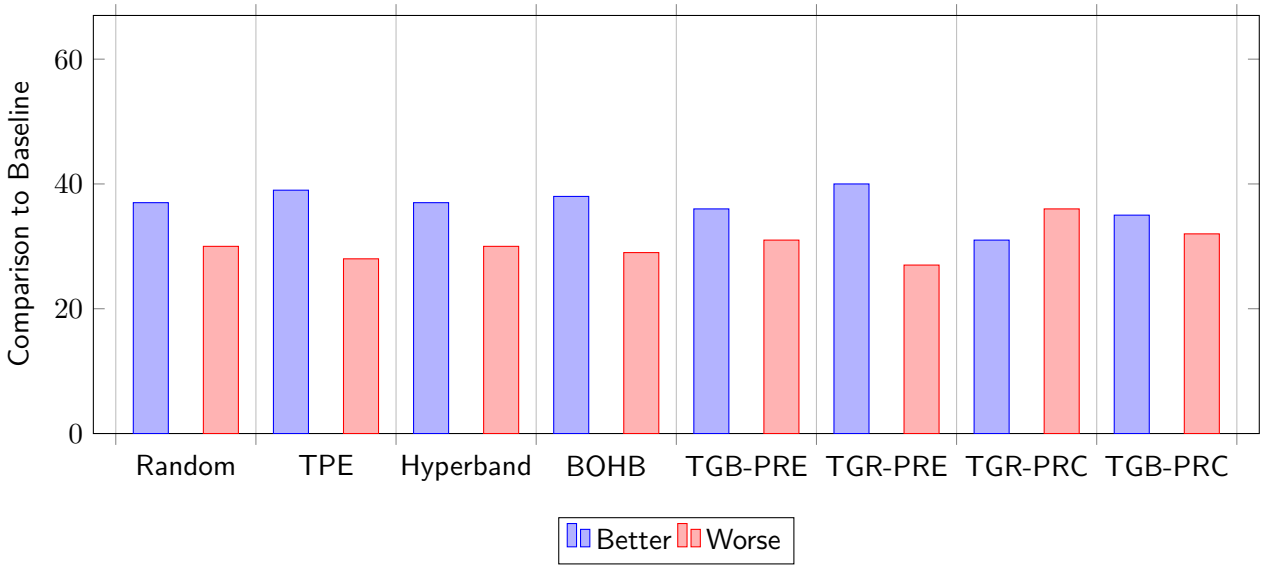


Figure 3.12: Comparison to Baseline at 3 trials

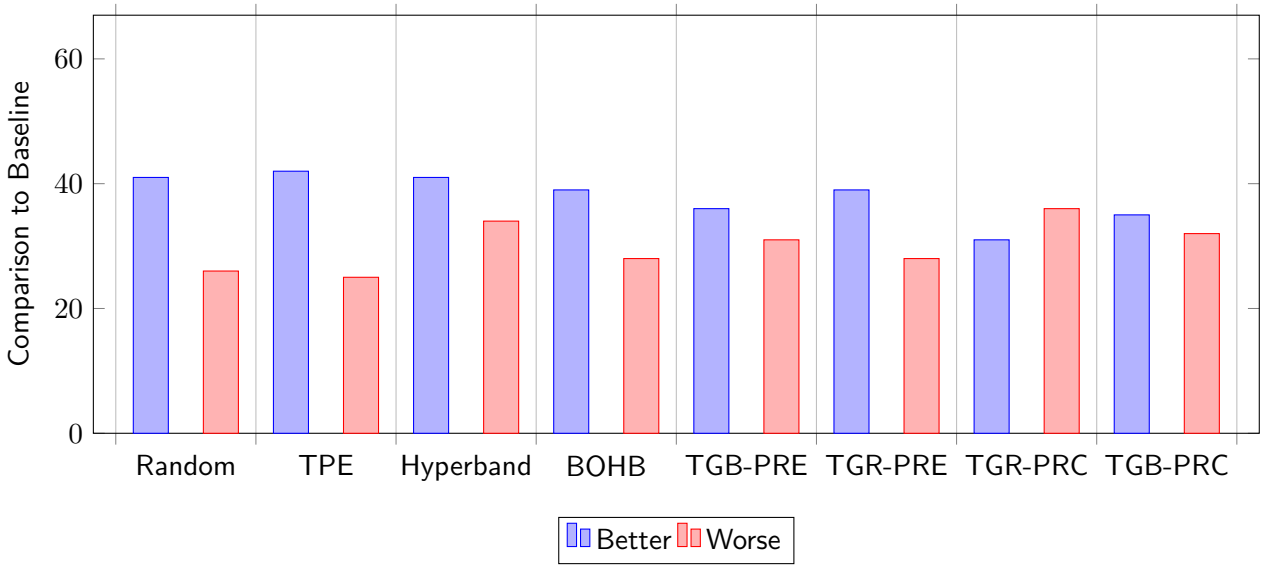


Figure 3.13: Comparison to Baseline at 4 trials

case, min child weight, colsample by level and reward. This supports the argument that the model is testing out new parameters, focusing on different parameters at different iterations, smoothing the process of navigating the hyper-parameter space.

To finish this section, it remains to analyse the mulihead attention mechanism responsible for the compatibility between the data statistics and the parameter sequences, figures 3.17, 3.18, 3.19 and 3.20. A quick analysis of these plots already shows that different heads are focusing on different statistics, while the first head is focusing on the number of instances, the second head is more focused on the data dimension. Looking closely, we can also conclude that the different heads keep paying little attention to the dimension of the logarithm and the kurtosis mean, which is to be expected based on the encoder graph attention plot. Similarly to the decoder graph attention, this attention mechanism also changes over iterations, as we can see in figure 3.21 which represents the attention plot of second head at the second iteration. We can easily see that for the colsample by tree parameter, the attention head increased the importance of the

data dimension statistic.

3.6.4 Generalization Test

We shall now provide a use for the datasets that were set aside. In figure 3.22 we are able to prove that our model is indeed capable of generalizing to unseen datasets for the same task, in this case, classification. For these results we used only our best performant method, TGR-PRE.

Despite not being the best model, TGR-PRE is capable, on unseen datasets, to keep on pair with the remaining methods. Thus, the capabilities of transfer learning of the transformer come into play, providing usability for a good hyper-parameter estimate, without the need of further training.

These results could definitely be improved with more training iterations, episodes collected and policy updates. In addition, even if just the dataset statistics led us to this point, a better way to represent the dataset should also be of interest, namely the idea of a dataset embedding that captures the features of a dataset, as the one introduced in [JSTG21].

3.7 Remarks

In this section we took advantage of the transformer as a function that learns a mapping between the encoder input and decoder output. This manner of thought enables us to produce an architecture capable of fine-tuning a model across general datasets for a specific task, in this case, classification. With that in mind, graph attention mechanisms were essential in enriching and capturing early dependencies between the studied variables, either in the data sequences or the parameter sequences. The parameter information is then combined with the data information with the purpose of establishing a connection with the dataset features. This weighted connection can, as we saw, change over time as the model receives new information from the previous predicted parameters. Furthermore, the attention plots provides us with a visual way of understanding which parameters the model is now focusing on to navigate the hyper-parameter space.

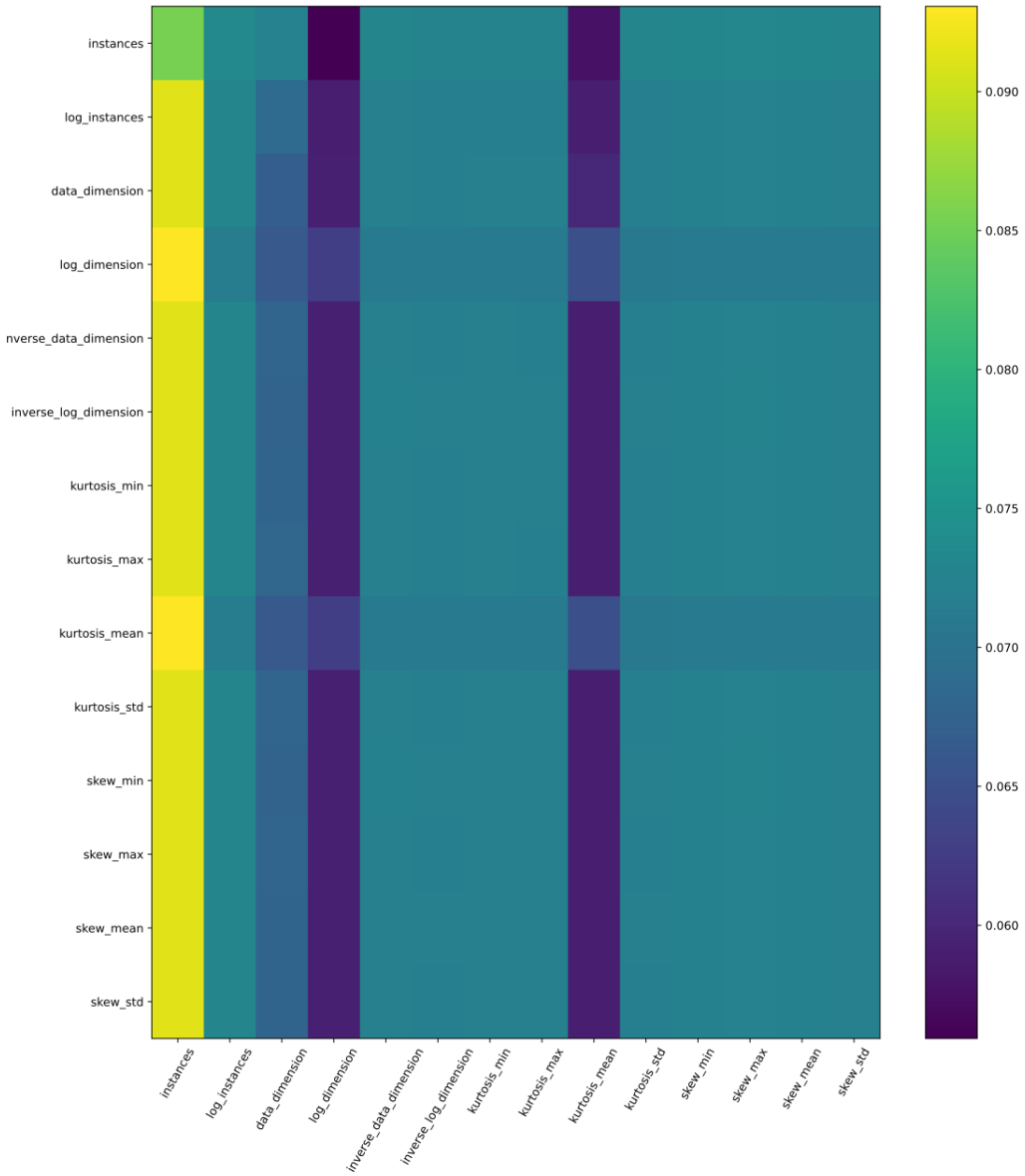


Figure 3.14: First Iteration Graph Attention Encoder Plot

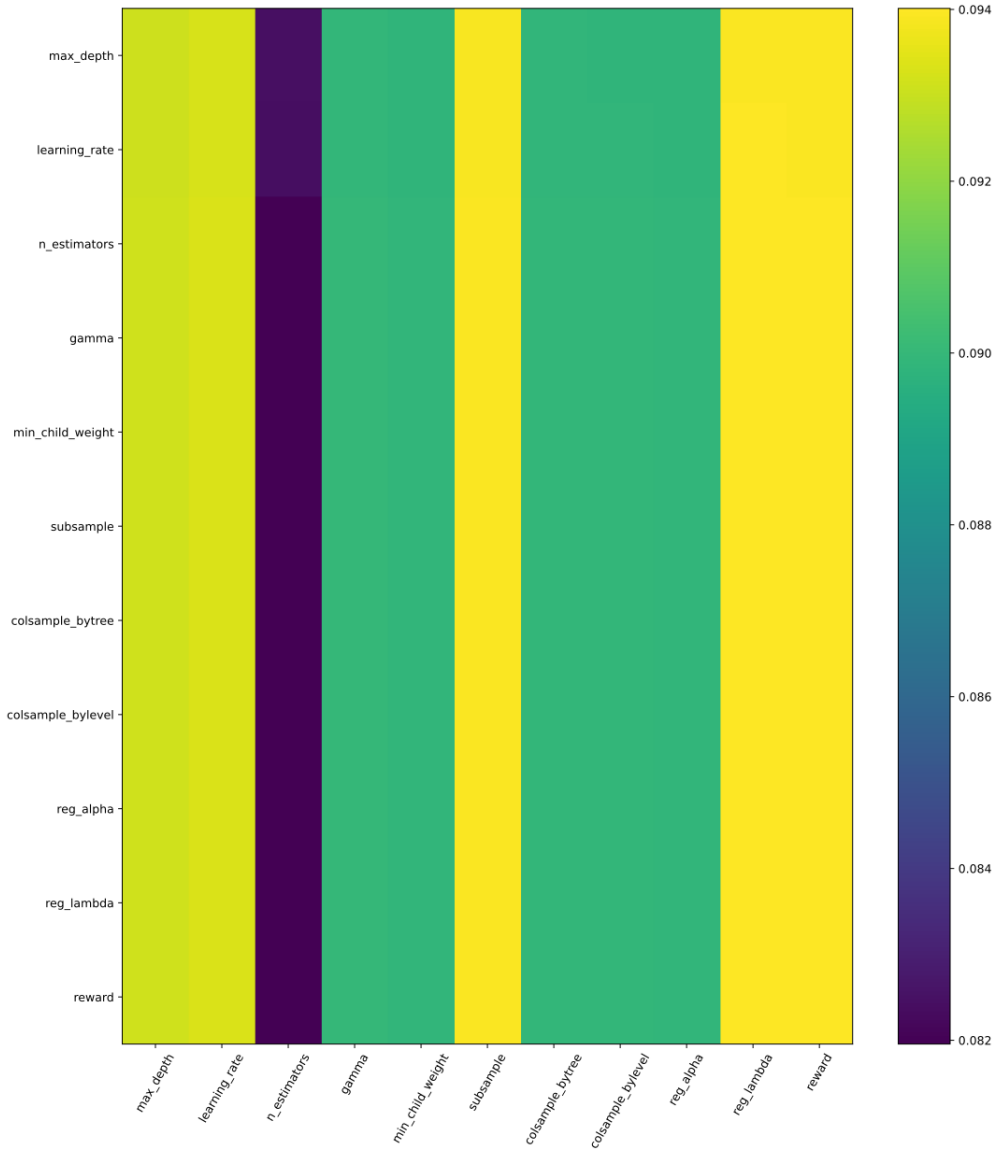


Figure 3.15: First Iteration Graph Attention Decoder Plot

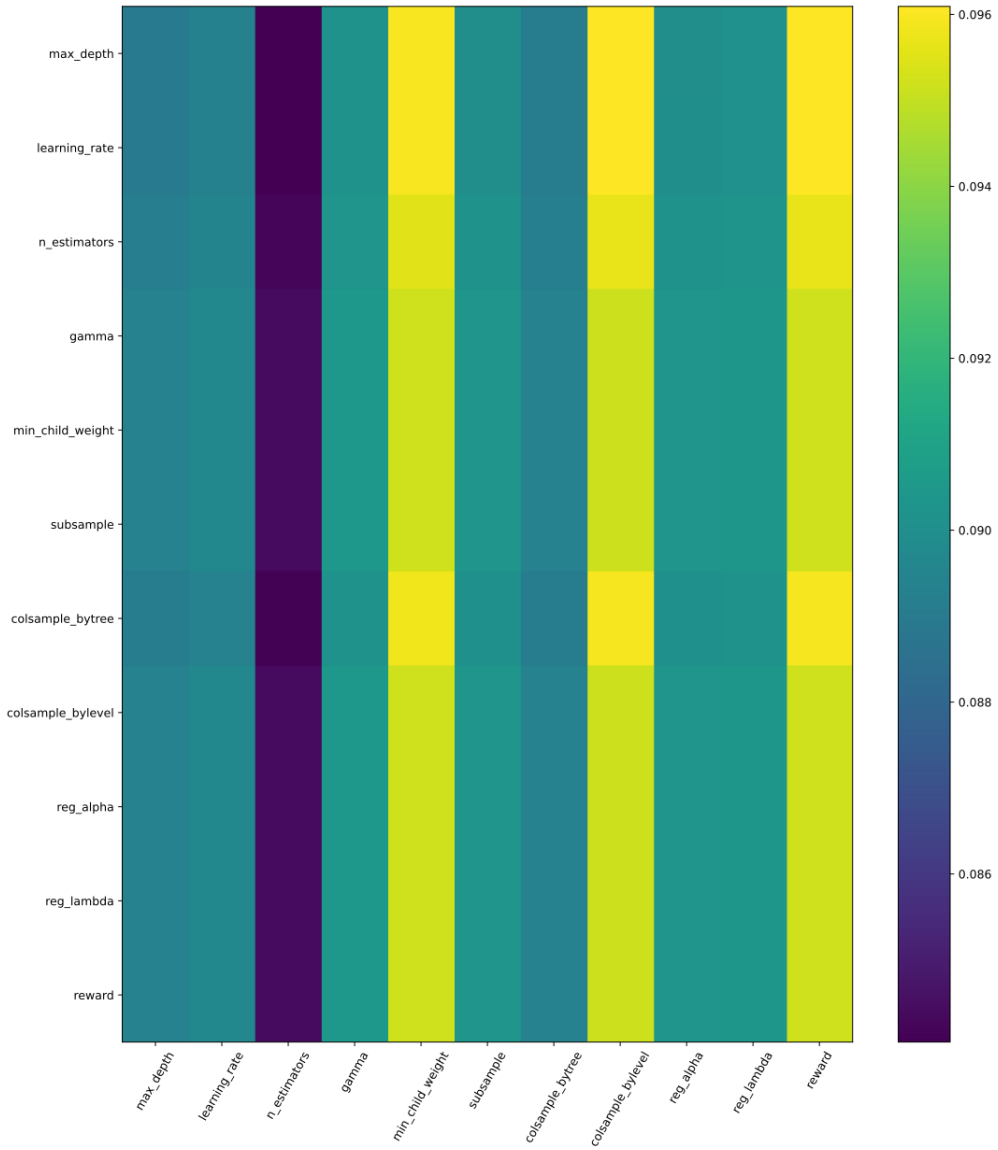


Figure 3.16: Second Iteration Graph Attention Decoder Plot

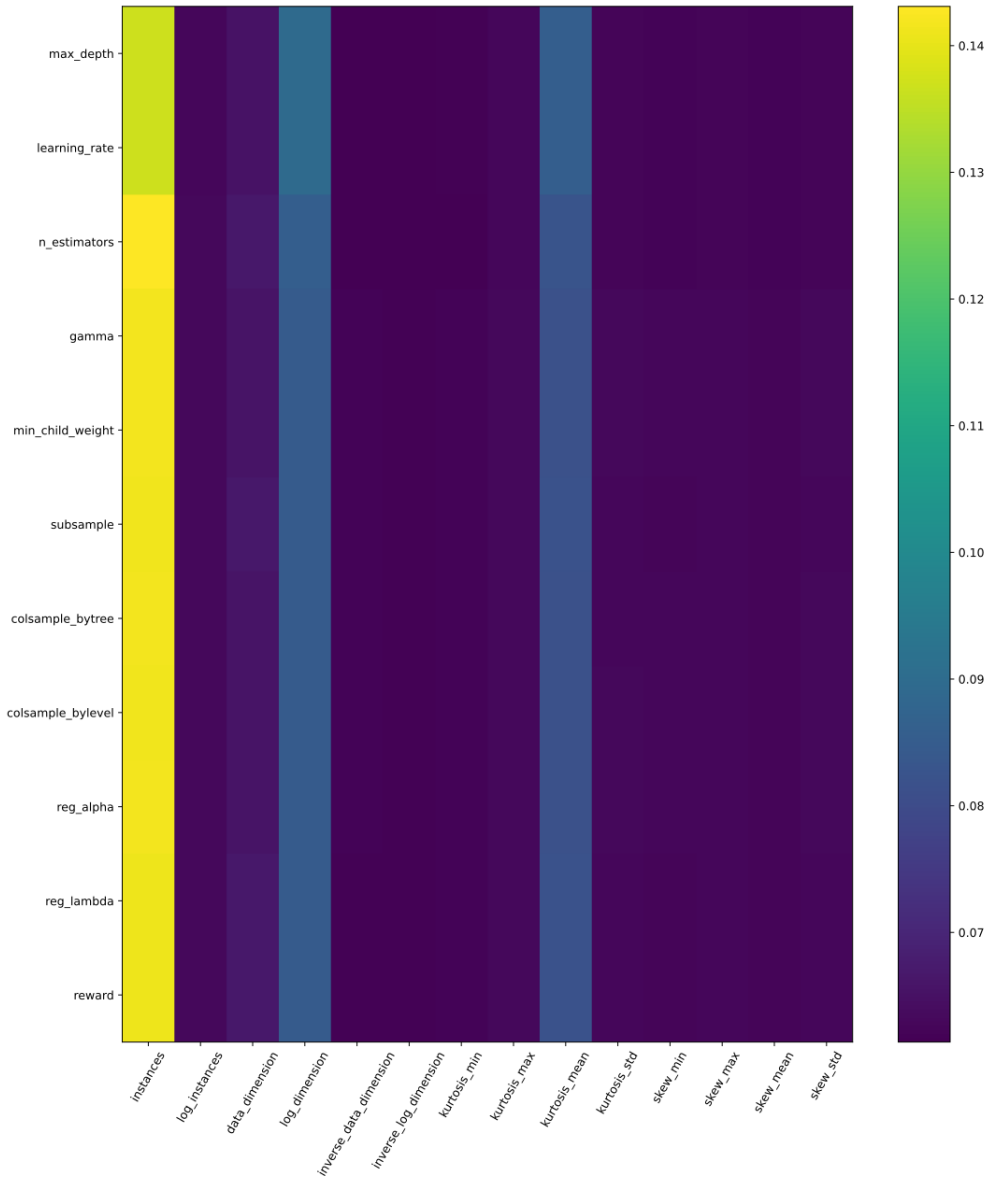


Figure 3.17: First Iteration Head-1 Attention Plot

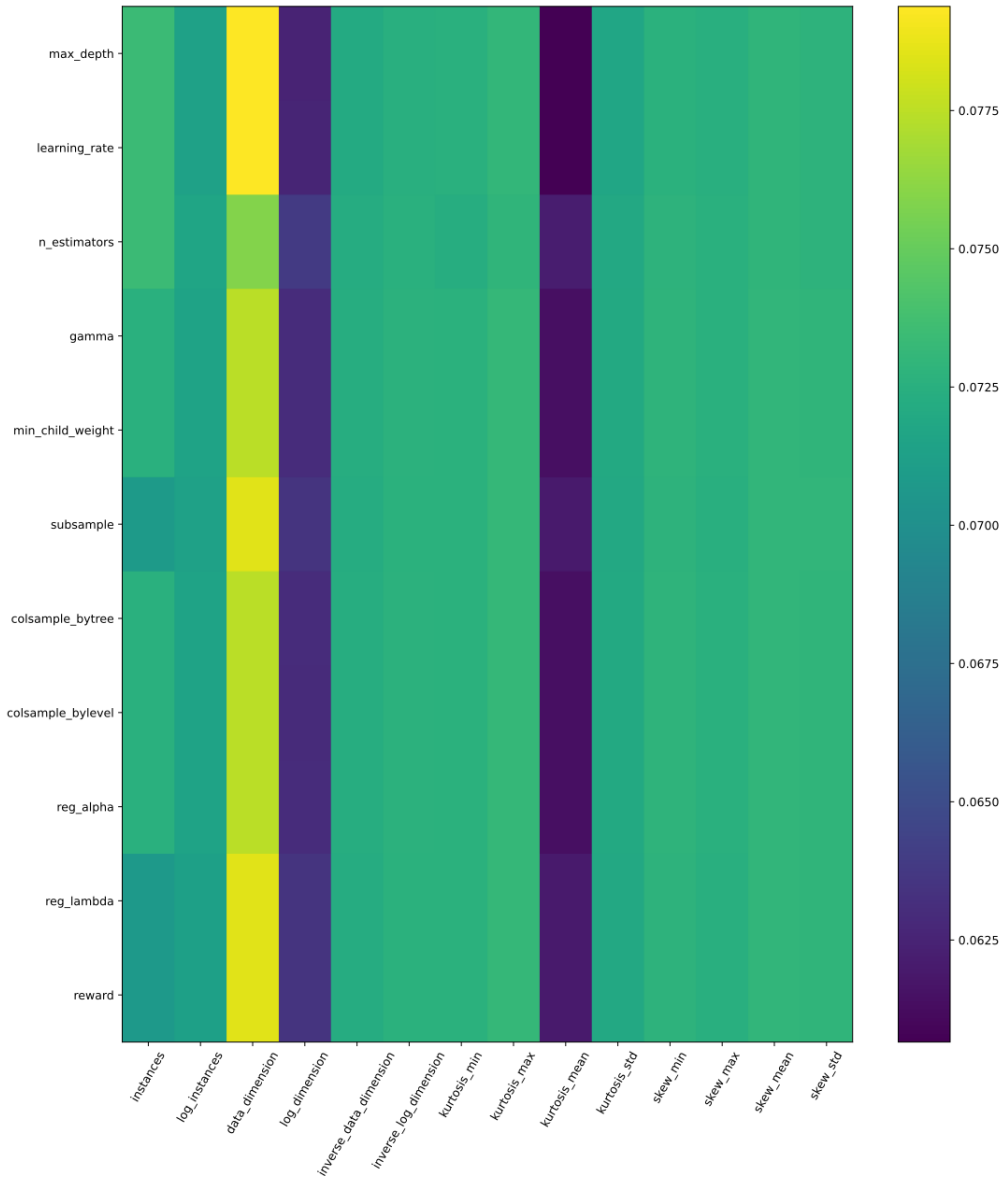


Figure 3.18: First Iteration Head-2 Attention Plot

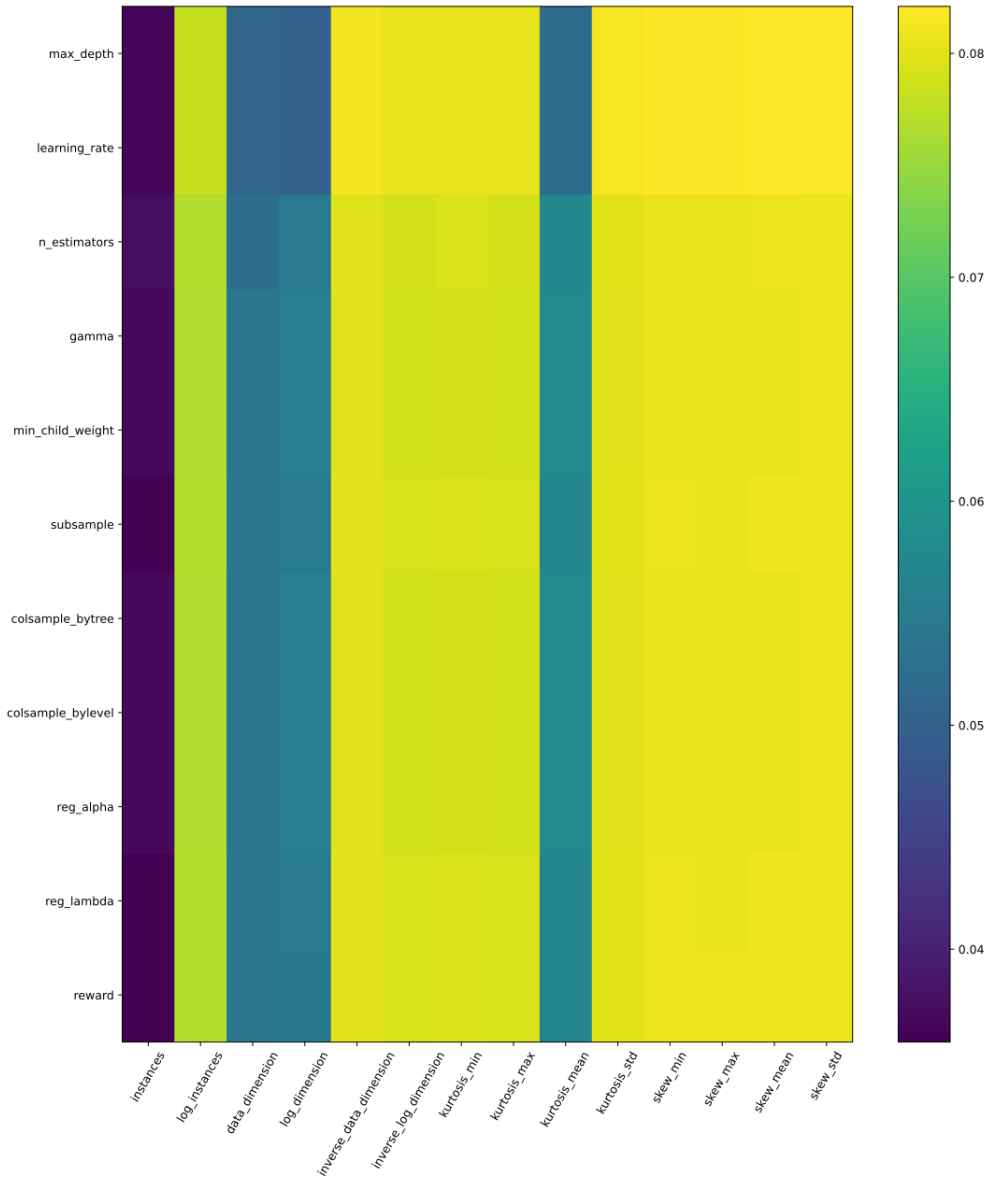


Figure 3.19: First Iteration Head-3 Attention Plot

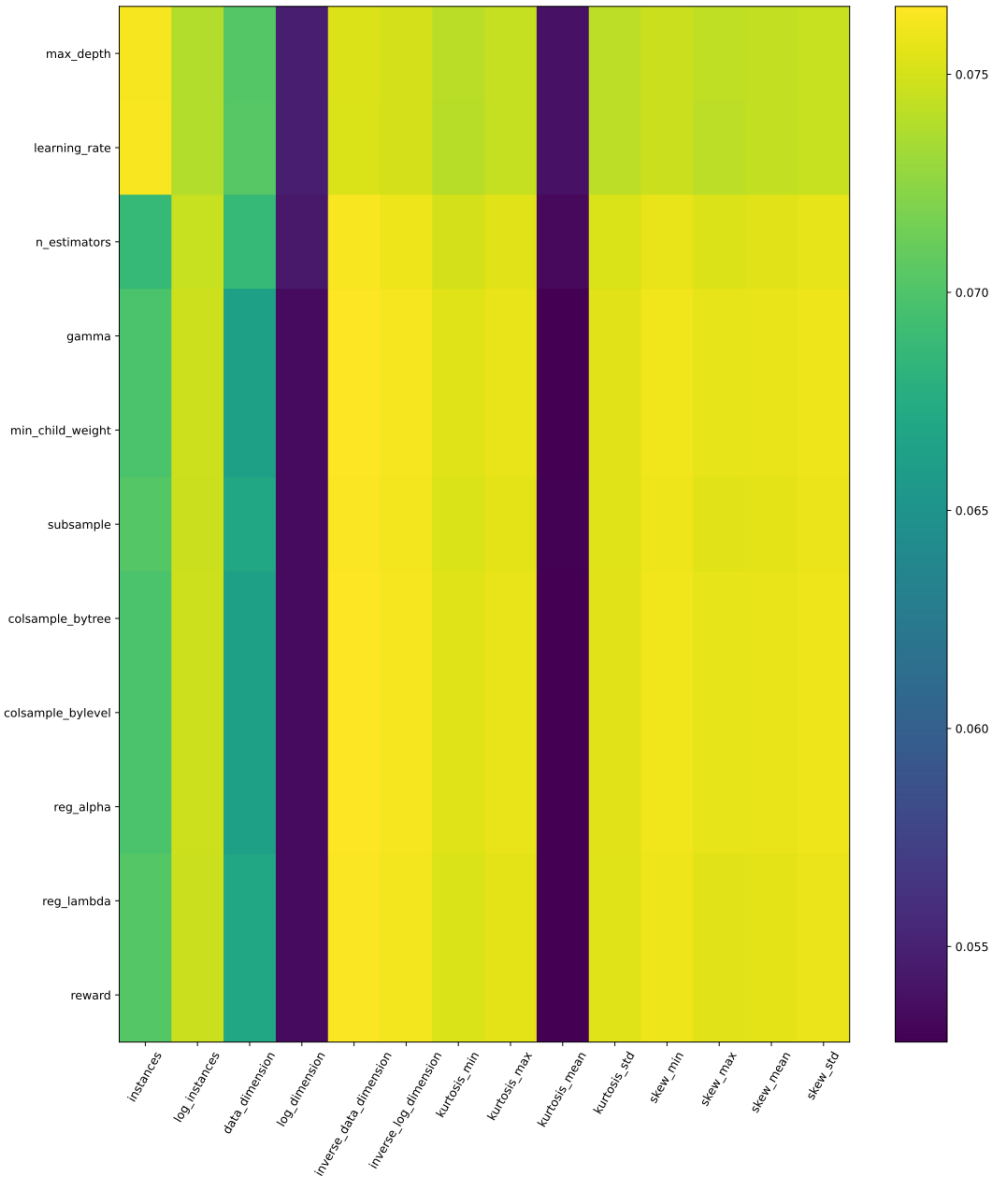


Figure 3.20: First Iteration Head-4 Attention Plot

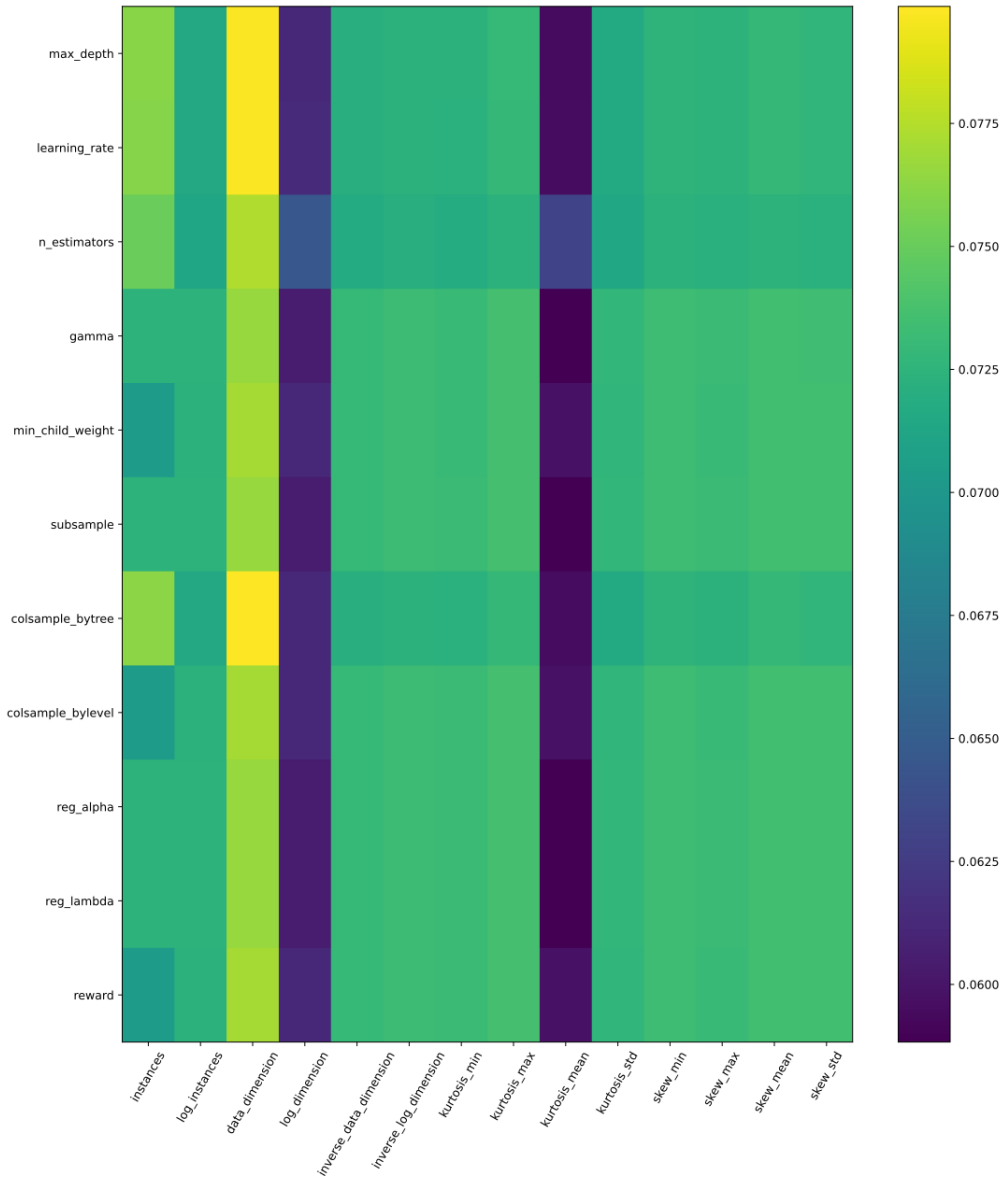


Figure 3.21: Second Iteration Head-2 Attention Plot

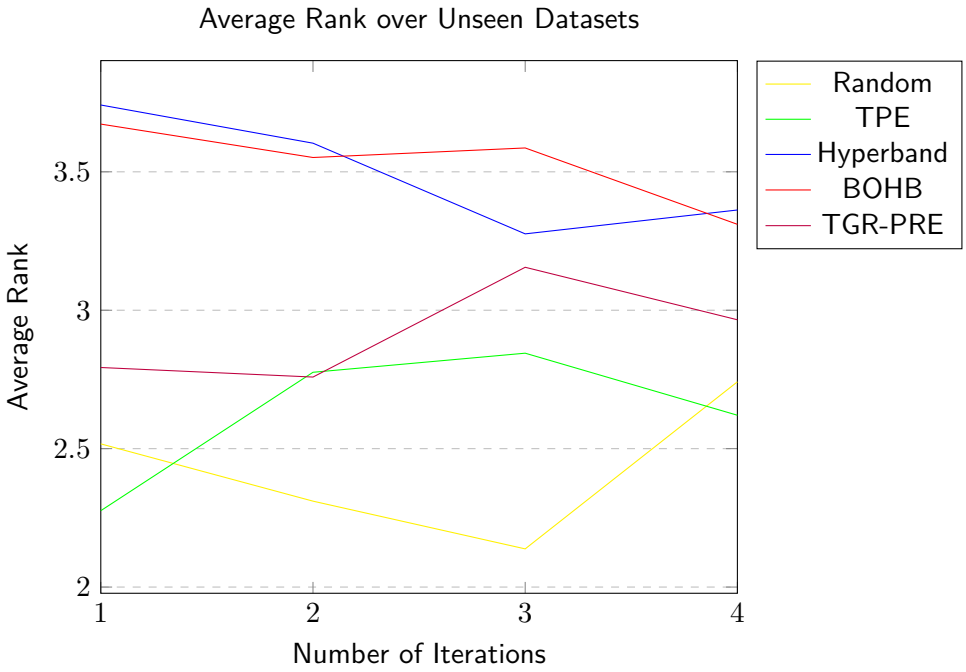


Figure 3.22: Average Rank on Unseen sets accuracy

4

Anomaly Detection

4.1 Context

We begin by defining a time sequence. A time sequence is a set of independent and identically distributed consecutive observations along a time frame, formally, $X = \{\mathsf{x}_t\}_{t=1}^{T}$, where T is the the number of time steps and x_t is the value of the sequence at a given time t. Note that, x_t can be composed of multiple features, that is, $\mathsf{x}_t \in \mathbb{R}^F$, where F is the number of features.

The main approach taken with reconstruction based anomaly detection methods, consists fundamentally of two parts: first we encode a given sequence into the latent space, which has a lower dimension than the original space; second, we decode the encoded latent representation to produce a reconstruction of the original sequence. Since auto encoders are trained to represent normal data, anomalous data won't be projected near the normal samples, which, in turn, will generate malformed reconstructions.

Anomaly detection plays an auxiliary, but important, role on this work, as we will discuss in the next section. During this section we limit ourselves to explore the use of transformers on the sequences' encodings, both for univariate and multivariate time sequences.

4.2 Models

4.2.1 Univariate

Based on [Per18] work, namely the variational self attention mechanism, we developed a more transformer like architecture. Concretely, we follow the same reasoning to obtain the contexts' mean, μ_{c_t} and standard deviation, σ_{c_t}. Nonetheless, we introduce a couple of changes in what concerns sequence encoding and deterministic attention. We will now provide a full description of the partial transformer architecture. First, we begin by splitting the architecture into four modules: base encoder, z encoder, c encoder and decoder.

Base Encoder

The base encoder, highlighted with green in figure 4.1, receives a corrupted sequence, $\tilde{\mathsf{x}}_t$, that is obtained through a noising process, $\tilde{\mathsf{x}}_t = \mathsf{x}_t + n$, where $n \sim \mathcal{N}(0, \sigma_{\mathsf{x}})$. Said sequence is then processed by a bidirectional LSTM layer. Here we have the first differences: we do not derive the sequence encoding from the LSTM's last hidden state, instead we pass all the hidden states trough the self attention mechanism, which will then be used to derive the sequence encoding in the z encoder. The outputs of the attention mechanism are then piped into the z encoder and c encoder, as figure 4.1 suggests.

z Encoder

Regarding the z encoder, highlighted in figure 4.1 with red, it takes as input the output of the base encoder, and begins by applying a Feed Forward layer, which, if paid due attention, completes the encoder stage of the full transformer architecture. Having this representation, we pass it trough an average self attention layer in order to obtain a generalized representation of the sequence, from which we derive the normal parameters, μ_z and σ_z. The mean is computed from a fully connected layer with linear activation function, whereas the standard deviation comes from a fully connected layer with soft-plus activation function, to ensure it is parametrized by a non-negative smooth function, as described in [Per18].The latent variable is then sampled from the approximate posterior $z \sim \mathcal{N}(\mu_z, \sigma_z^2 I)$ using the reparametrization trick:

$$z = \mu_z + \sigma_z \odot \epsilon$$

where $\epsilon \sim \mathcal{N}(0, I)$, which enables the model to backpropagate the gradients through μ_z and σ_z. while keeping the stochasticity of z.

The use of the generalized representation to derive the sequence's latent variable comes from the following reasons [FZL+21]: firstly, using only the last hidden states does not summarize the entire sequence; secondly, it is not useful if we want to maintain long-term knowledge. For this reasons, we summarize all the information into one vector using the average self attention layer.

c Encoder

The c encoder, highlighted with orange in figure 4.1, receives the attention outputs for the purpose of producing a context vector using a variational approach. That is, from the attention outputs, the deterministic attention, we apply two fully connected layers as way to obtain the contexts' mean μ_{c_t} and contexts' standard deviation σ_{c_t}. These fully connected layers follow the same pattern as the ones discussed in the z

encoder. Furthermore, the context vectors are sampled from the approximate posterior $c_t \sim \mathcal{N}(\mu_{c_t}, \sigma_{c_t}^2 I)$. Moreover, the context vectors share the dimension of the latent variable, specifically, $d_{c_t} = d_z$.

Decoder

Finally, before the decoding phase, we concatenate both the latent representation z, repeated across time steps, and the context vector c_t. The decoder consists of a bidirectional LSTM that will take the concatenated representation and output a sequence of hidden states. For each hidden state we, once again, apply two fully connect layers, one to obtain the mean μ_{x_t} and one to obtain the diversity b_{x_t}, the parameters of a Laplace distribution. As a consequence of using a Laplace distribution, the training will aim to minimize an ℓ_1 reconstruction loss $\propto \|x_t - \mu_{x_t}\|_1$, instead of a ℓ_2 reconstruction loss $\propto \|x_t - \mu_{x_t}\|_2^2$. Moreover, the minimization of a ℓ_1-norm promotes sparse reconstruction errors [Per18].

4.2.2 Multivariate

The last stages of the Multivariate Transformer matches the Univariate Transformer, namely, the z encoder, c encoder and the decoder. The two differ the most when it comes to the sequence encoding. Here, we apply a full transformer architecture to combine both feature and time encoding, inspired by the multivariate anomaly detection method presented in [ZWD+20]. Here the transformer is though as a data enriching mechanism, since we are enriching the time sequence with feature sequence information.

Since a multivariate sequence carries multiple feature information, it is of interest to analyze both the feature sequence and the time sequence. That is, if $X \in \mathbb{R}^{T \times F}$ represents the original sequence, which we call the time sequence, $X' = \{\mathsf{x'}_f\}_{f=1}^{F} \in \mathbb{R}^{F \times T}$ represents the feature sequence. If we consider X to be a matrix, X' would be its transpose, $X' = X^\top$. We begin by discussing the feature sequence encoding.

Feature Encoding

We begin by applying a Bidirectional LSTM to X', funnelling the produced hidden states to a Multi-Head Graph Attention Layer. In this case, this type of attention is the most useful for the following reasons: first we are analysing feature data, rather than time-step data; second, we want to infer possible relations between features, specifically, the compatibility of each feature to one another. For this reasons, we use a fully connected graph mask to model the dependencies. The remaining layers are what is expected of a transformer architecture, the result of the attention module is processed by a Feed Forward layer followed by a Normalization layer. This concludes the feature sequence encoding phase.

Time Encoding

The time encoding clearly matches the traditional decoder of the transformer architecture. We begin by encoding the sequence by means of a Bidirectional LSTM layer, whose produced hidden states are further processed by a Multi-Head Attention Layer. The result from the attention module is now merged with the feature encoding result, by means of another Multi-Head Attention Layer. Here we enrich the time sequence with the feature encoded information, as a way to not jeopardize the information a multi-feature can carry. The following layers are the typical Feed Forward and normalization layers.

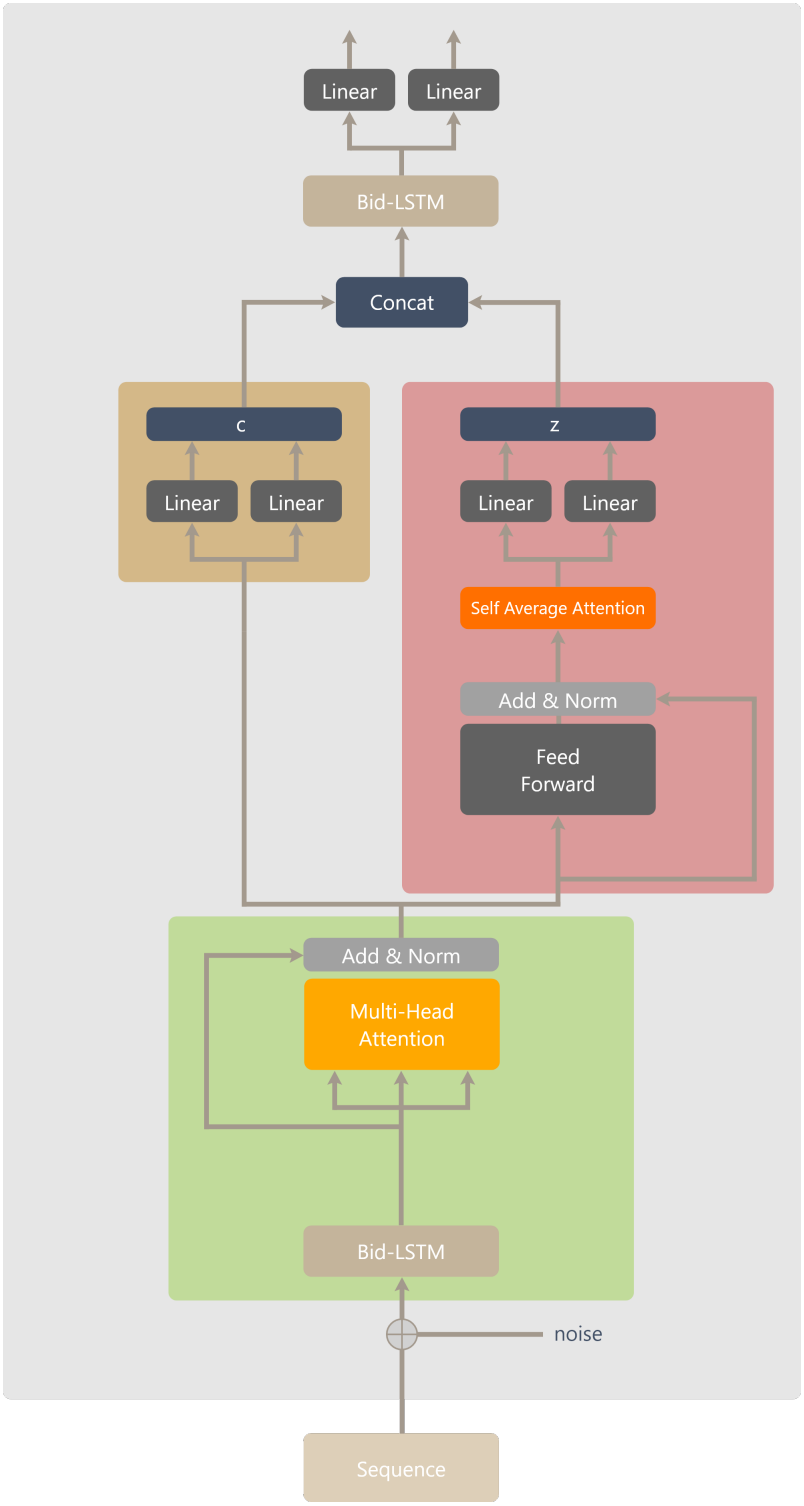


Figure 4.1: Univariate Transformer

The final layers of the architecture are identical to the univariate case. The time encoding output is piped to the same z encoder and c decoder, followed by the already described decoding phase.

4.3 Detecting Anomalies

In the unsupervised setting, the one of most interest for this work, we do not follow the latent space anomaly detection method, whose main interest relies on identifying an anomaly based on its latent space representation by means of a classifier, for example. Starting from the premise that the models are trained to reconstruct and represent normal data, when we feed the model a sequence that differs from the learnt normality, its reconstruction will be poor when compared to normal sequences. Since we are using a VAE, it learns to reconstruct the distribution parameters of the input sequence, instead of the sequence itself. This type of reconstruction allows us to use probability tools to determine if an observed sequence is anomalous or not. A procedure of interest is the one used in [Per18] works and first described by [AC15]. Here, the reconstruction probability is an estimate of the reconstruction term of the VAE loss by means of Monte Carlo integration:

$$\mathbb{E}_{\mathsf{z}_l \sim q_\phi(\mathsf{z}|\mathsf{x})}[\log p_\theta(\mathsf{x}|\mathsf{z})] \approx \frac{1}{L}\sum_{l=1}^{L} \log p_\theta(\mathsf{x}|\mathsf{z}_l) \tag{4.1}$$

In order to compute the reconstruction probability, we execute the following method: first we obtain the posterior mean, μ_z and standard deviation σ_z from encoder by feeding it the sequence x. Next, we draw L samples from the isotropic Gaussian distributions with the obtained parameters. Each of these samples is now piped into the decoder as a way to obtain the predicted distribution parameters of the reconstructed distribution. From this step, we take the log likelihood of the input sample x belonging to the reconstructed distribution, given the parameters obtained. For the last step, we simple average all the samples drawn. This algorithm is described in 7.

Algorithm 7: Reconstruction Probability

Input : $\mathsf{x} \in \mathbb{R}^{T \times F}$
Output: Reconstruction Probability $\in \mathbb{R}^T$

$(\mu_\mathsf{z}, \sigma_\mathsf{z}) \leftarrow$ Encoder(x)
for $l \in 1, \ldots, L$ **do**
  $\mathsf{z}_l \sim \mathcal{N}(\mu_\mathsf{z}, \sigma_\mathsf{z})$
  $(\mu_\mathsf{x}^l, \sigma_\mathsf{x}^l) \leftarrow$ Decoder(z)
  $\text{score}^l \leftarrow \log p(\mathsf{x}|\mu_\mathsf{x}^l, \sigma_\mathsf{x})$
Reconstruction Probability $\leftarrow \frac{1}{L}\sum_{l=1}^{L} \text{score}^l$
return *Reconstruction Probability*

Note that, since we are using the log likelihood, the anomaly score corresponds to the negative of the obtained results. This means that, the lower the reconstruction probability, the higher the chances of having encountered an anomaly.

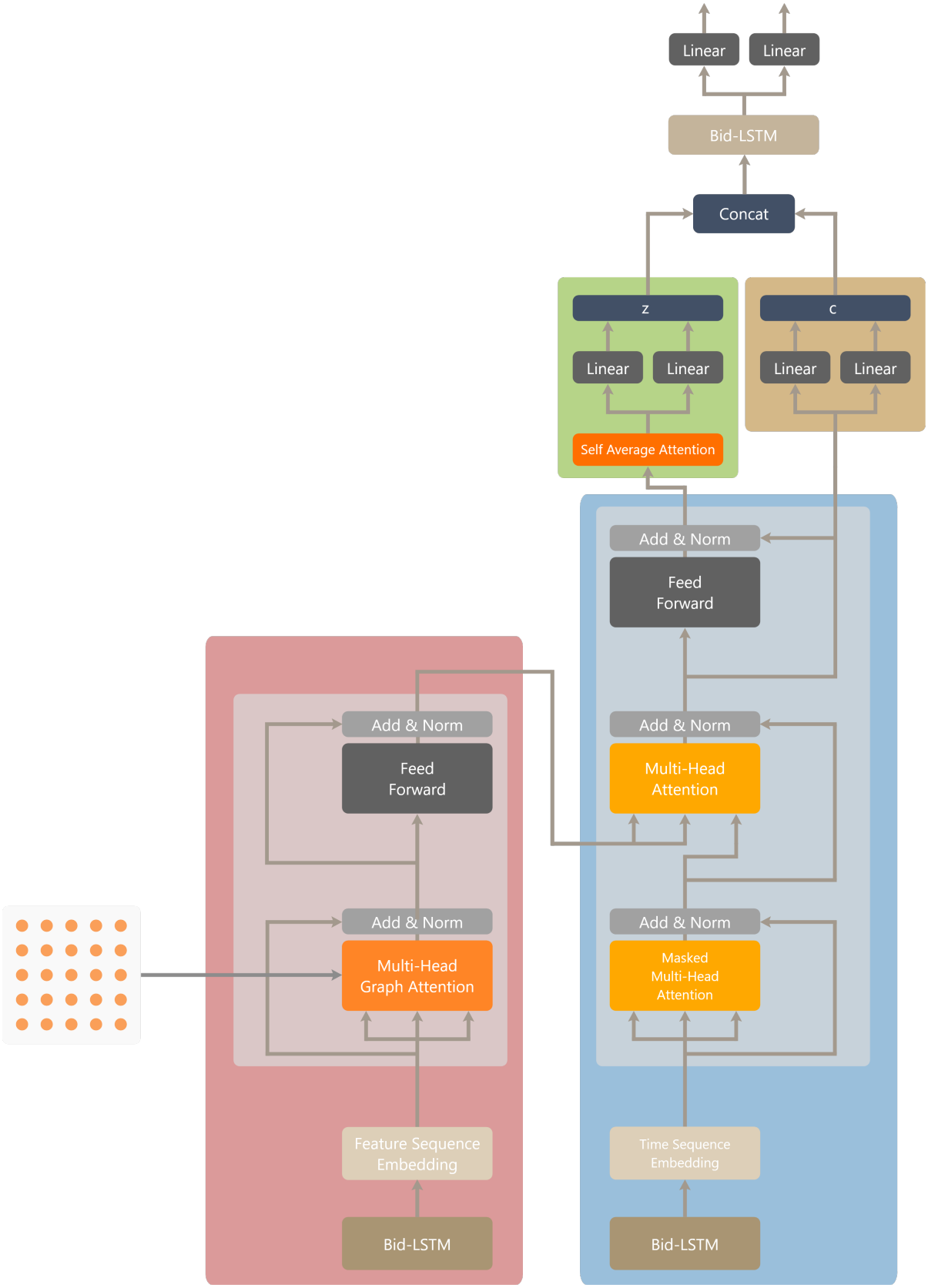


Figure 4.2: Multivariate Transformer

4.4 Dataset

For our experiments, we will be using a solar dataset [1] that contains meteorological data from the HI-SEAS weather station collected during the course of four months, starting in September 2016 and ending in December of the same year. The dataset consists of the following fields:

- UNIX time;
- date, following the yyyy/mm/dd format;
- time, following the 24 hour format hh/mm/ss;
- radiation;
- temperature;
- pressure;
- humidity;
- wind direction;
- wind speed;
- time of sun rise.

Each consecutive sample present in the dataset is taken with a 5 minutes time interval from the previous one, thus, a day consists of 288 samples.

We begin by combining the time and date fields to a unique field, date time, and dropping the fields UNIX time and time of sun rise, since they will be of no interest for our experiments. Since the remaining fields are not all in the same scale, we first normalize all features to the $[0, 1]$ scale using the following normalization technique:

$$z = \frac{z - z_{\text{min}}}{z_{\text{max}} - z_{\text{min}}}$$

From the resulting data, we retrieve a subset, called the normal dataset. This a dataset that consists of only normal sequences of data. What we consider to be a normal sequence is one that does not have sudden value drops or increases and is as smooth as possible, meaning that it gradually increases or decreases its values, preserving the seasonal properties of this type of data. These normal sequences are further sliced into consecutive sliding windows of 36 samples, which corresponds to 3 hours of observations. The normal dataset is further split in a training set and testing set following the usual split ratio of 80/20, respectively.

For the univariate case, we will only be using the radiation data, while on the multivariate case we will use radiation data and the fields that were felt out, namely: temperature, pressure, humidity, wind direction and wind speed.

[1] https://www.kaggle.com/dronio/SolarEnergy

4.5 Training

The two types of transformers, the univariate and multivariate, share the same parameters: a dimension of 256 and feed-forward size of 256 units; dropout rate of 0.1 and the output size is the same as the windows size, in this case, 36. This transformers have only one layer. We experimented with different number of attention heads, namely with 1, 4 and 8 heads. The models are trained for 1500 epochs with a batch size of 256, with a latent space dimension equal to 3.

Following the works of [Per18], we use the same loss function for a sequence x

$$\mathcal{L}(\theta, \phi; \mathsf{x}) = - \mathbb{E}_{\mathsf{z} \sim \tilde{q}_\phi(\mathsf{z}|\mathsf{x}), \mathsf{c}_t \sim \tilde{q}^a_\phi(\mathsf{c}_t|\mathsf{x})} \left[\log p_\theta(\mathsf{x}|\mathsf{z}, \mathsf{c})\right] \tag{4.2}$$

$$+ \lambda_{\mathsf{KL}} \left[\mathcal{D}_{\mathsf{KL}}(\tilde{q}_\phi(\mathsf{z}|\mathsf{x}) \| p_\theta(\mathsf{z})) + \eta \sum_{t=1}^{T} \mathcal{D}_{\mathsf{KL}}(\tilde{q}^a_\phi(\mathsf{c}_t|\mathsf{x}) \| p_\theta(\mathsf{c}_t)) \right] \tag{4.3}$$

where $\tilde{q}_\phi(\mathsf{z}|\mathsf{x})$ is the approximate posterior, $\tilde{q}^a_\phi(\mathsf{c}_t|\mathsf{x})$ is the approximate posterior of the context vectors, $p_\theta(\mathsf{z})$ is the prior over z and $p_\theta(\mathsf{c}_t)$ is the prior over the context vectors.

However, in this work, since the prior is learnt, $p_\theta(\mathsf{z})$ is not the standard Gaussian distribution $\mathcal{N}(0, I)$, but the isotropic Gaussian distribution with parameters obtained from the prior learning, but $p_\theta(\mathsf{c}_t)$ is still the standard Gaussian distribution. The prior is obtained in the same manner as the approximate posterior, using the output of the self average attention network to derive its variational parameters, as well. The hyper-parameter λ_{KL} evolves following a logistic annealing, starting at 0 at the beginning of training and slowly getting to 1 at the mid stages of training. The hyper-parameter η is equal to 0.01.

4.6 Discussion

4.6.1 Univariate

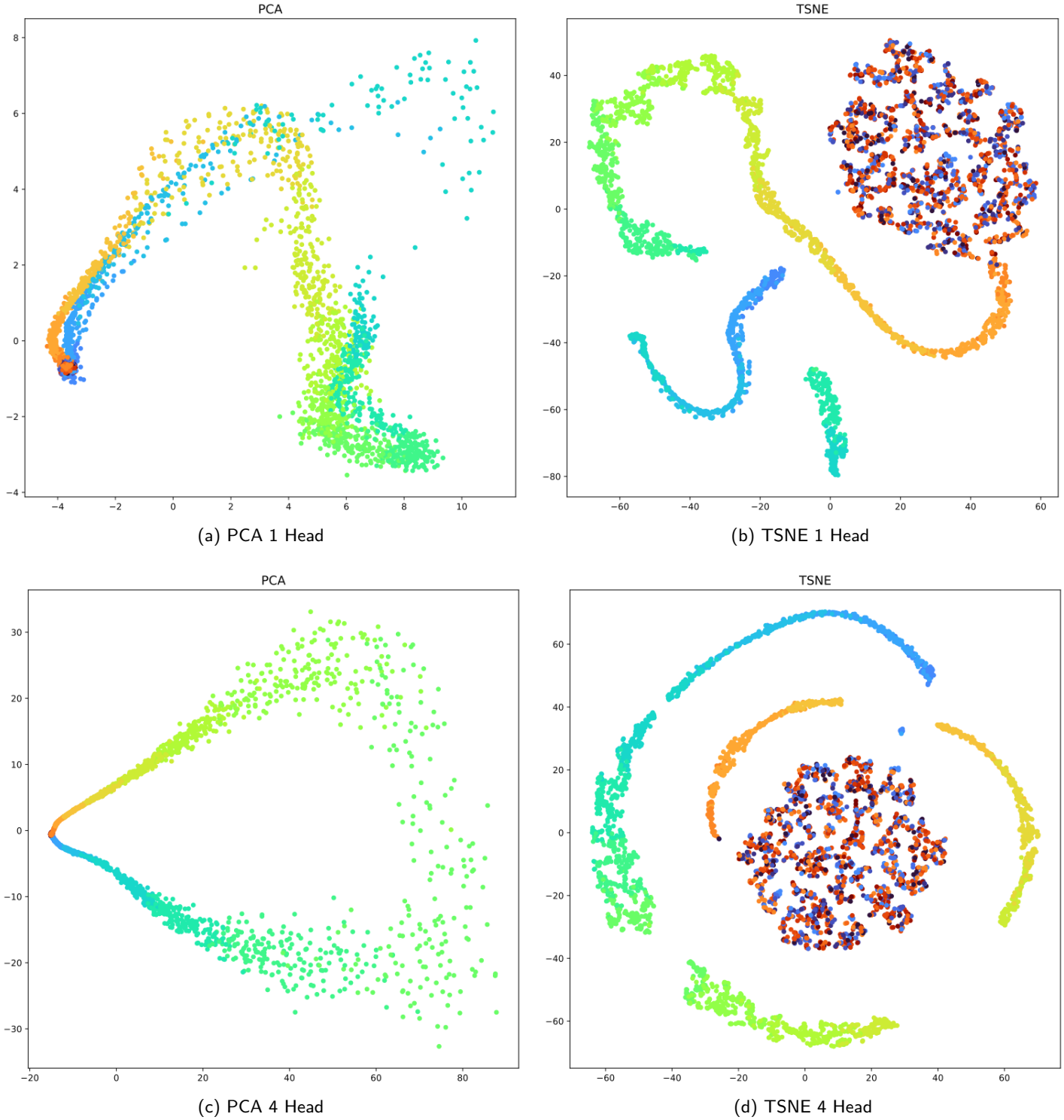
Latent Space Analysis

Looking at the produced latent space from the training set, figure 4.3, we clearly see that the mapped sequences form a trajectory based on the hour of the day. This phenomenon is known as time gradient, as also noted in [Per18] works. At the early and late stages of the day the sequences are more condensed, while at around mid-day the sequences are sparser, as we can conclude from the PCA and TSNE plots.

Another interesting fact to be noted is with a different number of attention heads, we get different time gradients, which will impact the anomalies detected. With 1 head attention, we get sparser representations, while with 4 and 8 heads the representations tend to be less sparser and more symmetric, favoring the seasonal nature of the radiation. As a consequence, this takes us to a better anomaly detection, as we will see in the following sections.

Attention Plot Analysis

We shall now discuss the nature of the attention plots with and without anomalies. Here we present the attention plots of the univariate transformer model with 4 heads for a full day, 288 samples.



(a) PCA 1 Head

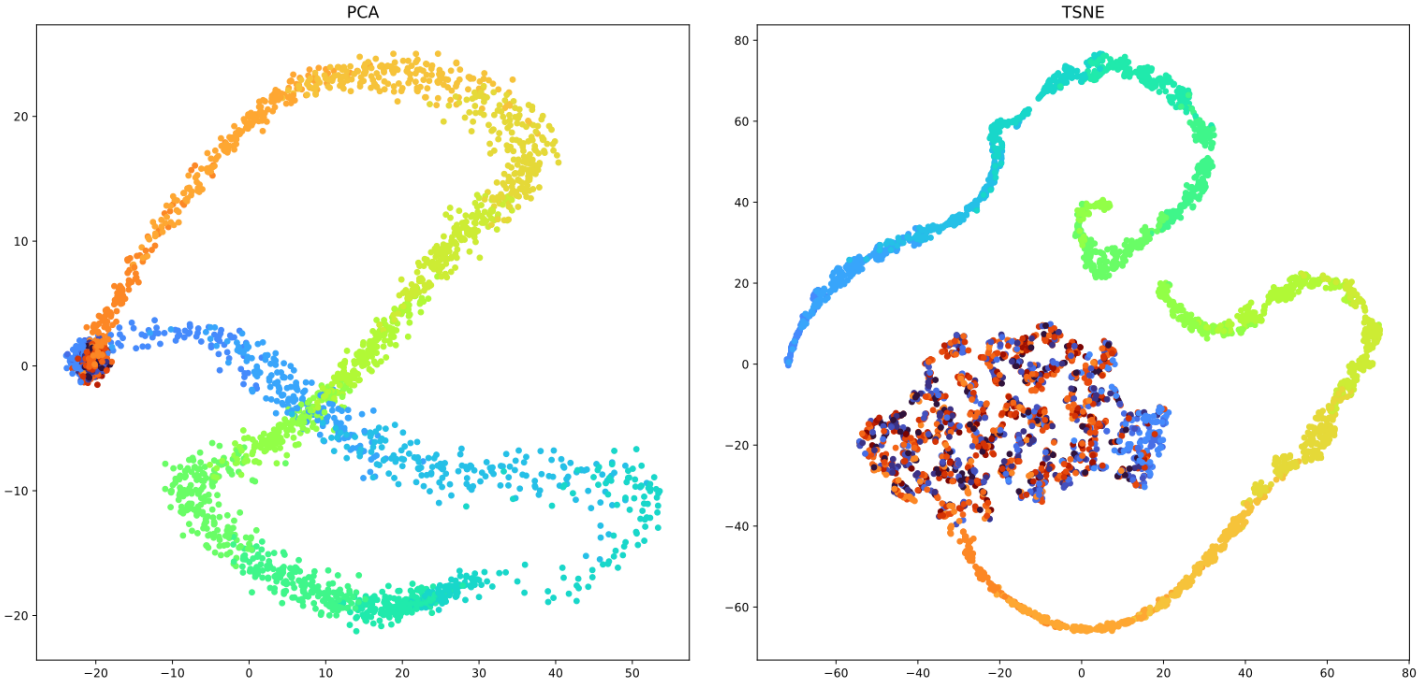
(b) TSNE 1 Head

(c) PCA 4 Head

(d) TSNE 4 Head

Figure 4.3: Latent Space Plots of Training Data with 1-head Attention

A normal attention plot, follows the traits of figure 4.4. The first head and third head focus on the compatibility of the same stages of day, while the second and forth head focus on the compatibility of different stages of the day. From the second head and fourth head attention plots, we can conclude that little compatibility exists between the early/late stages of the day with the mid-day stages, where the



(e) PCA 8 Head (f) TSNE 8 Head

Figure 4.3: Latent Space Plots of Training Data with Different Attention Heads

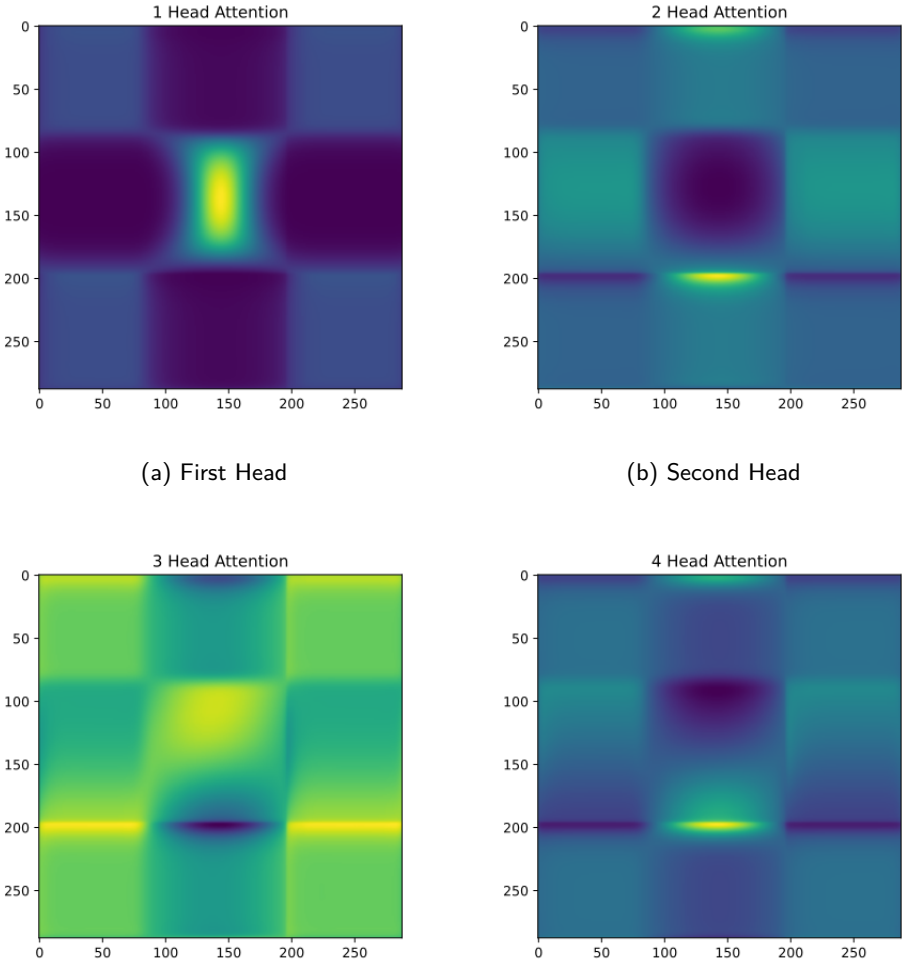
radiation is the highest.

The anomalies in this attention plots can be depicted very easily, figure 4.5. When an anomaly is present, the attention becomes less active in regions where one would expect high attention scores, the plots include lines crossing the plot and there is an increased distortion, mainly, in the third attention head. Therefore, by analysing the attention plot, we can have a clear visualization of the anomaly: heavier distortion and lines crossing the plot correspond to pronounced anomalies, while lesser lines and distortion correspond to subtle anomalies.

Detected Anomalies Analysis

Here we classified a time step as anomalous if its negative reconstruction probability is greater than 0. Furthermore, to compute the reconstruction probability, we used $L = 128$ samples. The number of attention heads influences the anomalies detected by the model, as suggested by figure 4.6. With 1 attention head the model is less sensitive leading to some anomalies being not detected, namely, right before 2016-11-09 is a good example of such case, having a very pronounced anomaly that was not detected. With 4 heads that is no such case, the model is much more sensitive catching the anomalies not caught by the 1 head model, even when the anomaly is subtle, as we can see by the detected anomaly right after 2016-11-09. The 8 head model catches some of the anomalies missed by the 1 head model, yet it isn't as much sensitive as the 4 head model is, leaving some clear anomalies out.

With the different number of heads we can already gain some control on the severity of the anomalies we want to detect, while the 1 and 8 model are less sensitive, reacting in the majority to pronounced anomalies, the 4 head model, being more sensitive, reacts to even small variations of a time step.



(a) First Head

(b) Second Head

(c) Third Head

(d) Fourth Head

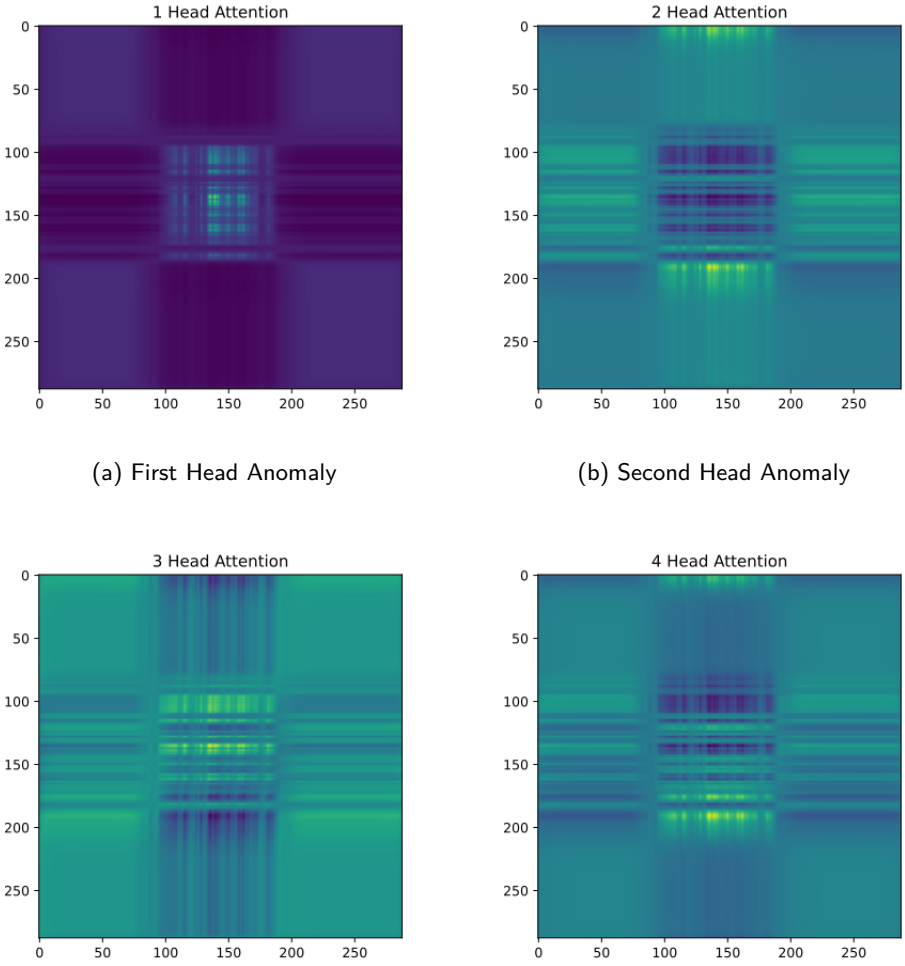
Figure 4.4: Attention Plot for Normal Data

4.6.2 Multivariate

Attention Plot Analysis

With the multivariate transformer based VAE, we can gain some insight at which features are more compatible with each other by looking at the encoder graph attention plot, figure 4.7. When dealing with normal sequences, all features are highly related to radiation, encoded as 0, and pressure, encoded as 2, being more related to pressure, as figure 4.7a suggests. By encountering an anomalous sequence, the attention paid to radiation increases, which can be an early indicator of said anomaly, figure 4.7b.

Looking now at the time encoder for a normal sequence, figure 4.8, we can promptly see that we have much more noise when compared to the univariate case. This is expected, since we are now considering more than one feature sequence. Nonetheless, there is some tendency maintained regarding the attention paid. The first head is still looking for the compatibility of the same stages of the day, while the second and fourth head are doing the opposite. However, the third head is now concentrated on a very late stage of the day, and the fourth head is focused on the relation between mid day and the early/late stages of the



(a) First Head Anomaly

(b) Second Head Anomaly

(c) Third Head Anomaly

(d) Fourth Head Anomaly

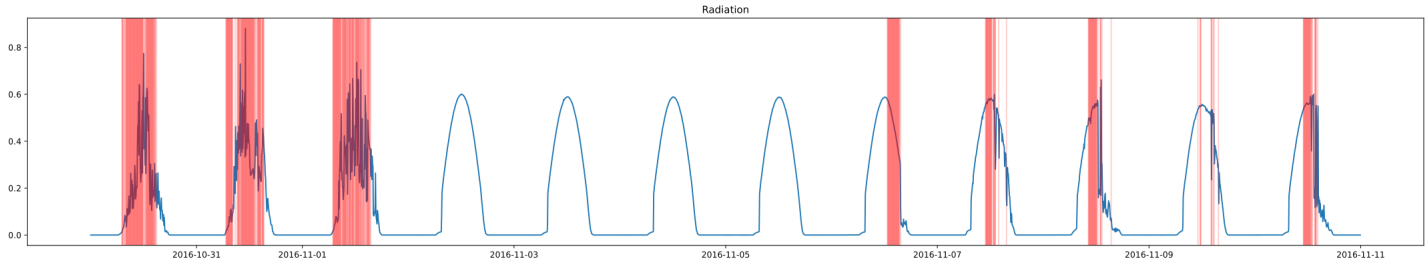
Figure 4.5: Attention Plot for Anomalous Data

day.

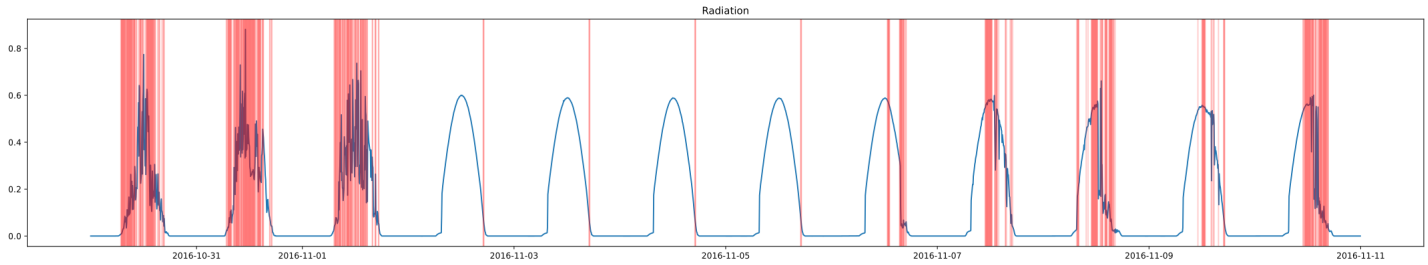
Analysing the time encoder attention plots, figure 4.9, for an anomalous sequence, as in the univariate case, the plots become more dashed and noisy. This is easier to see in the second and fourth head attention plots. The fourth head is not active in the mid-day, when comparing to the normal case, and the second head has much more noise and vertical dashed lines.

Finally, we must analyse the attention mechanism responsible for combining the feature encoding and the time encoding. Diving ourselves in the attention plots for normal sequences, figure 4.10, the attention heads focus in different features. While the first and fourth attention heads focus mainly into the 0, 2 and 3 features, which correspond to radiation, pressure and humidity, respectively, the second and third heads are more involved with how the remaining features interact with the observed sequences.

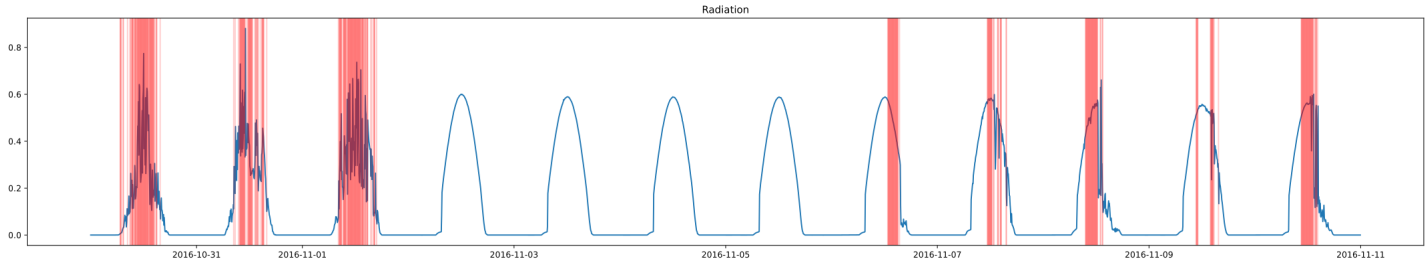
Having seen how a normal sequence is represented by the attention mechanism, we should expect some similarities when an anomaly sequence is at play, namely, lines crossing the plot and noisier plots. This assumptions are proved in figure 4.11. Furthermore, the first and fourth head are now more active around radiation, labeled as 0, as the graph encoder was. It should be the case, since this mechanism receives



(a) Radiation 1 Head

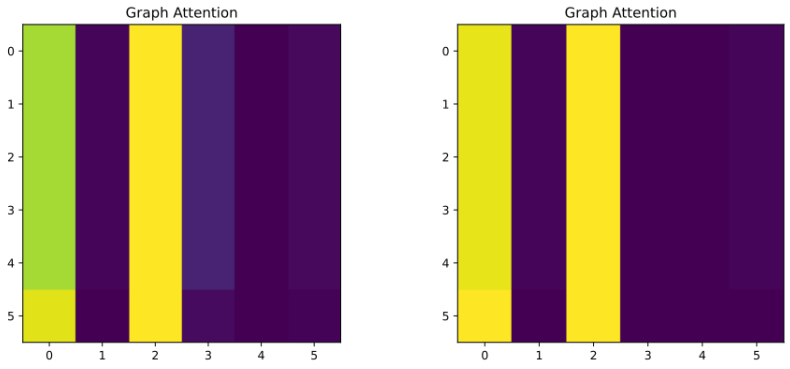


(b) Radiation 4 Head



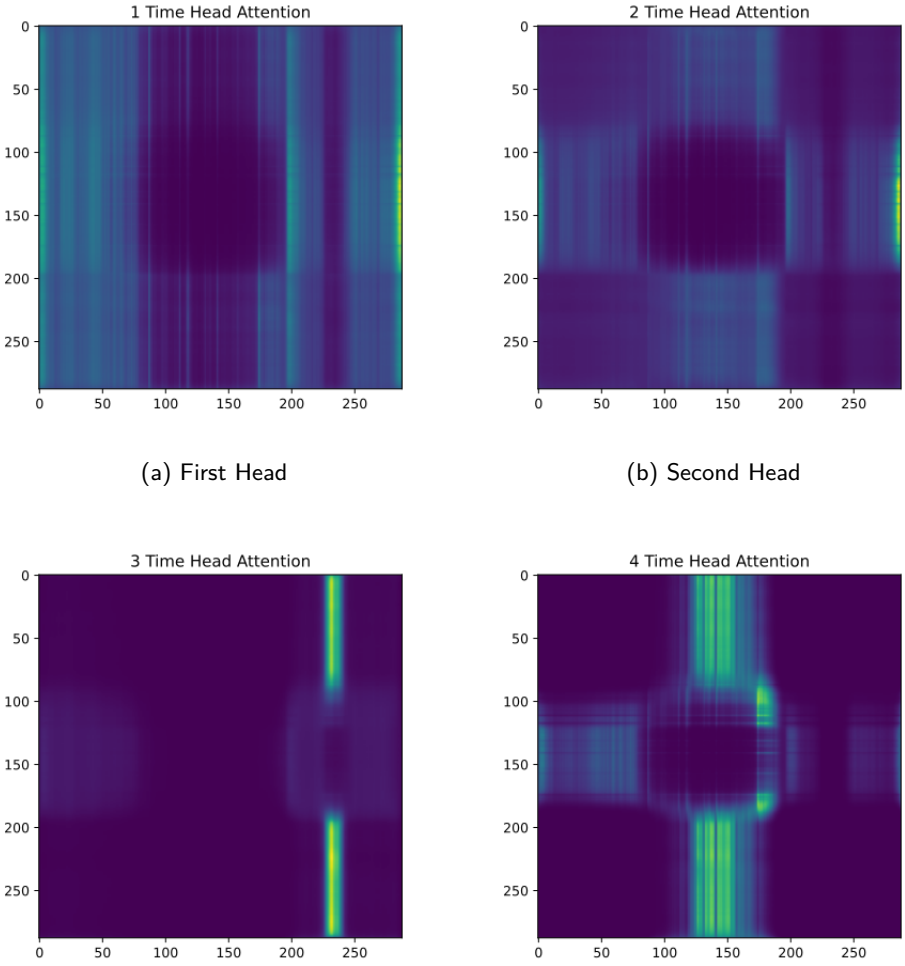
(c) Radiation 8 Head

Figure 4.6: Detected Anomalies for Univariate with various head



(a) Graph Attention Normal (b) Graph Attention Anomalous

Figure 4.7: Attention Plot Comparison Between Normal and Anomalous Features



(a) First Head

(b) Second Head

(c) Third Head

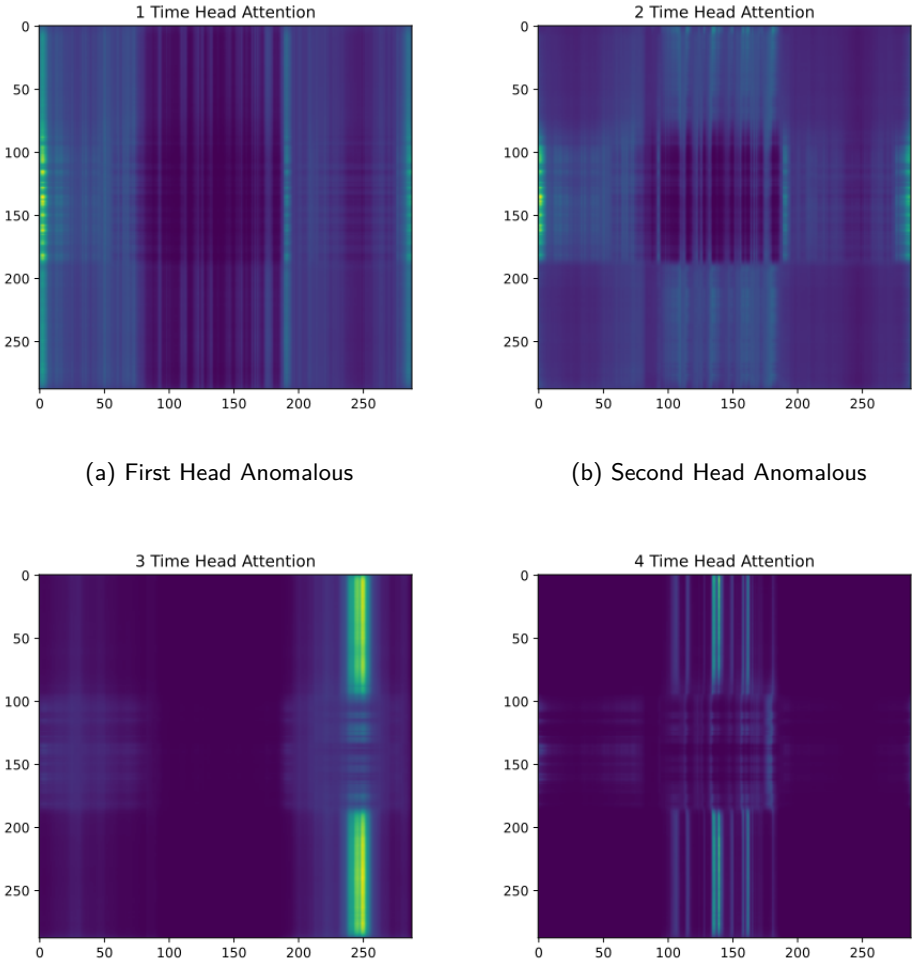
(d) Fourth Head

Figure 4.8: Attention Plot for Normal Data

information from the graph encoder.

Latent Space Analysis

Concerning the latent plots, figure 4.12, related to the multivariate case, they exhibit similarities with the univariate one, namely, the time gradient and the use of a different number of attention heads, yields different latent representations. However, the multivariate latent space is much more sparse than the univariate, which has a good explanation: we are dealing with more than one variable. Also, looking closely, there appears to be multiple trajectories, you can see in figure 4.12e a greater outer arc and a smaller inner arc, which is also present when using 4 heads, figure 4.12c. Some of the symmetry present in the univariate case is also lost, which can be consequence of the noise introduced by the remaining variables.



(a) First Head Anomalous

(b) Second Head Anomalous

(c) Third Head Anomalous

(d) Fourth Head Anomalous

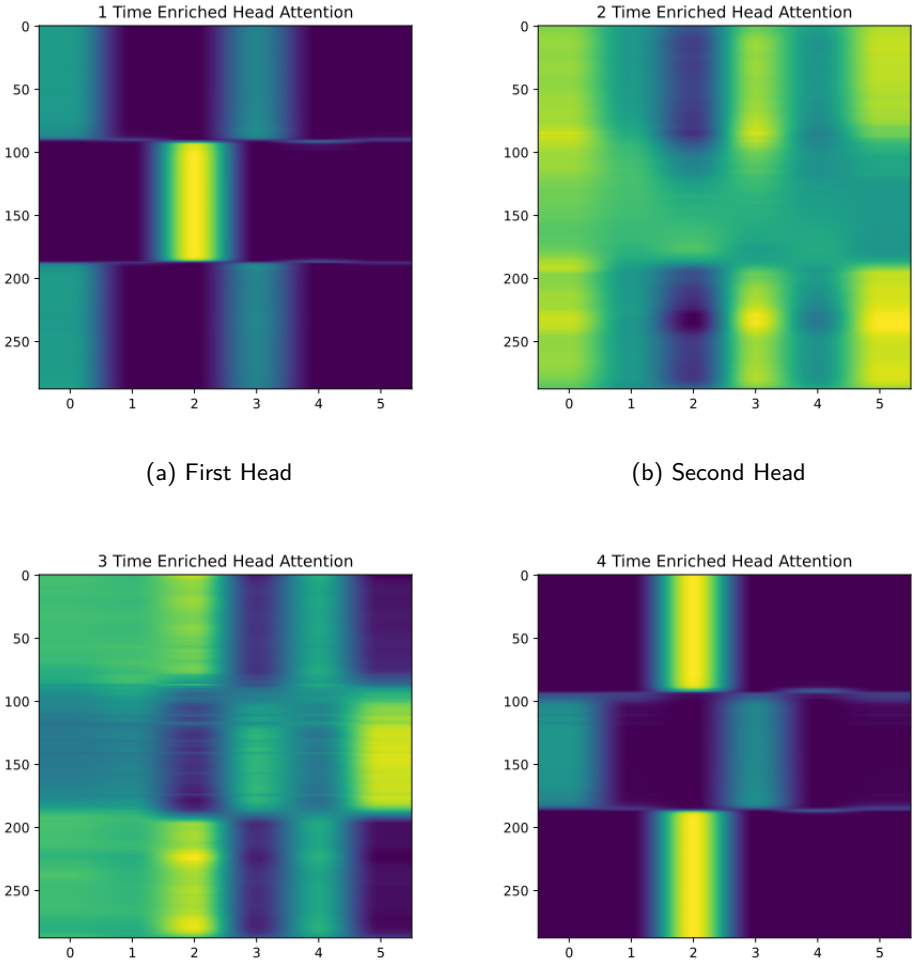
Figure 4.9: Attention Plot for Anomalous Data

Detected Anomalies Analysis

Here we follow the same procedure for detecting anomalies as in the univariate section. In addition, here we can depict which feature is anomalous in a given time-step, for the reconstruction probability of each feature can be obtained individually. In case of needing a general anomaly score, one can simply average the reconstruction probabilities of all features and take the negative.

In the univariate case, we concluded that the 4 head attention model was the most sensitive of all the tested models, and the 1 head was the less sensitive. Now, their roles are reversed, while the 8 head model still being the middle ground between the two. The 1 head model is much more sensitive to anomalous sequence, over-reacting in some cases where there isn't an anomaly present in some features. Take for example radiation right before date 2016-11-05 in figure 4.13. This reaction is probably provoked by the influence of the remaining features.

Contrary to the 1 head model, the 4 head model learnt to alert only in the case of when a major anomaly is at place, begin more robust to noise introduced by the other variables, as we can see from figures 4.14



(a) First Head

(b) Second Head

(c) Third Head

(d) Fourth Head

Figure 4.10: Attention Plot for Normal Data

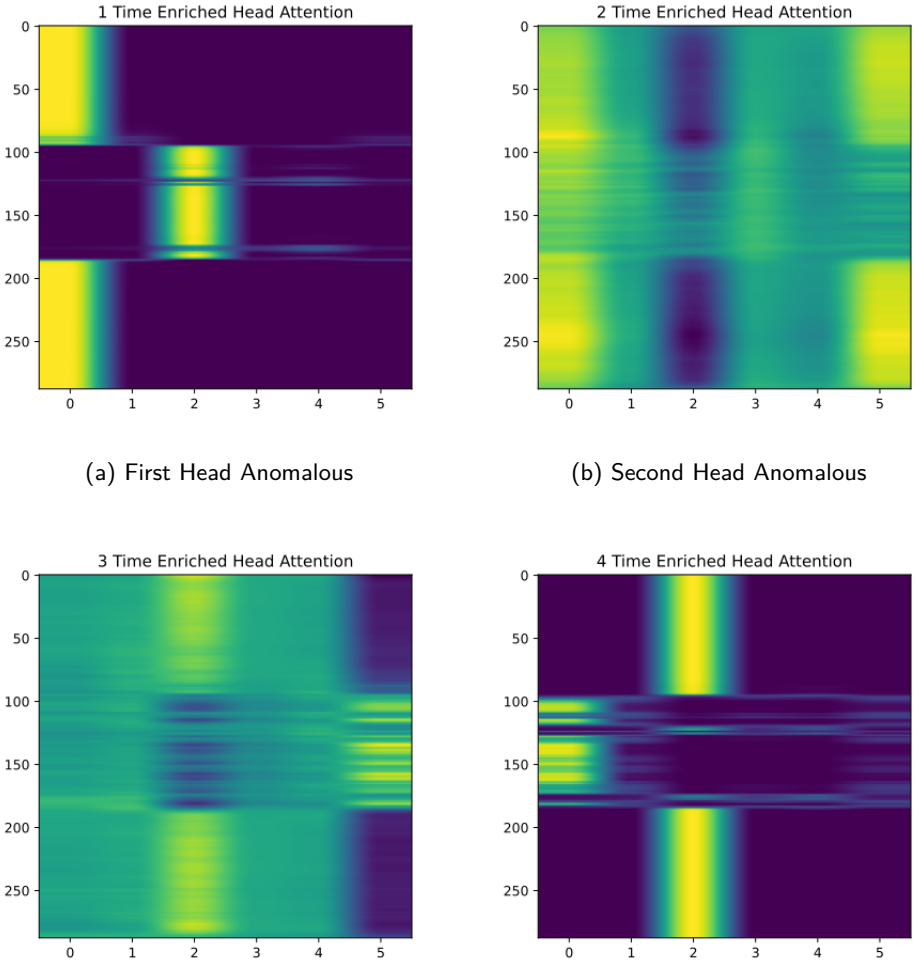
and 4.14.

The 8 head model, figures 4.15 and 4.15, suggests to be the safest choice here, since it reacts to major and subtle anomalies, but it doesn't overreact like the 1 head model does.

4.7 Remarks

In this section we extended the VAE architecture to take advantage of the transformer architecture, majorly the use of multi-head attention and its impacts on the sequence's representation over the latent space. We also consider all of the embeddings produced by the encoding layers to derive the latent representation, contrary to only considering the last embedding obtained from a LSTM network. Furthermore, we studied how the anomalies reflect themselves in attention plots and how the different number of heads affects the anomalies detected.

In addition, we experimented with the transformer architecture to enrich multivariable sequences with



(a) First Head Anomalous

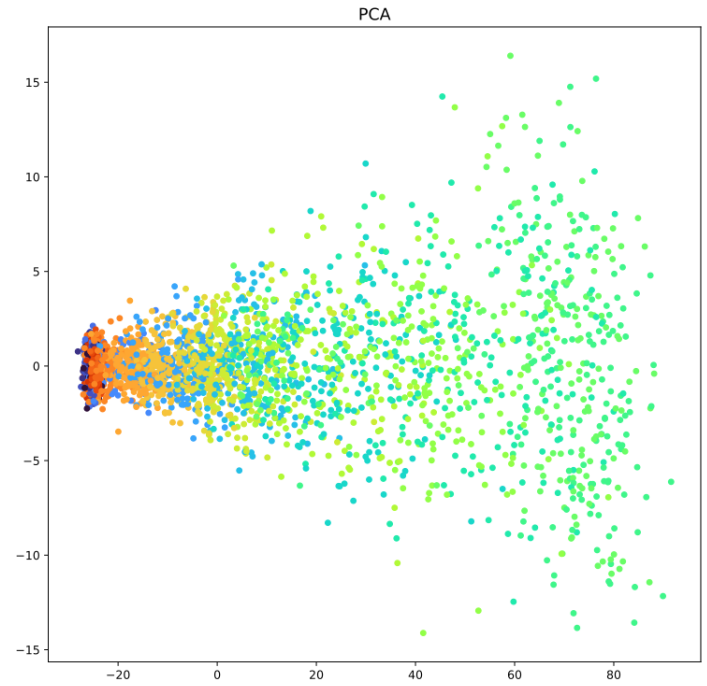
(b) Second Head Anomalous

(c) Third Head Anomalous

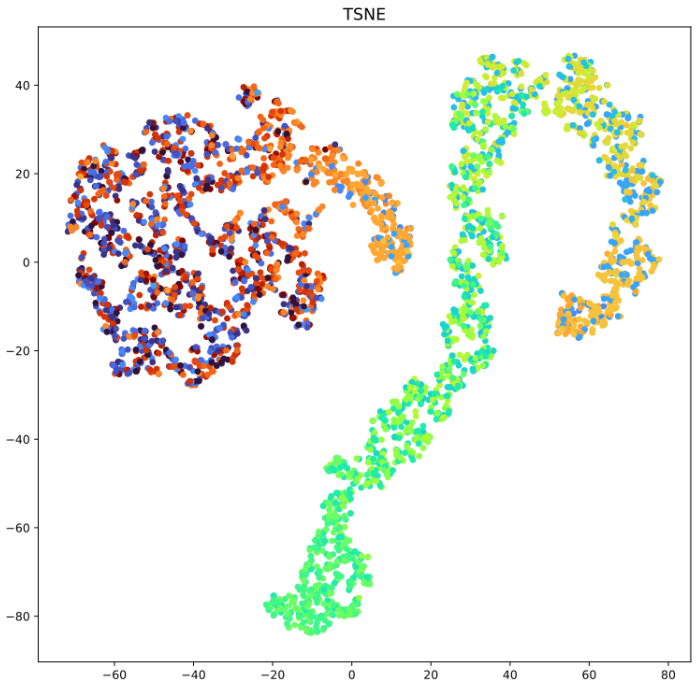
(d) Fourth Head Anomalous

Figure 4.11: Attention Plot for Anomalous Data

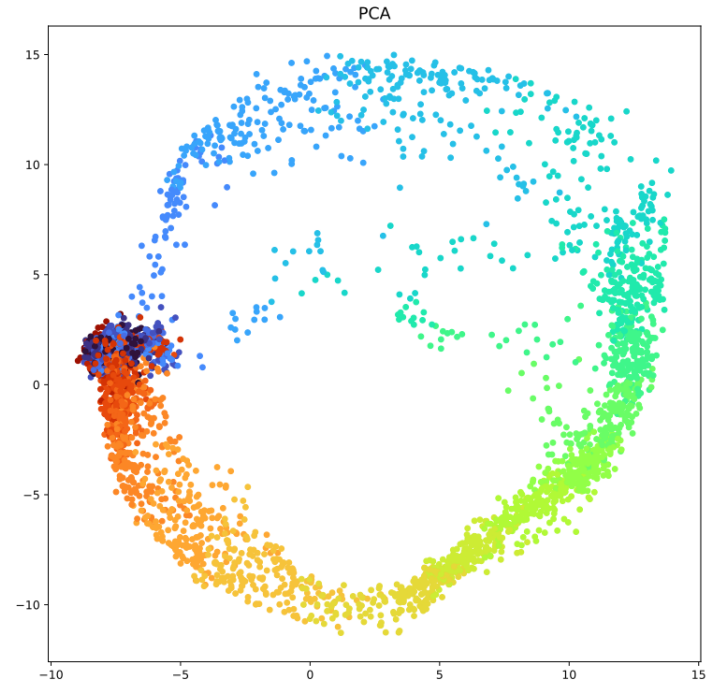
variable information, courtesy of the graph attention mechanisms, showing that from the graph attention plots we can already argue that an anomaly is occurring, opening up opportunities in the multivariate field.



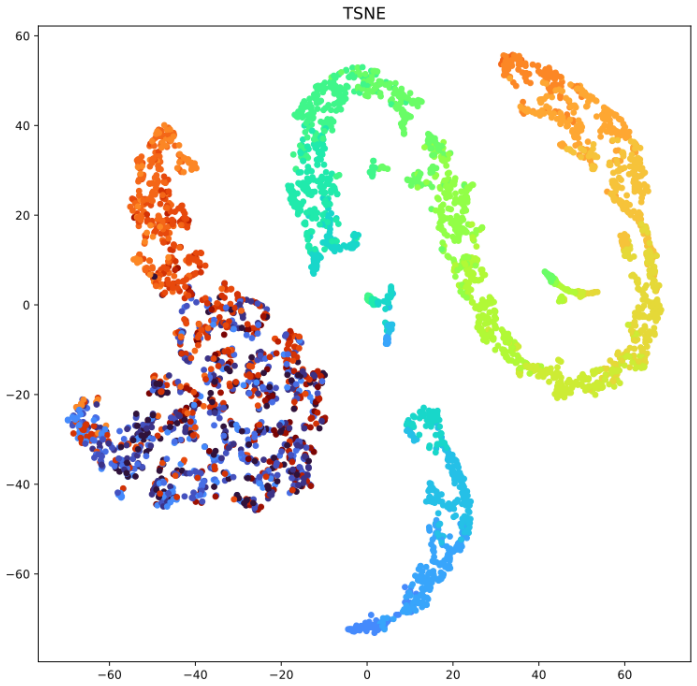
(a) PCA 1 Head



(b) TSNE 1 Head

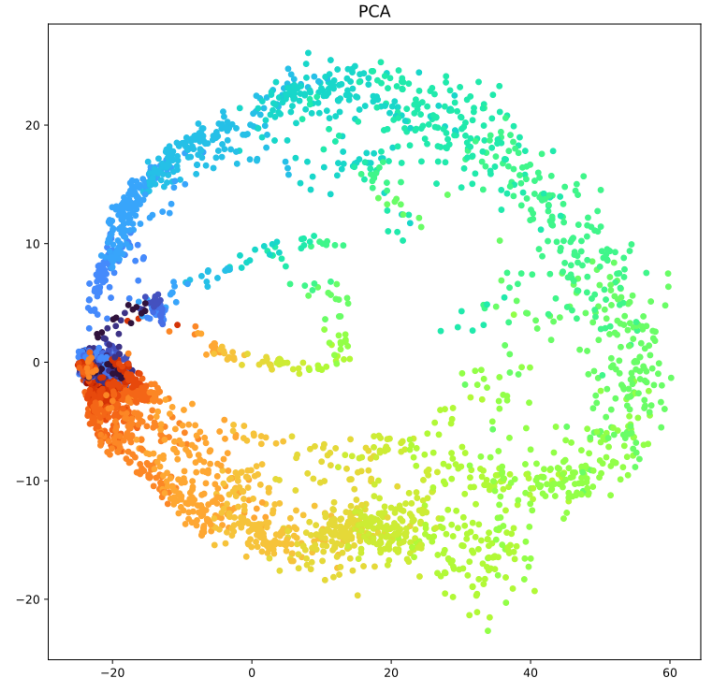


(c) PCA 4 Head

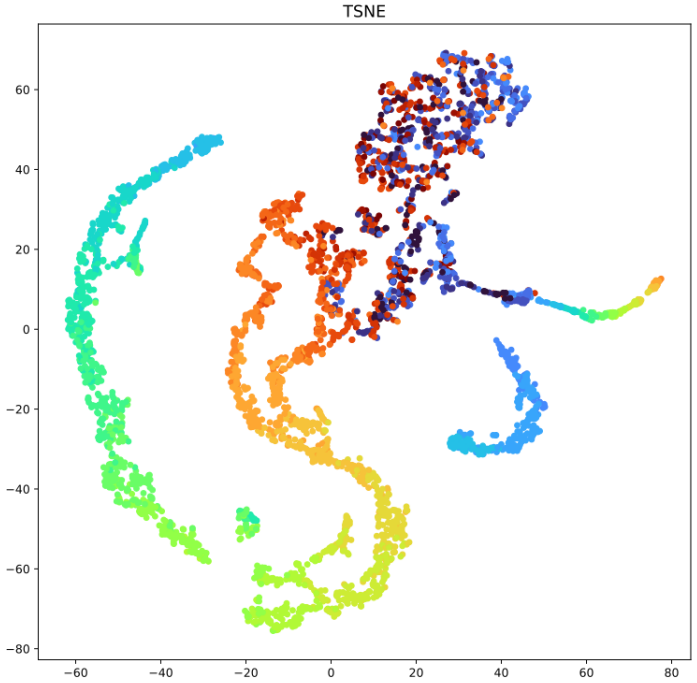


(d) TSNE 4 Head

Figure 4.12: Latent Space Plots of Training Data with Different Attention Heads

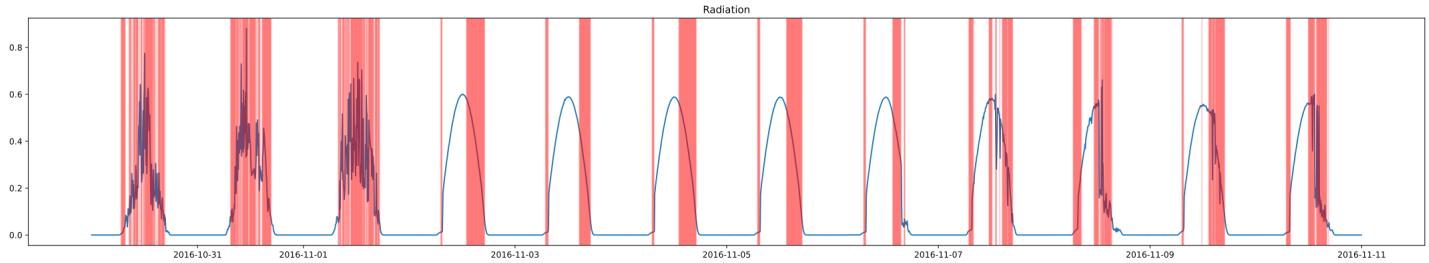


(e) PCA 8 Head

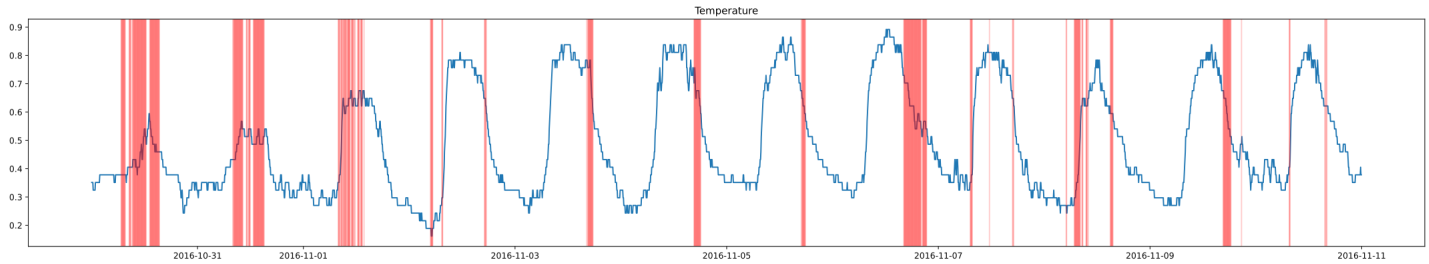


(f) TSNE 8 Head

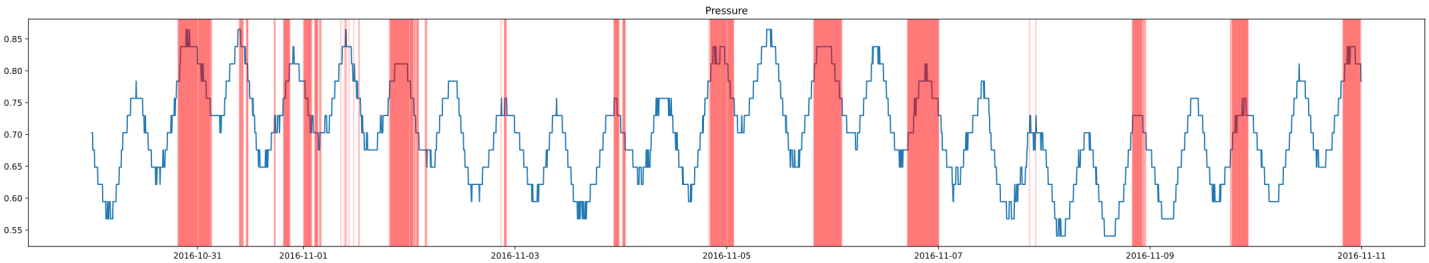
Figure 4.12: Latent Space Plots of Training Data with Different Attention Heads



(a) Radiation

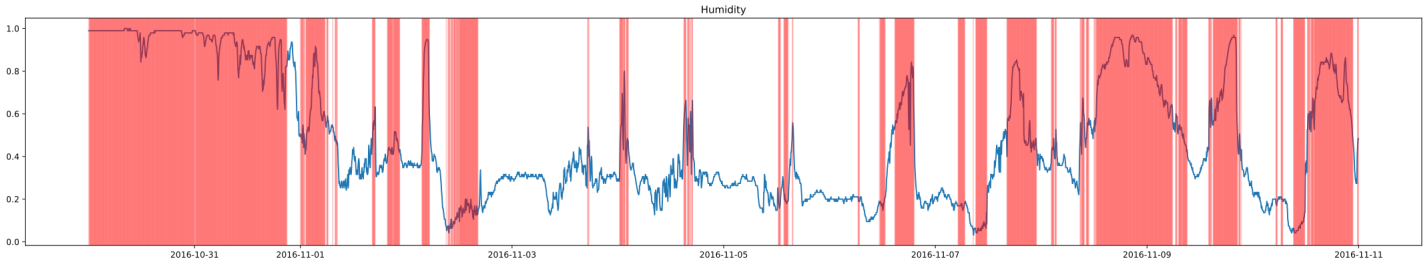


(b) Temperature

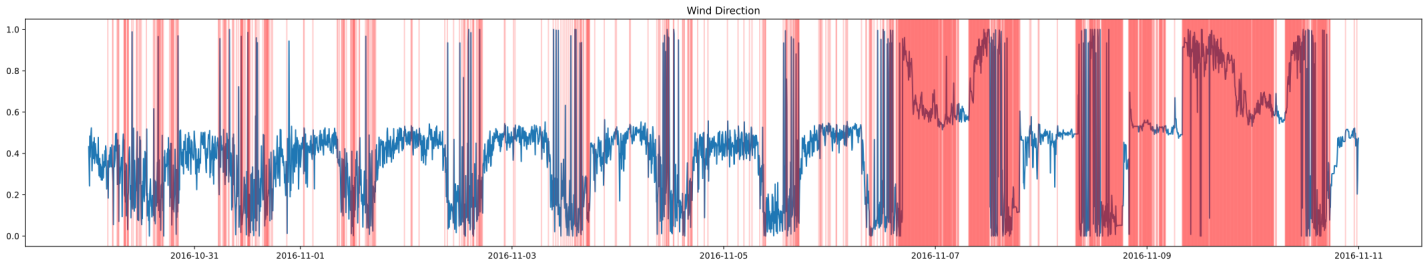


(c) Pressure

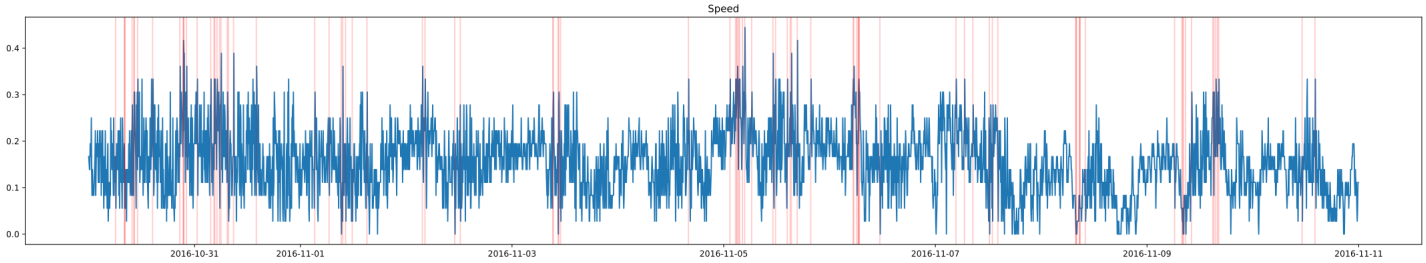
Figure 4.13: Detected Anomalies for Multivariate with 1-head Attention



(d) Humidity

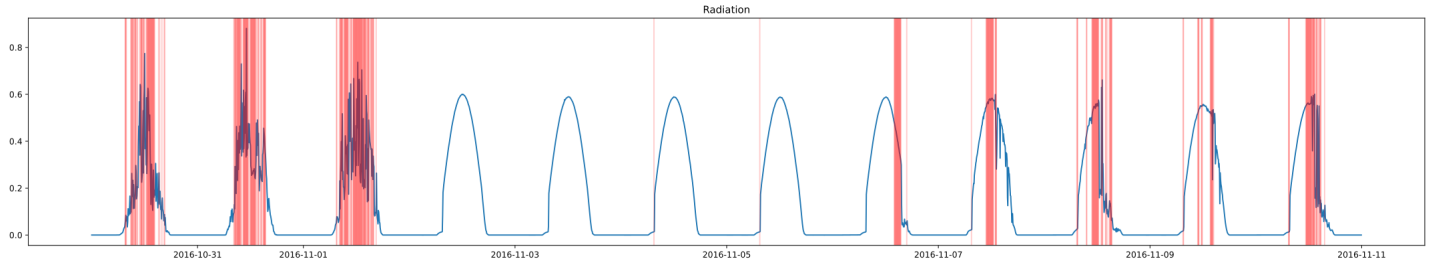


(e) Wind Direction

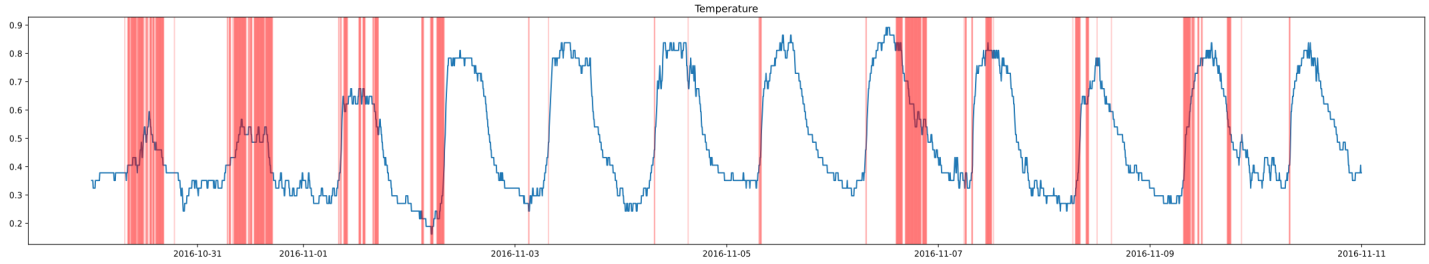


(f) Speed

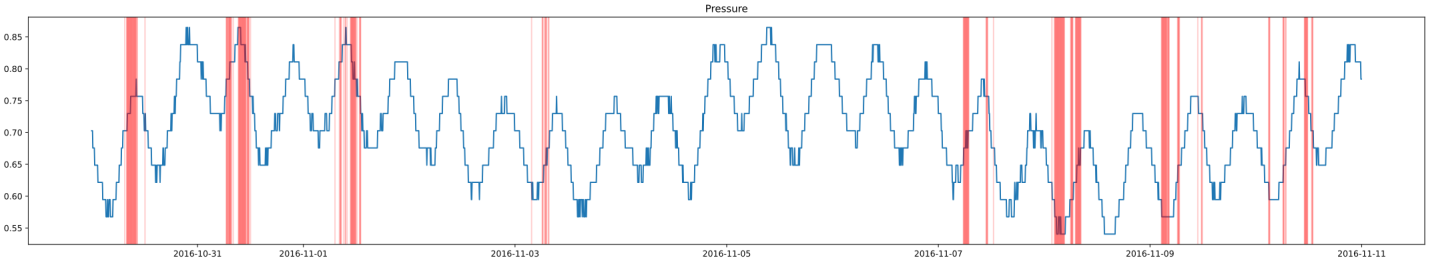
Figure 4.13: Detected Anomalies for Multivariate with 1-head Attention



(a) Radiation

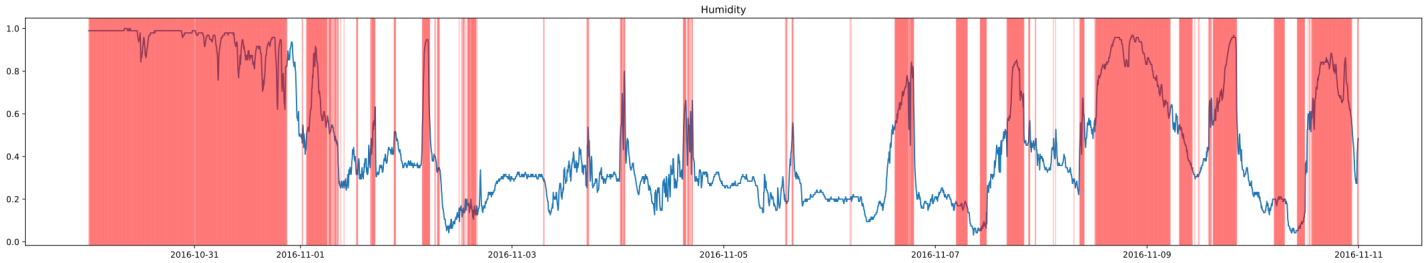


(b) Temperature

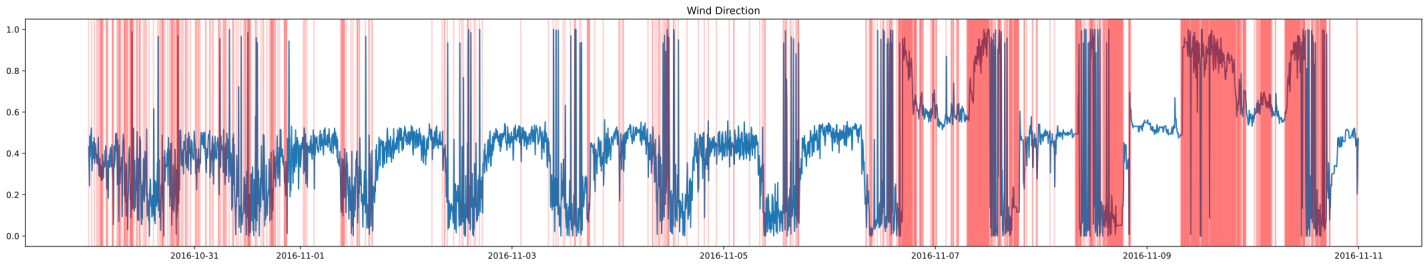


(c) Pressure

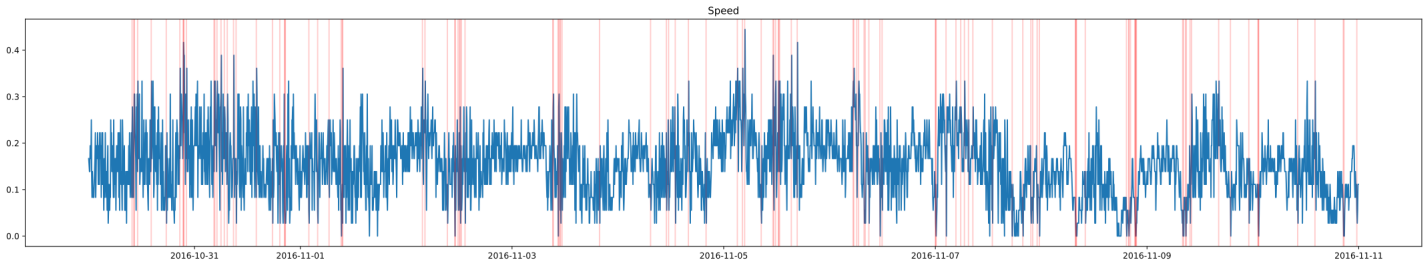
Figure 4.14: Detected Anomalies for Multivariate with 4-head Attention



(d) Humidity

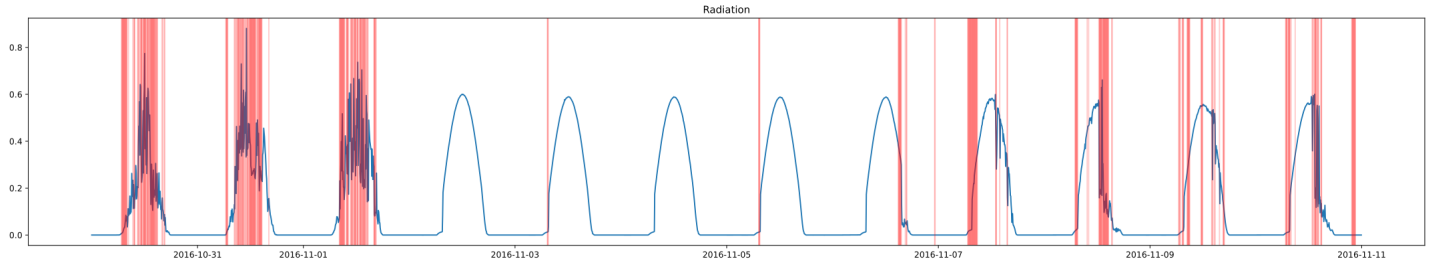


(e) Wind Direction

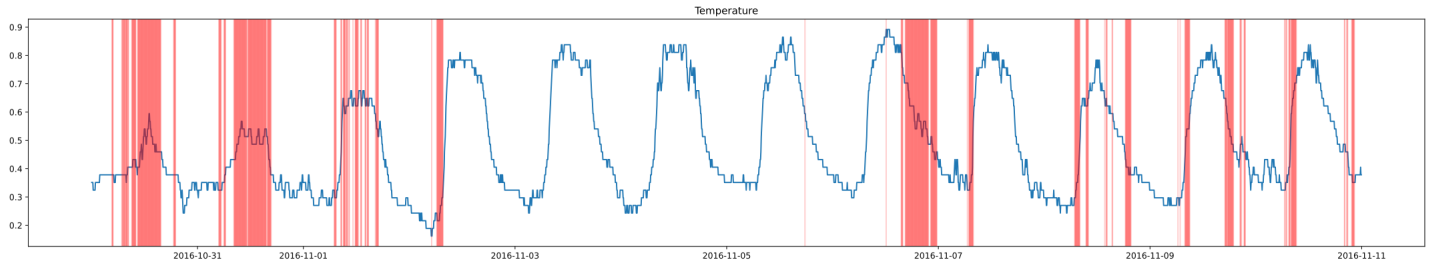


(f) Speed

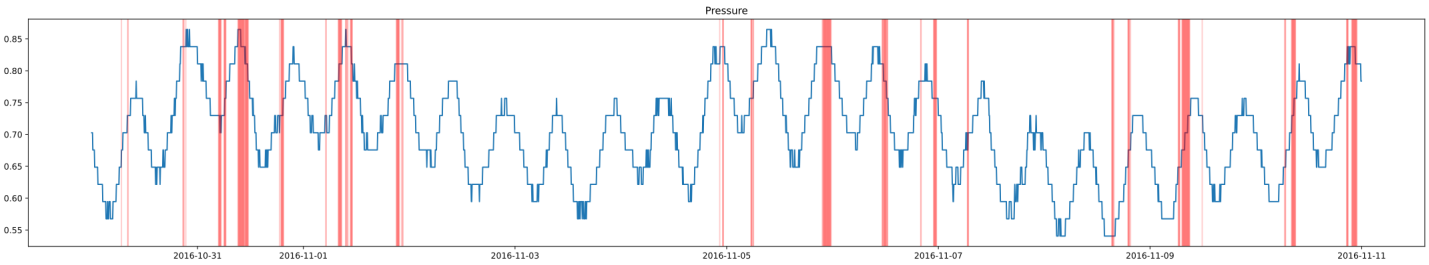
Figure 4.14: Detected Anomalies for Multivariate with 4-head Attention



(a) Radiation

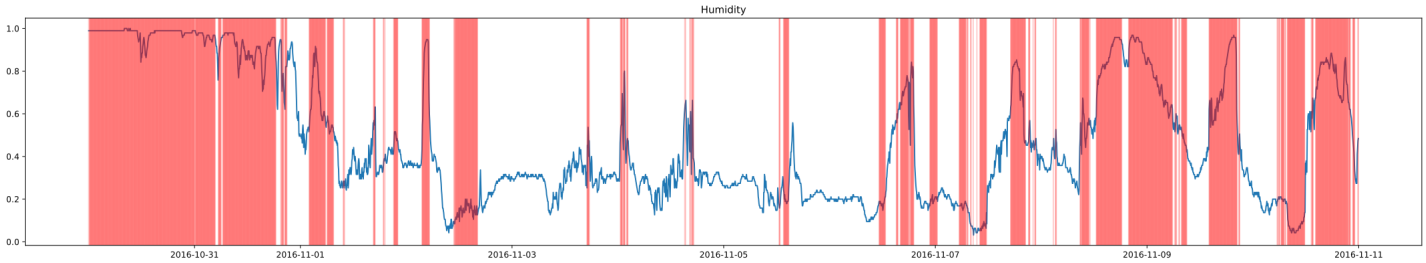


(b) Temperature

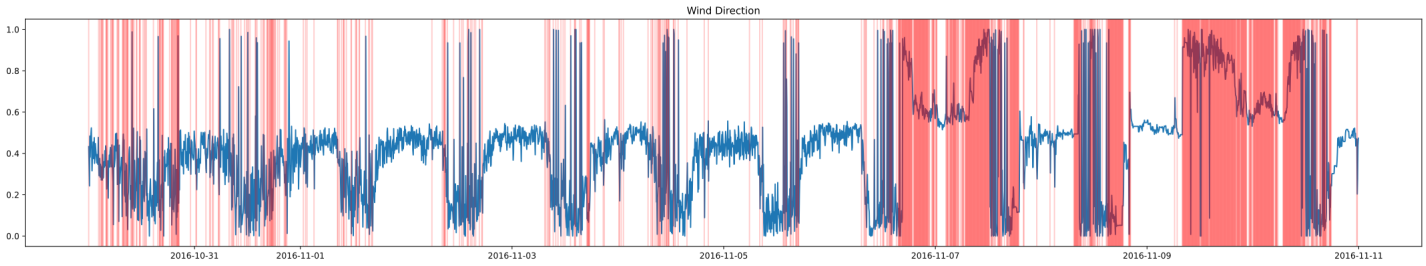


(c) Pressure

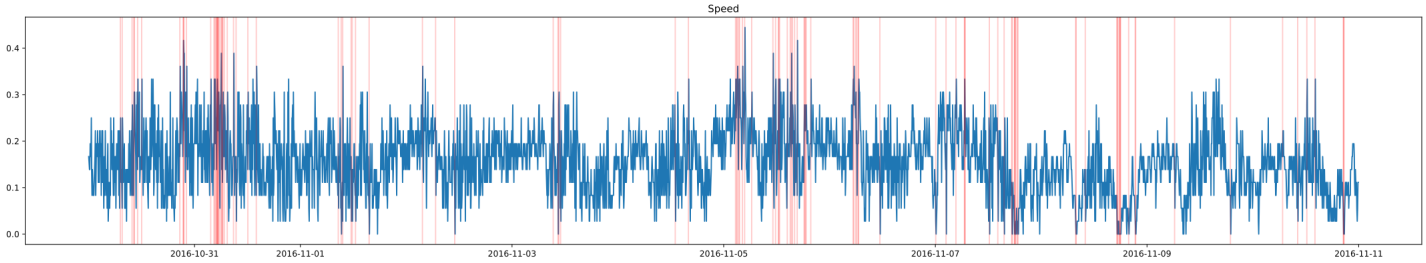
Figure 4.15: Detected Anomalies for Multivariate with 8-head Attention



(d) Humidity



(e) Wind Direction



(f) Speed

Figure 4.15: Detected Anomalies for Multivariate with 8-head Attention

5

Stream Tuning

5.1 Context

We tackled the problem of HPO in chapter 3, but this concerns finding the best parameters for a static problem dataset. How should we deal with models that work in an online paradigm, where data is changing over time? In order to try solving this problem, we re-frame the MDP in section 3 in the following way. Let Λ be a n-dimensional hyper-parameter space, such that $\Lambda = \Lambda_1 \times \Lambda_2 \times \cdots \times \Lambda_n$ where Λ_i is the hyper-parameter set of the i-th hyper-parameter. Denote the space of all possible models by $\mathcal{M}$, and the model $M \in \mathcal{M}$ equipped with hyper-parameters λ, by M_λ. Let $\mathcal{D}$ be the set of all datasets and $\mathcal{P}$ be a performance measure, that is, a function that measures how well the model performed given a task and a dataset of $\mathcal{D}$. We want to estimate a function $\mathcal{T}\colon \mathcal{D} \times \Lambda \mapsto \mathcal{M}$ such that:

$$\mathcal{T}(D_t, \lambda) = \underset{M_\lambda \in \mathcal{M}}{\arg\max}(\mathcal{P}(M_\lambda, D_t)) \tag{5.1}$$

where D_t is the current data batch. In other words, given a set of hyper-parameters, discrete or continuous, we want to retrieve the hyper-parameter configuration that maximises the performance $\mathcal{P}$ of our model M with the current data batch D_t.

From this formalism, we can provide a configuration for a MDP as follows:

- the data state S_D contains a sequence of data batch statistical information, that is

$$S_D = (D_{S_1}, D_{S_2}, \ldots, D_{S_n})$$

  where each D_{S_i} is the statistical information of data batch D_i;

- the parameter state S_P, which contains the sequence of tested hyper parameters together with its corresponding reward, that is

$$S_P = ((\lambda^1, r_1), (\lambda^2, r_2), \ldots, (\lambda^n, r_n))$$

  where each (λ^i, r_i) corresponds to the tested hyper-parameters for data batch i, and r_i is the respective reward;

- the overall state, S is just the tuple (S_D, S_P);
- the action set, A, represents the configurations we are able to select, in this case its Λ itself;
- π is the transition probabilities we want to estimate, that is, a policy for state transition;
- the reward, r, corresponds to the model's performance for a given hyper-parameter configuration.

5.2 Proposed Framework

The framework is quite simple, as figure 5.1 suggests. Upon receiving a new batch of data, it is processed by the model and we read the model's performance over the data batch. This performance value is posteriously added to a performance window, which stores the last n performance values. Said window is now fed to the anomaly detection module that will run the univariate VAE described in section 4, and then compute the reconstruction probability of the last performance value. If the negative reconstruction probability is greater than 0, we consider this an anomaly and proceed to the parameter estimation phase. Else, we proceed to process the next data batch, if available. In case of an anomaly being detected, we run a policy trained via dynamic training for m iterations, outputting the best performant parameters. Having these parameters, we simply retrain the model.

5.3 Dataset

To our knowledge, there is no dataset suited for this type of task, stream tuning. In face of this situation, we selected two 2D clustering datasets, one for training [1] and one for testing [2], and built a stream dataset as follows:

- First we declare the number of data batches to use and the number of minimum and maximum instances a data batch can hold.
- Secondly, for each batch to build, we pick a random size between the minimum and maximum values. Then we pick said number of instances from the dataset, without replacement to avoid duplicates.

[1] http://cs.joensuu.fi/sipu/datasets/a3.txt
[2] http://cs.joensuu.fi/sipu/datasets/a2.txt

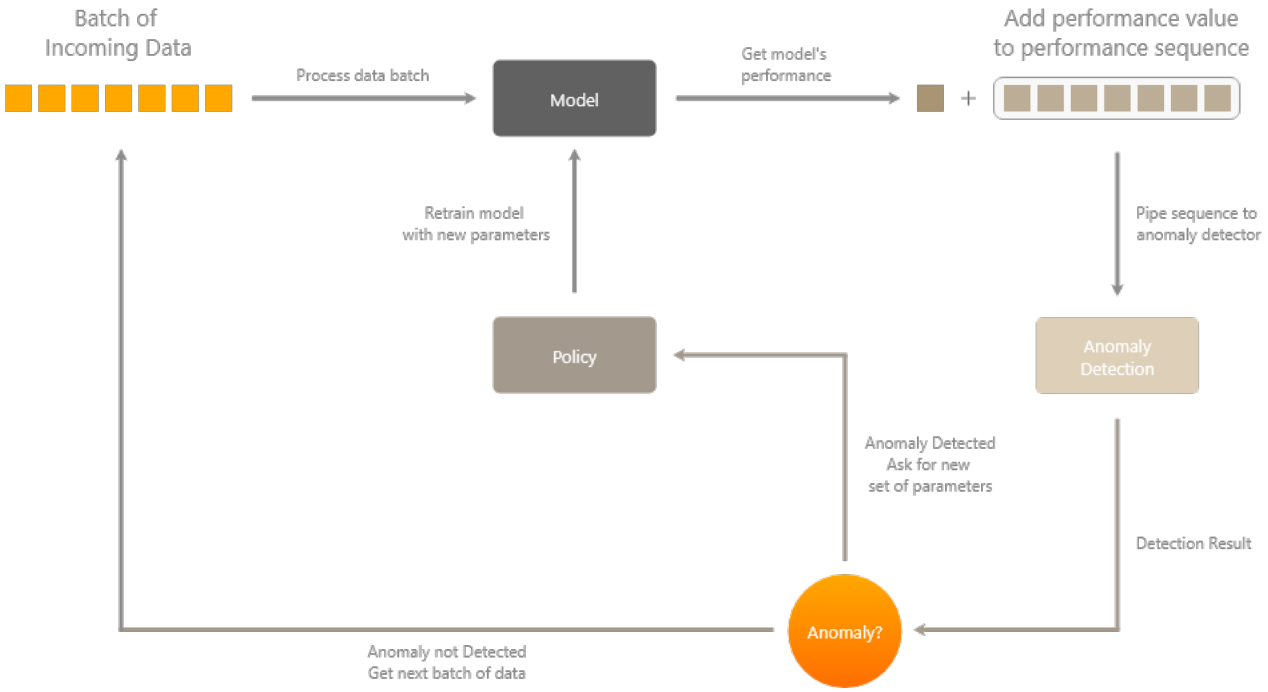


Figure 5.1: Stream Tuning Framework

Before building the stream datasets, the data is normalized so that each 2D vector as length 1.

For the training stream dataset, the number of batches is 50 with a minimum and maximum number of instances 100 and 1000, respectively. As for the test stream dataset, the number of batches is 150, the minimum number of instances is 10 and the maximum is 2000.

5.4 Training

5.4.1 Policy Training

The policy training is similar to the one employed in section 3. The main difference is that here we use dynamic training instead of static training. This means that in each episode iteration we select a different dataset for the policy to optimise, trying to mimic what could happen in an online environment. In what concerns parameters, the transformer and PPO parameters are the same as the ones used back in section 3.

The model to tune is also different. In this case, we are fine tuning an online cluster algorithm, Evolving Cluster Method (ECM) [SK02], which has only one parameter: the distance threshold. For said parameter, we explore the settings in table 5.1. For the performance measure we used the mean Silhouette Coefficient (SC) [Rou87], mainly because it is unsupervised and provides a numerical way to assert the clusters shape.

Hyper-Parameter	Min Value	Max Value	Type
Distance Threshold	0.001	0.01	Continuous

Table 5.1: ECM Hyper-parameters

To finalise, we made the following assumption while training and evaluating the policy: between episode

iterations, we assume that all the previous data was deleted from the cluster so that the cluster is clean to receive new data. Hence the the mean SC concerns only the current batch of data.

5.4.2 Anomaly Detector Training

Regarding the VAE training, we follow the same tactics as the ones described in section 4, furthermore, we employ the same parameters as the one used in the univariate case. Here we also consider a sequence of size 36.

Concerning what a normal sequence is in this context, we consider a normal sequence to be a sequence that is bounded between two desired thresholds. An example of such sequence is visually displayed in figure 5.2.

Since a clustering algorithm is at play and for its performance measure we use the mean SC, we chose the bounds to be 0.5 and 0.1. We now justify this choice of values. This stream tuning problem arose in the context of a cluster based recommendation system. Hence, it is of no interest to us if the clusters are too sparse, implying that each cluster's objects would have little information in common with each other, neither too dense, as this would make the cluster's objects less diverse. With this is mind, we chose the lower bound to be 0.1 and the upper bound to equal 0.5.

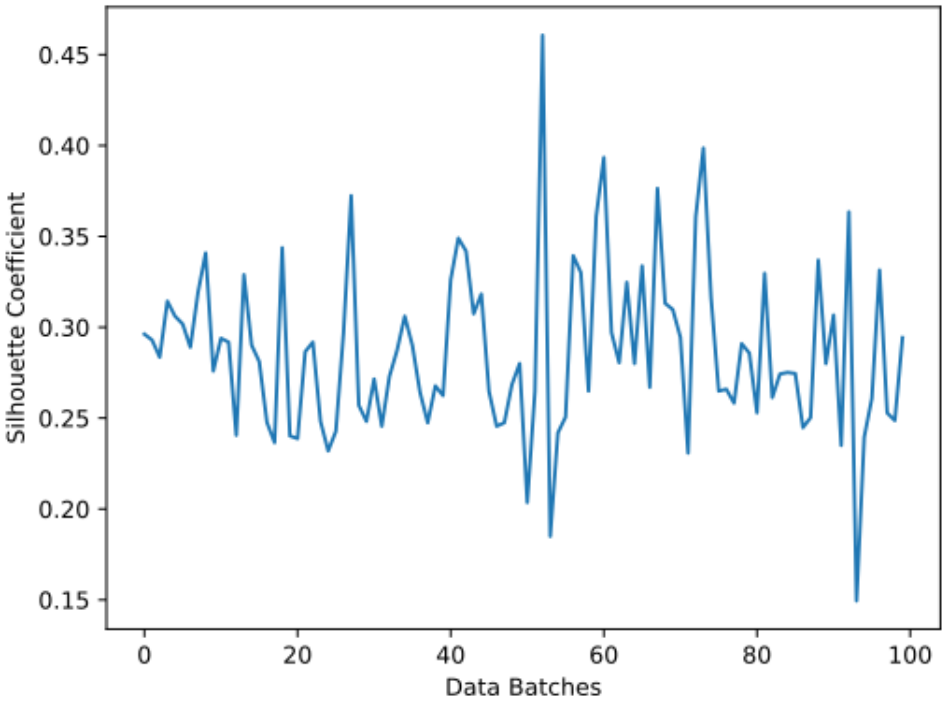


Figure 5.2: Normal Sequence

5.5 Evaluation

In order to evaluate the proposed procedure, we compare the results to the baseline using the rolling average rank. To compute this measure, we first obtain the rank for each data batch and then apply a rolling mean over the ranks. Here, we consider a rolling window of size 10.

The baseline consists of ECM with a fixed distance threshold of 0.001.

5.6 Discussion

5.6.1 Data Statistics Attention Analysis

Back in section 3, we noted that the parameters attention plot was changing over iterations. In a dynamical environment, as the one we are trying to reproduce, the data statistics attention plot also changes its focus over time. Below, in figures 5.3 and 5.4, we show the data statistics attention plots at 100 data batches and at 300 data batches.

At 100 iterations we conclude that attention mechanisms are paying the highest attention to the number of instances, followed by the logarithm of the number of instances, inverse data dimension, kurtosis min, kurtosis mean and skew min. At 300 iterations some of these features are still taken into consideration, yet the kurtosis min, skew min and inverse data dimension are discarded in favor of the data dimension, logarithm of the data dimension, kurtosis max and kurtosis std. This serves as way to show that the model is adapting to the changes in data, impacting the parameter choices.

5.6.2 Stream Analysis

It remains to discuss how the framework behaves on test time. Looking at figure 5.5, we can already depict some benefits of using this method. Around 120 batches of data, the baseline drops to a mean SC of -1.0, the anomaly detector picks this anomaly and tells the policy to find new parameters. The parameters found are able to put us back at a mean SC of 0.4. The remaining detected anomalies consist of drops on the mean SC, as we can conclude by comparing figures 5.5a and 5.5b.

Moreover, it also seems that the trained policy is besting the baseline in the majority of the batches. We can get a more in depth analysis by visualizing the rolling average rank plot, figure 5.6.

Until roughly 100 data batches, the ranks are equal, given that no anomalies were detected, the two scores match. As the first anomaly is detected, the policy starts dominating the baseline. This behaviour keeps going at around 110 batches, switching the ranks. At last, the policy retakes the lead.

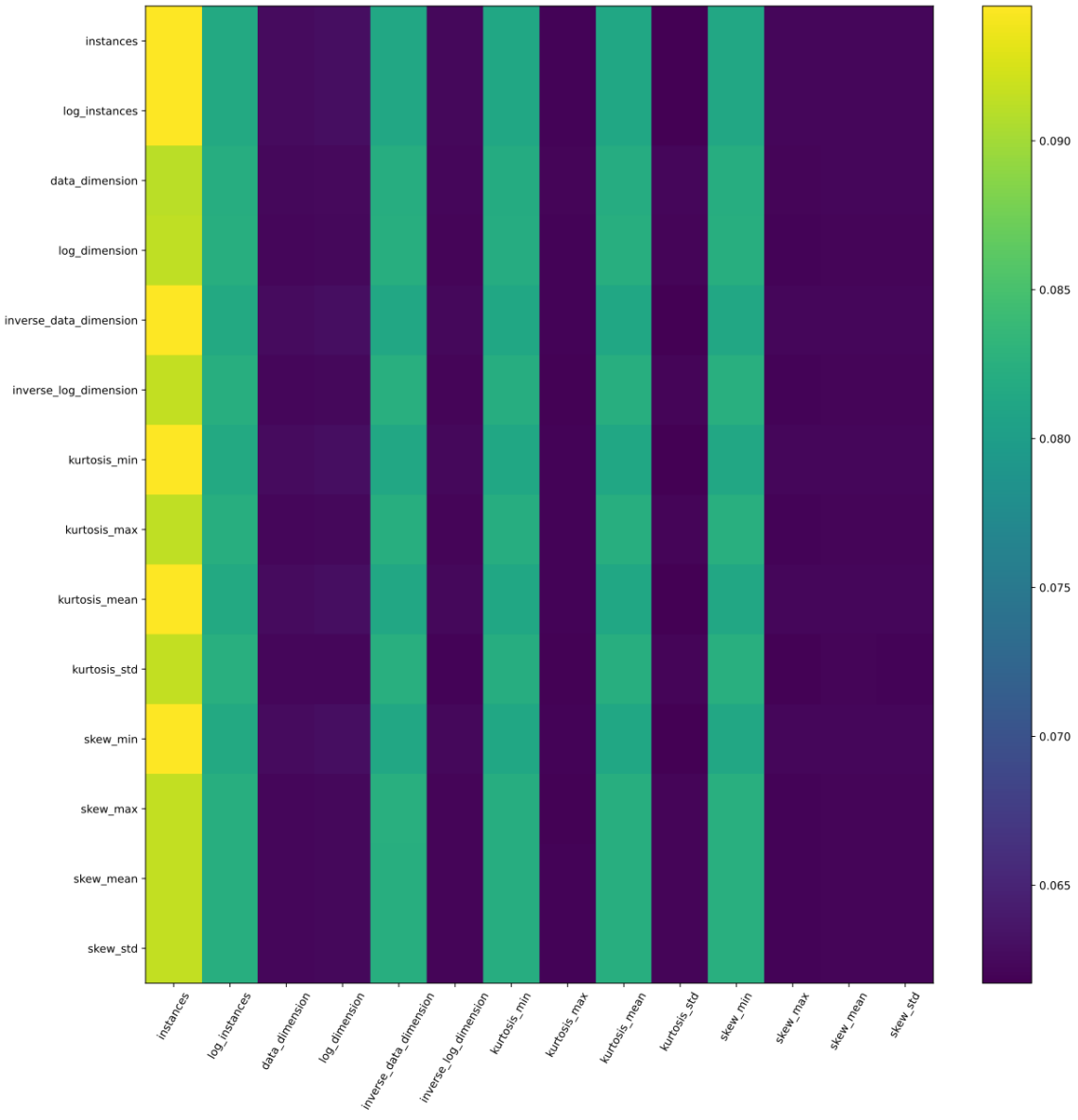


Figure 5.3: Transformer Encoder Graph Attention Plot at 100 batches

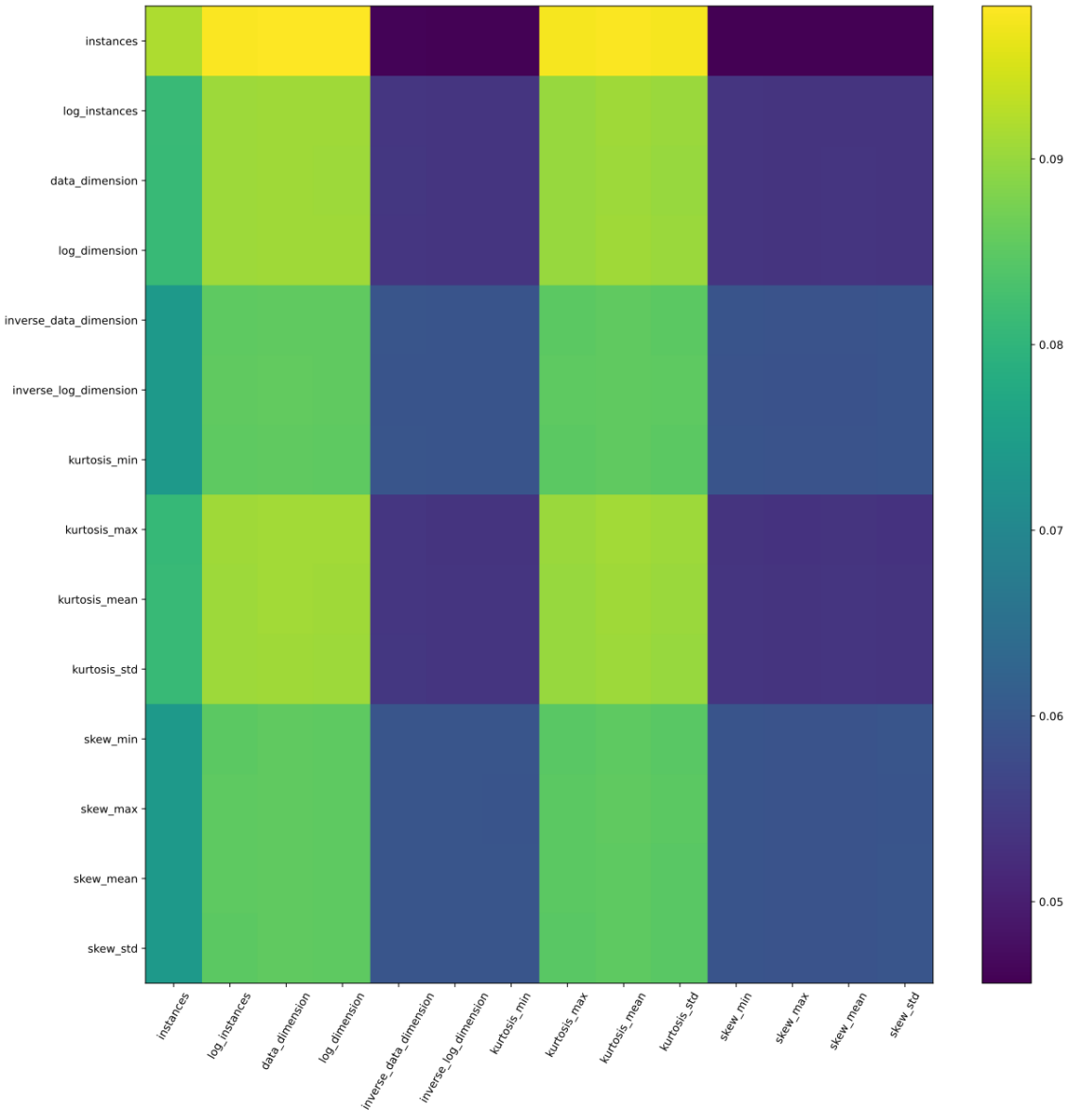
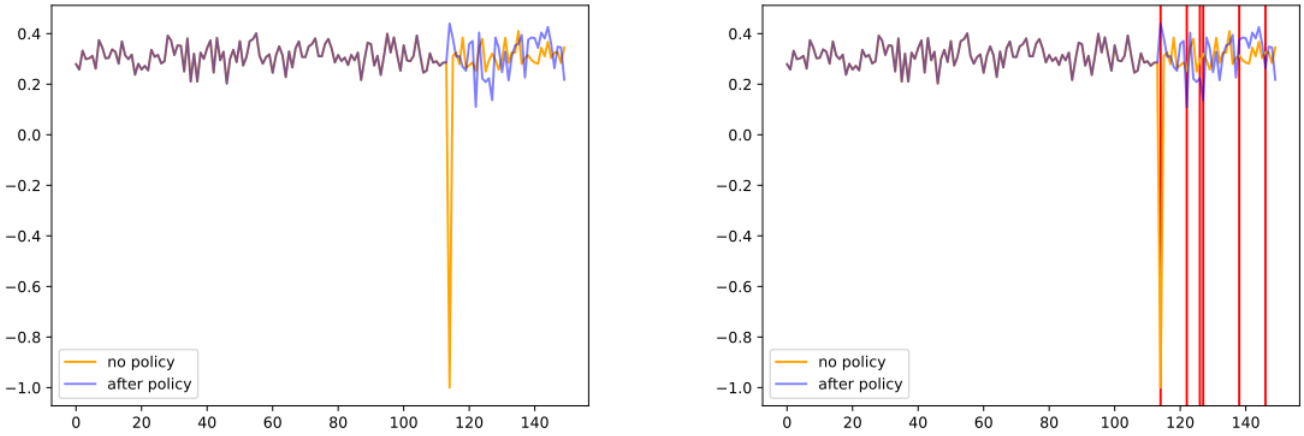


Figure 5.4: Transformer Encoder Graph Attention Plot at 300 batches



(a) ECM SS No Anomalies Plotted

(b) ECM SS With Anomalies Plotted

Figure 5.5: ECM scores over data batches with and without policy

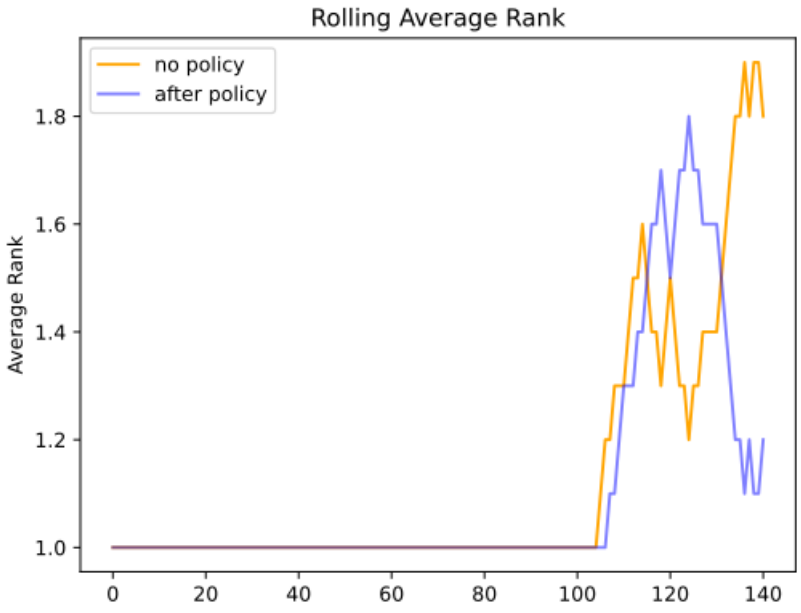


Figure 5.6: Rolling Average Rank with a window of size 10

6

Conclusion

We began this work with the objective of proposing a framework capable of tuning an algorithm in an online context. Thus, we posed to major questions: how to fine tune such model and when to fine tune it. Regarding the first question, we pursued the path of HPO trough a RL approach. As for the second question, anomaly detection methods seemed the most suitable for the task.

At the core of our work is the transformer architecture, which has proven to be useful in the contexts applied. Back in section 3 we explored the transformer as function capable to map data statistics and previous parameter information to the next parameters to test. For that, we modified the standard transformer architecture to include graph attention mechanisms. These mechanisms treat information as a graph, which we use to model the compatibility of the parameters with each other. This strategy is also extended to the data statistics, values that describe a dataset. However, the transformer by itself is of no use if we cannot provide a setting where the model can learn. With this in mind, we took advantage of RL techniques, namely the PPO algorithm, and modelled the HPO problem as a MDP. With this, we have been successful in achieving state of the art results when compared to well known solutions like Hyperband and BOHB. Moreover, given we are using a transformer, it would be a waste if it was only useful to tune datasets it has seen. We showed as well that this is not case, for the transformer architecture is able to

generalize to completely unseen datasets, aiding the hyper-parameters decision process in datasets of the same type and algorithm.

Section 4 extends the previously anomaly detection methods, namely the VAE, to take more advantage of the properties of the transformer architecture. One of the major points was the use of multi-head attention mechanisms and the use of full sequence encoding information to produce the latent space representations. We noted that a different number of attention heads produces different latent-space representations, hence, different anomalies are detected by a reconstruction approach. Thus, the number of attention heads proved to be an early way of selecting the type of anomalies we want to detect. Furthermore, we studied how the anomalies are represented in the attention mechanisms plots. Additionally, using a full transformer architecture, we extended the transformer applied to the univariate case, to a more general scenario where we might be dealing with multiple variables. With this, we are able to analyze the variable features and sequences in separate, extracting the most information out of it, to further combine both information before computing the corresponding latent representation. This method employs the transformer as enriching a sequence with feature information.

Section 5 combines the methods developed in the previous sections to provide a possible framework to stream tune an algorithm. In this case we are treating each data batch as an individual dataset, with respect to which the underlying model is tuned. The need of a tuning phase follows naturally from the problem, yet running the policy to every received batch of data would waste computational resources. Therefore, we only tune when it's needed, hence the necessity of an anomaly detection method. Furthermore, we showed the framework in action, proving itself to be a starting point in this type of problem.

6.1 Extensions and Improvements

Naturally, the problem of HPO can be extended to a problem of architecture search. Instead of defining which parameters to explore in given domains, we simply define building blocks for an architecture that we would like to explore for solving a given problem. In this case, we are not navigating the hyper-parameter space but the architecture space.

One major improvement of the proposed method would be to not use the buffer matrices responsible for holding the previously tested parameters, corresponding rewards and the data statistics. The filling of these matrices before the actual parameter estimation wastes iterations, which is a downside when comparing to the remaining methods.

Regarding the anomaly detection transformers, the multivariate transformer is an experimental approach that combines feature information with time information. This introduces some noise into the attention plots. Thus, using methods capable of eliminating the noise produced by the multiple variables would greatly benefit the model. Furthermore, the feature encoder consists solely of analysing the time sequence for each feature as a whole, and then derive feature compatibility via graph attention mechanisms. Hence, a finer feature encoder would also produce better results.

6.2 Future Work

Apart from the already noted improvements, during the course of this work, some questions posed themselves. The first question touches the datasets and their attention plots. Can a dataset be uniquely characterized by its attention plot? Furthermore, is there a unique way to describe a dataset? If it is, then similar datasets should share the same parameters when it comes to maximize a model performance over some task. That is, if two datasets are similar, then their hyper-parameter surfaces for some model and

performance measure should be isomorphic in some sense. Hence, the parameters that are a maximum on the first dataset hyper-parameter space, should also be a maximum on the second dataset hyper-parameter space.

The second question regards the number of attention heads. We noted that in the case of using 4 attention heads, they complement themselves, namely in the anomaly detection setting. Is this a consequence of how attention mechanisms work? Do a different number of attention heads interfere with each other or do they always work constructively?

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[AC15] Jinwon An and S. Cho. Variational autoencoder based anomaly detection using reconstruction probability. 2015.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(10):281–305, 2012.

[BBKB11] James Bergstra, R. Bardenet, Balázs Kégl, and Y. Bengio. Algorithms for hyper-parameter optimization. 12 2011.

[BCB16] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[BGNR17] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning, 2017.

[BKK18] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling, 2018.

[BMVP18] Hareesh Bahuleyan, Lili Mou, Olga Vechtomova, and Pascal Poupart. Variational attention for sequence-to-sequence models, 2018.

[BSGL09] Shalabh Bhatnagar, Richard S. Sutton, Mohammad Ghavamzadeh, and Mark Lee. Natural actor–critic algorithms. *Automatica*, 45(11):2471–2482, 2009.

[CG16] Tianqi Chen and Carlos Guestrin. Xgboost. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug 2016.

[FKH18] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale, 2018.

[FZL+21] Le Fang, Tao Zeng, Chaochun Liu, Liefeng Bo, Wen Dong, and Changyou Chen. Transformer-based conditional variational autoencoder for controllable story generation, 2021.

[HDV18] Danijar Hafner, James Davidson, and Vincent Vanhoucke. Tensorflow agents: Efficient batched reinforcement learning in tensorflow, 2018.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[JGST19] Hadi S. Jomaa, Josif Grabocka, and Lars Schmidt-Thieme. Hyp-rl : Hyperparameter optimization by reinforcement learning, 2019.

[JSTG21] Hadi S. Jomaa, Lars Schmidt-Thieme, and Josif Grabocka. Dataset2vec: learning dataset meta-features. *Data Mining and Knowledge Discovery*, 35(3):964–985, May 2021.

[KGN+20] Chepuri Shri Krishna, Ashish Gupta, Swarnim Narayan, Himanshu Rai, and Diksha Manchanda. Hyperparameter optimization with reinforce and transformers, 2020.

[KW14] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2014.

[KW19] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 12(4):307–392, 2019.

[LJD+18] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization, 2018.

[LL19] Petro Liashchynskyi and Pavlo Liashchynskyi. Grid search, random search, genetic algorithm: A big comparison for nas, 2019.

[Per18] João Pereira. Unsupervised anomaly detection in time series data using deep learning, 11 2018.

[PSCH21] Guansong Pang, Chunhua Shen, Longbing Cao, and Anton Van Den Hengel. Deep learning for anomaly detection. *ACM Computing Surveys*, 54(2):1–38, Apr 2021.

[RMS+17] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers, 2017.

[Rou87] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[Rud17] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[SK02] Qun Song and Nikola Kasabov. Ecm - a novel on-line, evolving clustering method and its applications. 08 2002.

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

[VCC+18] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[WCL20] Jia Wu, SenPeng Chen, and XiYuan Liu. Efficient hyperparameter optimization through model-based reinforcement learning. *Neurocomputing*, 409:381–393, 2020.

[XYB+20] Xueli Xiao, Ming Yan, Sunitha Basodi, Chunyan Ji, and Yi Pan. Efficient hyperparameter optimization in deep learning using a variable length genetic algorithm, 2020.

[YRK+15] Steven R. Young, Derek C. Rose, Thomas P. Karnowski, Seung-Hwan Lim, and Robert M. Patton. Optimizing deep learning hyper-parameters through an evolutionary algorithm. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, MLHPC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[ZWD+20] Hang Zhao, Yujing Wang, Juanyong Duan, Congrui Huang, Defu Cao, Yunhai Tong, Bixiong Xu, Jing Bai, Jie Tong, and Qi Zhang. Multivariate time-series anomaly detection via graph attention network, 2020.

UNIVERSIDADE DE ÉVORA
INSTITUTO DE INVESTIGAÇÃO
E FORMAÇÃO AVANÇADA

Contactos:
Universidade de Évora
Instituto de Investigação e Formação Avançada — IIFA
Palácio do Vimioso | Largo Marquês de Marialva, Apart. 94
7002 - 554 Évora | Portugal
Tel: (+351) 266 706 581
Fax: (+351) 266 744 677
email: iifa@uevora.pt