Universiteit Leiden

Master Computer Science

Multivariate Time Series Classification In Radial Drilling Applications

Name:	W.J. Stokman
Student ID:	s2025418
Date:	10/08/2023
Specialisation:	Artificial Intelligence
1st supervisor:	Dr. E.M. Bakker
2nd supervisor:	Prof. dr. M.S. Lew

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

Abstract

The application of Radial Drilling is an emerging field in the geothermal, oil- and gas industry. By creating radial extensions from existing wells into existing reservoirs, the extraction of resources can be made easier - improving the yields of these wells at a relatively low cost. We aim to automatically classify the main steps in the Radial-Drilling process based on the previous outputs of 4 of the most important sensors. We have designed and implemented an open source machine learning pipeline for these kinds of multivariate time-series classification tasks. This pipeline consists of MVTS-analyzer, a platform for visualizing annotating and analyzing multivariate time-series data and Configurun, an app installable as a python package that can be used to (remotely) create, manage and run python configurations - on top of which we have implemented our machine-learning framework, making all (novel) classification methods available in an intuitive UI.

MVTS-analyzer is used to annotate the raw Radial-Drilling data. Configurun and the implemented training-framework are then used to train and test the performance of 7 models on this novel classification-task and on 10 existing time-series classification tasks.

By combining two state-of-the-art classification methods (Rocket and XGBoost) we create a new classifier that shows robust results on the tested datasets. Based on a state-of-the-art Time Series Transformer architecture (TST) we design and implement our own convolutional-transformer classification architecture (CTST). We expand upon this idea by applying multiple convolutional layers (MCTST), allowing for a higher degree of feature extraction before self-attention is applied. We train and test these and several state-of-the-art models and compare our found results to the author-reported performance. From our results, the average accuracy of TST is still slightly lower than that of the state-of-the-art methods. CTST shows a slight overall increase in accuracy over TST, while MCTST shows some improvement on select datasets, but does not improve the accuracy on average. On the novel Radial-Drilling classification dataset, we achieved a maximum prediction accuracy of 82% using our proposed method of combining Rocket and XGBoost.

1 Introduction

Radial Drilling - in the geothermal oil- and gas industry - is a process in which radial-extensions are drilled from existing resource wells. It is a field that focuses on improving the extraction efficiency of wells. Radial-Drilling operations are most often performed on wells with relatively low performance in an attempt to increase yields by improving the reach to existing reservoirs by branching off from the well at predetermined depths. Over the last decades, the Radial-Drilling process has seen considerable success, improving well efficiency in various places around the world such as in Egypt [30] and Kuwait [22].

During the drilling-process, the operator is presented with a multitude of continuous sensor outputs. Based on these outputs, the operator decides how to proceed. At the same time, a log of the reached (sub)-steps is manually kept. This log is later used, in combination with the sensor data, to create a report on the success of the drilling-operation.

Proper post-operation analysis of the sensor data is a valuable resource, as it is essential to demonstrate the success of the operation to the client. Analysis of these results also allows for further optimizations of future Radial-Drilling operations by identifying the most and least successful approaches.

The segmentation of the data into the various sub-process makes this task significantly more manageable and allows for more accurate predictions on the actual success of a drilling-operation. Coupling the logs made by the operator to the recorded sensor data after the fact, however, is a time-consuming task. Furthermore, knowing when certain milestones in the (sub)processes have been reached requires the operator to keep constant track of the sensor outputs while at the same time, the rig still needs to be controlled. All the while the logs have to be kept as accurately as possible.

Automatic segmentation of the drilling-data into the various sub-processes, therefore, has the potential to significantly improve the efficiency of the Radial-Drilling process. It furthermore enables the use of previously recorded (unlabeled) data in sub-process analysis tasks without the need for meticulous manual labeling of weeks of sensor outputs.

Automatic classification of the main processes during run-time additionally provides a basis for live-process monitoring, which could aid the operator in more accurately determining the current underground state to allow for better decision-making.

Most publicly available research, however, is done on the individual mechanical components of the Radial-Drilling process, such as the jetting nozzle [20]. Some studies have tried to apply machine learning to the geological well-parameters, to predict the success of a location in advance [24]. Classification of the main processes of the Radial-Drilling operation has, as far as we know, not yet been done.

Multivariate time-series classification is a difficult task, for which not many open-source tools are available. This makes the process of annotating, visualizing and analyzing multivariate time-series data much less accessible, especially for non-experts. After annotation, determining what classifier is best suited for the task at hand is another challenge, as is managing the various hyperparameters of each classifier and keeping track of the results. The end-to-process of creating solutions to these problems is therefore time-consuming - indicating that a general-use pipeline that makes this process easier, would be a valuable resource in all fields that deal with multivariate time-series data.

1.1 Research Questions

From the abovementioned, we formulate the following research questions:

1. Can we accurately predict the main process steps of the Radial-Drilling process based on the previous sensor outputs of the drilling rig?
2. Can we create an accessible end-to-end machine learning pipeline for multivariate time-series classification tasks?

2 Background and Related Work

2.1 Related work

2.1.1 Time Series Classification

Historically, time series data has been difficult to work with. Classification methods on this type of data are often more complex than their univariate counterparts and popular methods have not always been able to make efficient use of parallelization (e.g. using GPU's). While the field of univariate time-series prediction has seen considerable advances, the creation of competitive multivariate classification-specific methods has been shown, probably due to this added complexity, to lag behind [8].

A problem with simple higher-dimensional time series data, especially when doing similarity search, is the exponential increase in the search space as the amount of dimensions increases. Schäfer et al. [32] propose SFA which is able to handle up to a factor 5-10 more indexed dimensions than previous approaches.

Chen and Guestrin have created XGBoost [6], a tree-based algorithm with an emphasis on scalability. XGBoost allowed up to 10 times faster runtimes on single machines compared to other popular solutions at the time. This makes this algorithm more suitable for decision problems operating on very large datasets. XGBoost has been used successfully in, for example, Kaggle competitions. In 2015, out of 29 winning solutions, 17 solutions used XGBoost. The improvements of this algorithm over normal gradient boosting methods are explained further in Section 3.1.1.

Bagnall et al. [2] combine the idea of previously shown performance improvements due to data-transformations into alternative spaces where discriminatory features are more easily detected, with the principle that, in many situations, improved understanding of the data can be achieved through ensemble-schemes, combining multiple classifiers into one. They create the Collective of Transformation-Based Ensembles (COTE) using classification methods in the time, frequency, change and shapelet transformation domains. They show that the ensemble outperforms any of its components and any other previously published time-series classification method.

Lines, Taylor, and Bagnall [28] attempt to apply convolutional neural networks to time-series classification problems. They also propose the Hierarchical Vote Collective of Transformation-based Ensembles (HIVE-COTE), as an improvement upon the aforementioned COTE ensemble. The authors note that Flat-COTE (their most effective ensembling strategy) performs significantly more accurate than 2 implemented neural-network based approaches.

Tang, Liu, and Long [36] [37] propose Dual Prototypical Shapelet Networks (DPSN) for few-show time series classification. Their method both trains a neural network-based model and interprets the model using representative shapelets - the discriminating shapelet that appears in all instances of a certain class. The authors test their method on 18 few-shot TSC datasets and compare it to several baseline methods. They observe their method to be especially robust compared to the baselines when the amount of training data is severely limited.

Dempster, Petitjean, and Webb [11], considered the recent success of convolutional neural networks for time-series classification tasks. They show that random convolutional kernels can be leveraged to achieve state-of-the art performance on large datasets, at the cost of only a relatively small amount of computational power. Rocket is explained further in Section 3.2.

Although neural-network based approaches are now the dominantly used methods in both computer-vision and natural language processing tasks, the same cannot be said for the field of time-series classification, where methods such as XGBoost and Rocket, in many cases, represent the state-of-the-art. Recently, however, neural-network based approaches have started to bridge this gap in performance. Mainly because of the demonstrated dominance in other sequence-to-sequence domains, the neural network based approaches represent an interesting alternative to the more traditional methods, one that shows a lot of potential.

Neural-network based approaches to time-sequence tasks initially relied mainly on recurrence, for example with LSTM-units [21]. The canonical LSTM-model, however, often struggles with learning long-term dependencies [4]. Over the years, several improvements were proposed to remedy this problem. Gers, Schmidhuber, and Cummins [19] proposed a forget-gate that can be used by the LSTM-layer to reset itself. Later on Cho et al. [7] proposed gated recurrent units, which can be seen as a simplification of the canonical LSTM-unit. GRU's has been shown to perform similarly to LSTM-units in some tasks [9]. Shen et al. [33] use GRU's for financial sequence prediction tasks due to their ability to better learn long-term dependencies.

Fawaz et al. [14] introduce *Inceptiontime*. This architecture, based on the InceptionV3 image recognition model, focuses on scalability and is able to achieve similar results to HIVE-COTE on the UCR univariate time-series classification-task, while cutting down on calculation times by two orders of magnitude.

Kieu et al. [23] propose a sparsely connected Recurrent Neural network auto-encoder ensemble in the field of outlier detection. The authors experiment with training unsupervised auto-encoder ensemble frameworks in both a joint and independent manner. Auto-encoders represent the input data in a compact, hidden representation from which the decoder can only reconstruct representative features. By comparing a reconstructed output - as generated by the decoder - to the original input, the authors are able to outperform all tested state-of-the-art methods on their outlier detection task.

In *Attention Is All You Need* [39], the authors introduce a novel way of sequence transduction. They propose the Transformer architecture: an architecture that leverages attention modules, allowing the model to attend to information at different positions in the input sequence without the need for recurrence (also see Section 3.3).

The creators of the language representation model BERT ([12]), which builds on this transformer architecture, gain considerable performance improvements in the field of Natural Language Processing (NLP). Deviating from earlier methods, they utilize both left and right context when processing text, instead of only attending to tokens previously encountered as many previous state-of-the-art methods did. The authors improve fine-tuning methods on their NLP model by using a masked language model pre-training objective, which aims to predict masked words in sentences. Additionally, a sentence prediction task is used to pre-train text-pair representations. The authors report that the leveraged transformer units also allowed for much better transfer learning performance than achieved before due to previous limitations in transfer learning capabilities introduced by the traditional LSTM-units.

Zerveas et al. [41] introduce *A transformer-based Framework for Multivariate Time Series Representation Learning* in which they directly apply transformers to time series data. They propose and implement a general-use transformer architecture and make it available as a combined framework. They have tested their model on several multivariate time series regression and classification tasks and reported it to have outperformed all the current state-of-the-art approaches. The transformer units allow for unsupervised pre-training on unlabeled datasets using sequence masking and a data imputation task. The authors note that the networks, consisting of "at most hundreds of thousands of parameters, can be efficiently trained: a commodity GPU allows them to be trained approximately as fast as lean non-deep learning based approaches". The findings and proposed architecture serve as the basis for many of the experiments done in this paper, this is explained further in Section 3.3.4.

The use of convolutional layers in time series classification tasks, with or without the use of self-attention mechanisms, has shown promise in a number of studies. Causal convolutions have been shown to be effective in creating embeddings[16] for time series data, allowing for an easier interpretation of long sequences. Using convolutions allows the embeddings to be created in an unsupervised manner, while avoiding the need for recurrence - thereby being more suited for longer time series data.

Cirstea et al. [10] use a combination of recurrence with convolutional neural networks. They experiment both with employing convolutions on each individual time series while applying the

RNN on top, and adding auto-encoders to the convolutional neural networks. The proposed methods are shown to be effective compared to their baselines.

Elbayad, Besacier, and Verbeek [13] apply 2D convolutional neural networks to sequence-to-sequence prediction tasks and introduce an approach that relies on convolutional layers to re-encode source tokens based on the output sequence produced so far. They report competitive results compared to state-of-the-arts methods, while being conceptually simpler and having fewer parameters.

Zhang et al. [42] introduce an attentional prototype network which uses a random group permutation method, combined with multi-layer convolutional networks to extract low-dimensional features from multivariate time-series data.

Li et al. [27] observe a limitation of using Transformers in time-series forecasting tasks due to the fact that similarities between queries and keys are normally calculated based on their point-wise values. Without taking into account the local context of the query, key and value in the transformer, the model might often be unable to distinguish whether an observed value is an outlier, or part of an existing (but slightly changed) pattern in the data. In an attempt to remedy this, they propose convolutional self-attention, which makes use of convolutional layers to generate the Query and Key pair in an attempt to preserve this context. Zhou et al. [43] use a similar idea for their Informer architecture, which is centered around improving the performance on Long Sequence time-series forecasting. They use convolutional layers in the encoder to create the inputs to the attention-blocks.

Shen and Wang similarly use convolutional layers in combination with transformer-modules. They propose three tightly coupled convolutional transformer (TCCT) architectures and an architecture in which these novel TCCT modules are combined with a transformer model.

2.1.2 Time-Series annotation

When dealing with time-series classification problems, one of the main difficulties is the acquisition of quality training and testing data. In particular, the annotation of time-series data is a time-consuming process for which the right tools are important to allow for efficient labeling while maintaining a high level of accuracy.

Fedjajevs et al. [15] implemented a platform for analysis and labeling of medical times series (PALMS). Their python-based user-interface allows a medical professional to annotate time series with fiducials (points of interest, R-peaks of an ECG, for example), events with an arbitrary duration (for example: arrhythmic episodes) and signal quality (data parts corrupted by motion artifacts). PALMS was mainly designed with medical applications in mind.

Tools such as `SUMsarizer`[1], and its evolved online-version `TRAINSET` [2] focus on the annotation of time series data, particularly for Internet of Things-applications. The creators claim that most users use the tool to detect cooking events from temperature sensors called Stove Use Monitoring Systems (SUMS).

Other, more commercially oriented tools such as Visplore [3] exist, but they are not open-source and restrict the data size and/or feature count when labeling on a non-professional plan.

Tkachenko et al. [38] have created *Labelstudio: The most flexible data labeling platform to fine-tune LLMs, prepare training data or validate AI models.* This open-source software platform built using python, implements labeling-tools in a web-UI environment for a variety of tasks such as image classification, object detection, semantic segmentation, but also (multivariate) sequence tasks. The tool is open-source and is provided under the Apache-2.0 license.

[1] https://github.com/geocene/sumsarizer
[2] https://trainset.geocene.com/
[3] https://visplore.com/

2.2 Radial-Drilling - Technical Background

The Radial-Drilling process can, in general, be divided into several steps:

1. Inserting a guide - the **shoe** or **deflector** (see Figure 1) - into the existing well at target depth
2. Inserting a hose with a drill-bit into an existing well
3. Drilling into the existing well-casing at target depth (milling)
4. Pulling out the drill-bit
5. Inserting a jet-nozzle into the existing well
6. Jetting a radial hole from the casing into the surrounding soil (jetting)
7. (Optionally) some form of stimulation of the newly created link to an existing reservoir
8. Pulling the jetting nozzle out of the well

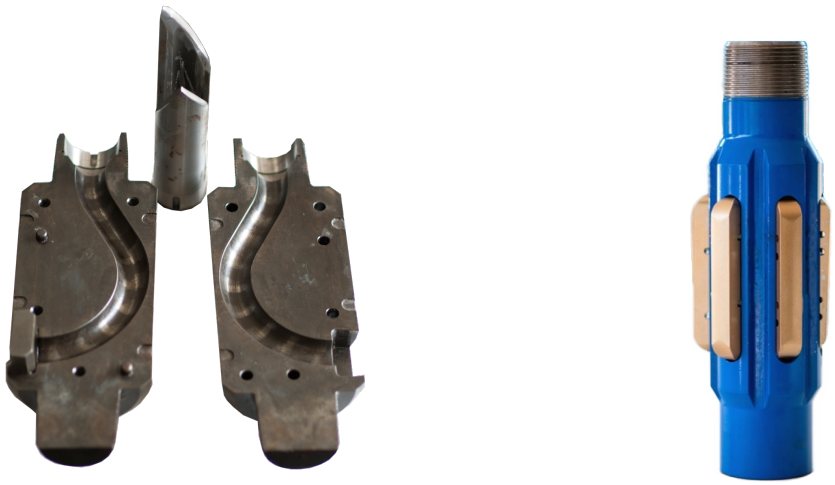

Figure 1: A cross-section and side-view of the shoe. The shoe is (in the displayed orientation) inserted into the existing well and serves as a guide to drill the hole in the existing casing at the desired location and, afterwards, for the jetting-nozzle to enter the reservoir[4].

The operations by Radial-Drilling Europe B.V. are performed by a mobile unit consisting of a trailer-mounted drilling rig. It carries a spool with high-pressure hose, on which a drill-bit or jetting nozzle can be mounted depending on the task at hand. The unit is capable of creating radial wells up to a depth of several kilometers.

Steps 2 to 4 are only performed, if the well in question has a casing, a surrounding tube that prevents well-collapse and/or contamination of the well. The casing, in general, consists of a metal tube surrounded by cement.

The radial-drilling process can be repeated at varying depths for the same well, in whatever direction is deemed fit. Using the inserted guide - the shoe - specific targets can be aimed for: in particular existing reservoirs that have previously been mapped.

Various sensor-outputs are made available to the operator during the drilling operation, based on which the operator can make decisions on how to proceed. Especially around target-depth, the sensor outputs are essential for the operator to determine whether the shoe has been reached, whether the drill-bit or jetting-nozzle has been inserted, and whether the shoe has been passed.

[4]Images of the shoe were provided by Radial-Drilling Europe B.V.

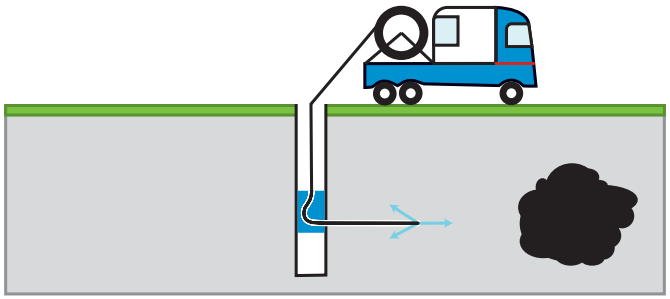

Figure 2: A schematic overview of the jetting step in the radial-drilling process. Existing target-reservoirs in the soil are drilled to using a jetting-nozzle, which is guided into the reservoir through the previously milled casing-hole at target depth by using the shoe (in blue). The jetting nozzle propels itself forward using back-facing jets of water, while simultaneously grinding away the soil in front using the main front-facing jet of water. Reservoirs of oil, water, or gas (depending on the well) can be targeted in order to improve the well-yield.

Both the jetting-nozzle and the drill-bit are hydraulically powered. High-pressure water is pumped through the hose. A hydraulic motor at the end of the hose converts the water-pressure into rotational movement when milling. During jetting, the water is forced through one or more small nozzles at the end of the hose, creating a high pressure jet of water that is used to cut through the existing soil and create the radial well-extension. Optionally, backwards facing jets can be used to propel the nozzle forwards during the jetting operation.

3 Methodology

In this section, we discuss the selected baseline methods used in the experiments in Section 7 - and the motivation behind these choices.

3.1 Gradient Boosting

Friedman [17][18] first introduced gradient boosting in *Greedy Function Approximation: A Gradient Boosting Machine*. Gradient boosting works by creating an ensemble of (individually) weak base learners to create an input/output (x to y) mapping that aims to approximate the output variables based on the input variables: an approximation function $\hat{F}(x)$. Over multiple iterations, it tries to find $\hat{F}$ by using training samples for which input and output are known - y_i, x_i - to minimize the expected value of a loss function $L(y, F(x))$ (for example, the mean squared error loss function).

$$\hat{F} = \arg\min_{F}(E_{y,x}L(y, F(x))) \tag{1}$$

Gradient boosting takes a greedy approach to this problem, by iteratively adding new weak learners (typically decision trees), each one attempting to minimize the loss function further.

The algorithm has been widely used in the field of machine learning. Its popularity can be partly attributed to its performance in Kaggle competitions, where it has been used many times in winning solutions - especially the XGBoost-variant (see Section 3.1.1). Its established robust performance in a variety of time-series tasks is the reason we choose to include it in our experiments.

During our experiments, we used the Scikit-learn [29] implementation of the gradient boosting classifier. This classifier fits `n_classes` regression trees on the negative gradient of the loss function. In the case of a binary classification problem, one single tree is used.

3.1.1 XGBoost

Many implementations of gradient boosting algorithms (such as the previously mentioned Scikit-learn implementation) are unable to efficiently make use of computational resources, which often results in very long training times on larger datasets. Chen and Guestrin [6] have implemented *XGBoost: A Scalable Tree Boosting System*, as its name implies: a scalability-oriented gradient boosting system.

The main changes of XGBoost, compared to the canonical gradient boosting algorithm, in terms of computational efficiency are:

- Column block for Parallel Learning: by distributing data into blocks, and sorting the data within these blocks according to their respective feature values, scanning new split candidates can be done more efficiently. When using approximate algorithms, multiple blocks can be used, allowing the distribution of blocks across machines.
- Cache aware access: using a prefetching algorithm and/or by fine-tuning block-sizes, among other things, makes data-fetching more efficient, by reducing the amount of cache-misses.
- Blocks for Out-of-Core Computation: by enabling the use of data-compression, prefetching and block-sharding when using out-of-core computation, disk-access times are minimized.

In Section 7, we compare the default implementations of XGBoost to the default implementation of the Scikit-Learn gradient boosting classifier, to see, besides the computational efficiency, whether the base-implementation of one or the other offers any advantage on the tested datasets.

3.2 Rocket

Dempster, Petitjean, and Webb [11] propose *Rocket* - **R**and**O**m **C**onvolutional **KE**rnel **T**ransform. This method relies on generating n random convolutional kernels, which are then applied to the target data before a classifier is used. According to the authors, using random convolutional kernels for feature extraction has been shown to be effective in multiple earlier studies, which was their main motivation for further exploring this method.

The Rocket-method distinguishes itself from previous random-kernel-transform based methods in 4 main ways. The first one being that Rocket uses a relatively large amount of kernels. Because computing these kernels is computationally inexpensive, a large number of them can be used without a significant increase in the total computational cost. Secondly, the authors report the variety of kernels used to be relatively great - length, dilation, padding, weights and bias are all picked randomly, generating a wide variety of features (though stride is always 1). The third difference relates to the second one in the fact that dilation is also strongly varied - capturing both long and short-term dependencies, which is unique. Lastly, and most importantly, Rocket uses a novel approach in which not only the maximum of the feature maps generated by the random kernels is used, but also the proportion of positive values (*ppv*). This enables the classifier to take into account the prevalence of a certain feature. The authors note that this is the main addition that sets the success of Rocket apart from other random kernel methods.

When training or testing, the target data is first transformed using the generated random kernels. Using these random kernels, random "features" are extracted. This output can then, in the case of classification, be used as an input to a classifier, for which the authors note that linear classifiers are often most effective. In the default implementation, the authors use a ridge-regression classifier.

Dempster et al. have evaluated Rocket on the UCR-archive, and report Rocket to be competitive with state-of-the-art methods. They note that Rocket is able to obtain the best mean rank across the 85 'bake-off'-dataset compared to all used baselines. Rocket is also evaluated by Zerveas et al. [41], coming second in their ranking of evaluated methods, which further solidifies our choice to include it in our experiments.

3.3 Transformer

Transformer-based methods have taken the field of Natural Language Processing by storm. The ability to efficiently make use of long-term dependencies sets these models apart from previous methods. Recurrent Neural Networks, for example, allow for a lot of flexibility when it comes to sequence-dependent tasks, but they can suffer from vanishing gradient problems when the input sequences become longer [25]. The Transformer architecture makes use of self attention mechanisms over the full input sequence, allowing it to attend to different parts of the input without applying recurrence. Self-attention in the Transformer architecture furthermore consists of relatively inexpensive computational operations that can be done in parallel, allowing for much more efficient use of computational resources. Additionally, transformer-based architectures such as BERT [12] have been shown to be more robust in transfer-learning than previously used RNN-architectures.

The same properties that make the Transformer-based architectures effective in NLP tasks, also make these models a promising addition to the field of multivariate time-series classification. Zerveas et al. show a variant of the transformer architecture to be effective in a variety of multivariate time-series tasks, for which they report state-of-the-art performance. This architecture forms the basis for our proposed architectural additions and the experiments done in Section 7.

The following subsections will further explain the main building blocks of the Transformer-architecture and will introduce the time-series classification architecture as presented by Zerveas et al.

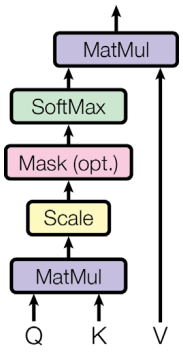

Figure 3: Scaled dot-product attention[5]

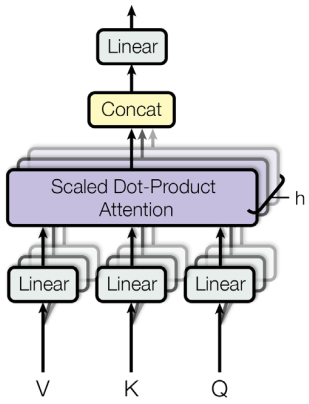

Figure 4: Multi-head attention[5]

3.3.1 Attention-module

The transformer architecture makes use of Scaled Dot-Product Attention units (see Figure 3). The input to the attention-units consists of a set of queries, keys and values of dimensions d_k, d_k and d_v respectively. First, the dot-product of the queries and keys is calculated. This essentially tries to map the queries to their appropriate keys, creating a similarity score between the two. The model learns to generate these queries and keys in such a way that this similarity-score can attend to different parts of the input sequence based on relevance.

The next step differs from normal dot-product attention. The authors noted that normal, additive attention, outperformed normal dot-product attention for large values of d_k. They theorize this to happen due to the dot product growing too large in magnitude, which results in minute gradients in the softmax layer. This is why the output of the dot-product module is scaled by $\frac{1}{\sqrt{d_k}}$ to prevent values from getting too large. The result is then passed through the previously mentioned softmax-function.

The output of the attention-module is thus calculated as follows:

$$\text{Attention}(Q, K, V) = \text{Softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{2}$$

Where Q, K and V denote the matrices for the queries, keys and values respectively.

3.3.2 Multi-Head Attention

Multi-Head Attention first uses h linear layers in parallel, which learn to project the input values, keys and queries in different ways. On each output, scaled dot-product attention is then applied (see Figure 4). In theory, this allows the model to relate parts of the input sequence to itself in different ways. Additionally, the calculations can be done in parallel, allowing for more efficient use of computational resources. The output of the attention-modules are then concatenated and linearly projected into the expected dimensionality.

Multi-head attention is thus computed as follows:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{Head}_1, ..., \text{Head}_{\text{n\_heads}})W^O \tag{3}$$

With:

$$\text{Head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{4}$$

Where $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$ and $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ denote the linear projections (resp.) of the queries, keys and values for Head_i. Attention(...) is defined in Equation 2.

[5] Image reproduced from *Attention is all you need* [39] under its allowed usage in scholarly works by Google

3.3.3 Transformer Encoder

In *Attention is all you need*, the Transformer encoder is created using a stack of N encoder layers, consisting of Multi-Head Attention, followed by a feed-forward layer, as depicted in Figure 5. Residual connections are added between the in- and output of both the feedforward layer and multi-head attention layer, both followed by layer normalization. The addition of the residual connections improves the training performance by making it possible for the gradients to flow through the network more easily.

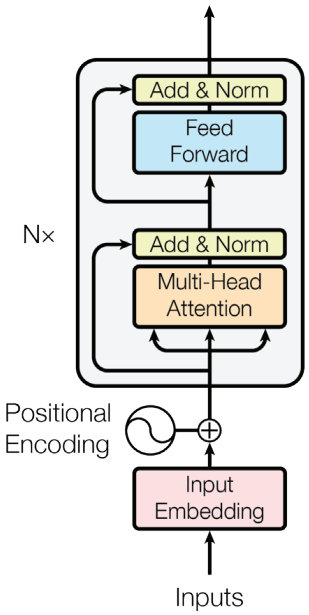

Figure 5: The original Transformer-encoder layer. Multiple encoder layers ($\times N$) are stacked to create the full encoder[5].

In the original Transformer-model architecture, the output of the stacked encoder layers is then fed into decoder-blocks that are used in sequence-to-sequence tasks. In our used architectures, this decoder-block is not included as it is less suitable for classification tasks, which is further explained in Section 3.3.4.

3.3.4 Time-series Transformer

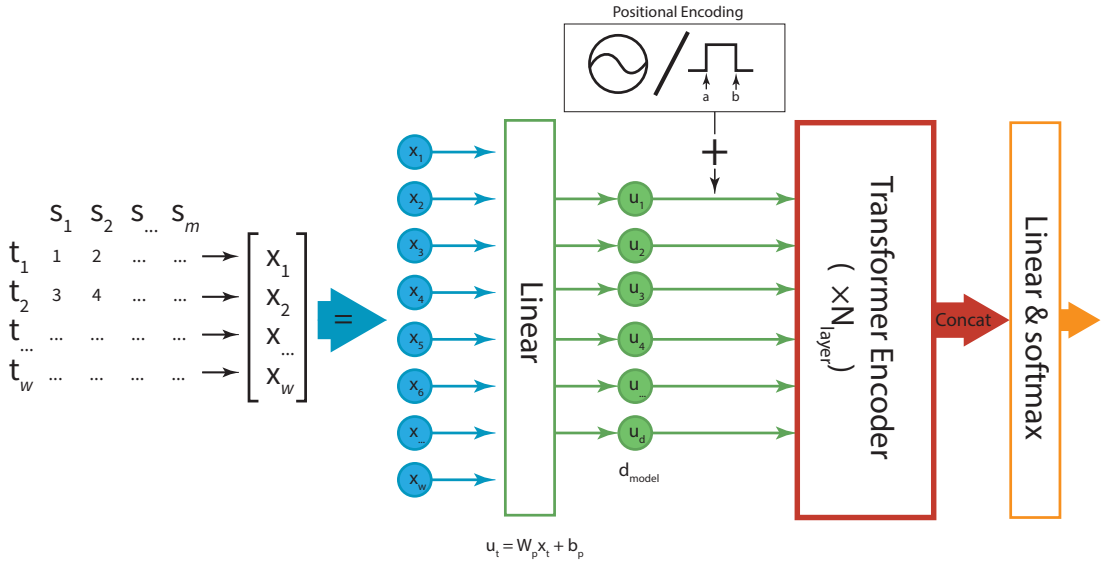

Figure 6: An overview of the TST-classification architecture as presented by Zerveas et al.

The transformer architecture as presented by Zerveas et al. - henceforth called **T**ime **S**eries **T**ransformer (**TST**) - differs slightly from the architecture presented in *Attention is all you need* (see Figure 6). In particular, the authors forego the inclusion of the decoder-part of the original architecture in favor of a generally applicable network using only the encoder part of the original architecture. Since the decoder would need the masked ground truth sequence as an input, this makes the full architecture less suitable for classification tasks.

The TST-model takes a sample $X \in \mathbb{R}^{w \times m}$ as input - consisting of a multivariate time-series of length w, with m dimensions equal to the amount of features. In the case of the radial drilling problem, m would correspond to the amount of sensors recording data, and w would correspond to the amount of time-steps per sample

Input X is passed through a linear layer, outputting the full input sequence into a specified amount of dimensions (d_{model}):

$$u_t = W_p x_t + b_p$$

Where $W_p \in \mathbb{R}^{d \times m}$ and $b_p \in \mathbb{R}^d$ denote the learnable parameters of the linear layer. u_t is analogous to the word-embeddings used in NLP tasks. Similarly, a positional encoding is then added to this input sequence, which allows the model to take the order of the input sequence into account. In the original method, *sine* and *cosine*-functions are used to add this positional encoding to the word-embeddings. Zerveas et al. have experimentally found that in their time-series experiments, a learnable positional encoding showed better results. The positional encoding is generated using random samples from a uniform distribution between -0.02 and 0.02.

Zerveas et al. have applied their proposed model on a variety of time-series classification tasks from the UAE-datasets (see Section 5.2) and have reported the model to outperform all other state-of-the-art time-series classification methods.

4 Contributions

In this section, the main contributions of this work are introduced, both in terms of the (novel) classification methods tested and the machine learning pipeline that we have created.

4.1 Time Series Classification Pipeline

Multivariate time-series classification is a difficult task, for which not many open-source tools are available. This makes the process of annotating the data and selecting a viable classification method an inaccessible and time-consuming process for the average user. We have implemented several open-source tools in order to create a cross-platform general-use machine learning pipeline for time series classification tasks.

In the following subsections, the 3 components of this pipeline will be discussed:

- **MVTS-Analyzer** - A platform for visualizing, analyzing and annotating multivariate time-series data.
- **Configurun** - A cross-platform python package app for (remotely) managing and running python configurations.
- **MVTS-Learner** - A machine learning framework built using Configurun - specifically for multivariate time series (classification) tasks.

4.1.1 MVTS-Analyzer

We implement *MVTS-Analyzer: an open-source, general use multivariate time series annotation and analysis tool*[6]. Although some time-series annotation tools exist, these are often limited in their use when dealing with larger continuous datasets. The most promising option we found, was *Labelstudio*. This annotation-environment is implemented in a web-UI, which is not ideal when dealing with a large amount of data-points. Furthermore, multivariate time-series are plotted in multiple parallel plots, making it difficult to keep track of all values when dealing with more than 2 sensors at the same time.

MVTS-Analyzer was developed in Python using PySide6 (a python-wrapper around Qt) with `Matplotlib` and `Pandas` at its core.

In particular, our implemented tool is useful when visualizing and annotating data with significant difference in scale between the features, as is the case with the Radial-Drilling dataset for which MVTS-analyzer was originally designed. Its main strength is its flexibility - we can directly operate on the pandas data-frame that was loaded into the tool, allowing for easy data manipulation.

The main window consists of a plot that allows the user to label individual time-steps using lasso, square and range selection tools (see Figure 7). Selection can be done additively, subtractively or complementarily. The user can then set the label of the selected data-points. They can either create a new label in the side-view, or select from all available labels for the selected label-column. Time series can be easily exported and loaded using several formats.

A side-window is made available to the user in which the domain of the selected x-axis is continually determined such that the user can select which part of the data they would like to have in focus. Each time the focus is changed, the data is normalized to the new focus window. Every currently selected column (e.g. sensor) gets its own axis that is individually moveable and scalable to allow for easy inspection of the data. The user can furthermore select the columns to display, the font size to use, the plot type and the plot-coloring method.

The user can also open multiple views of the same loaded dataset at once. Data-point-selection is shared between all views, and the user can choose to plot different variables against each other in each view. This allows for easy selection of outliers, as well as easy isolation of

[6] https://github.com/Woutah/MVTS-analyzer

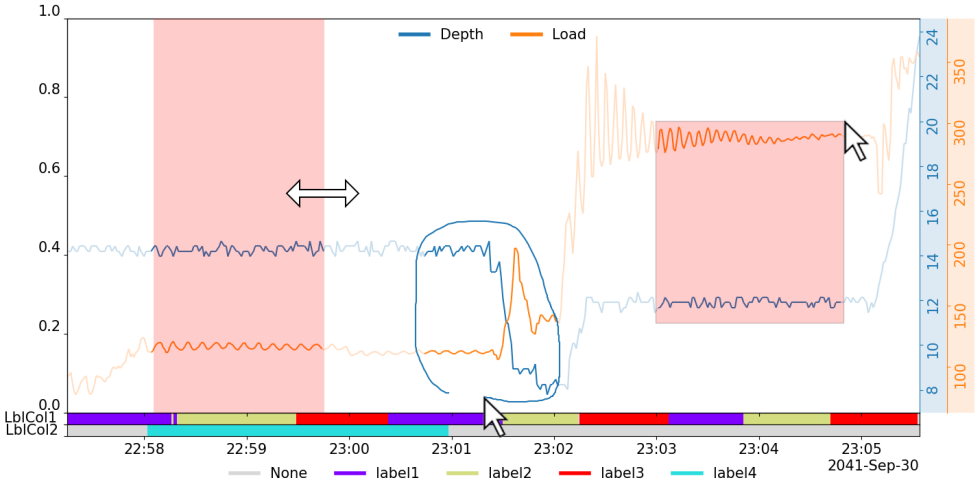

Figure 7: The data-viewer in MVTS-Analyzer. The user can select data-points using a variety of selection methods, and label them accordingly. Multiple label-columns can be added that can then be used when splitting data.

specific patterns in the data which might be difficult to label when only viewing the data over the time-domain.

The python-window allows for on-the-fly python code execution on the currently loaded dataframe, as well as the current selection. Python-scripts can be saved for later use, allowing for easy re-use of the same data processing methods on different (sub-)datasets.

When annotating, we can create as many label-columns as we want, allowing for more complex labeling methods. In turn, we can use these columns in our implemented *MVTS-Learner*-framework to split the data to make sure train, validation and test-datasets respect the desired group-separation.

4.1.2 Configurun

Although some tools exist to automatically generate UI's based on command-line arguments, the user-friendliness of these tools quickly deteriorates when dealing with more complex configurations. In many cases, it would be desirable to disable or enable certain options based on the value of other options in order to keep the UI clean and intuitive. For example: it would make no sense to show options for Neural-Network based training when the user has selected a Random-Forest classifier. Furthermore, tools that also enable the user to quickly load and save such configurations are not readily available. Especially an all-in-one solution that also allows for easy (remote) queuing and running of configurations has, to our knowledge, not been created before. We implement Configurun[7], a cross-platform PySide6-based package with an app to save, manage and (remotely) run python configurations. Configurun can be easily installed from PyPi, for example with pip using: `pip install configurun`.

Configurun was designed mainly with machine-learning tasks in mind, but can be used for any python script or framework that takes arguments as an input. Editable UI's are created automatically based on either an `argparse.Argumentparser` or python-`@dataclass`(es). Editing-widgets are based on the types provided by this argparse-object or the `dataclass`. Supported types include `int`, `str`, `bool`, `float`, choices of any type using a `Literal`, as well as (complex) combinations of these types which can be created quickly using `list` and `union` type-hints (see Figure 8). We can dynamically switch between configuration templates based on user-choices in

[7]https://github.com/Woutah/configurun

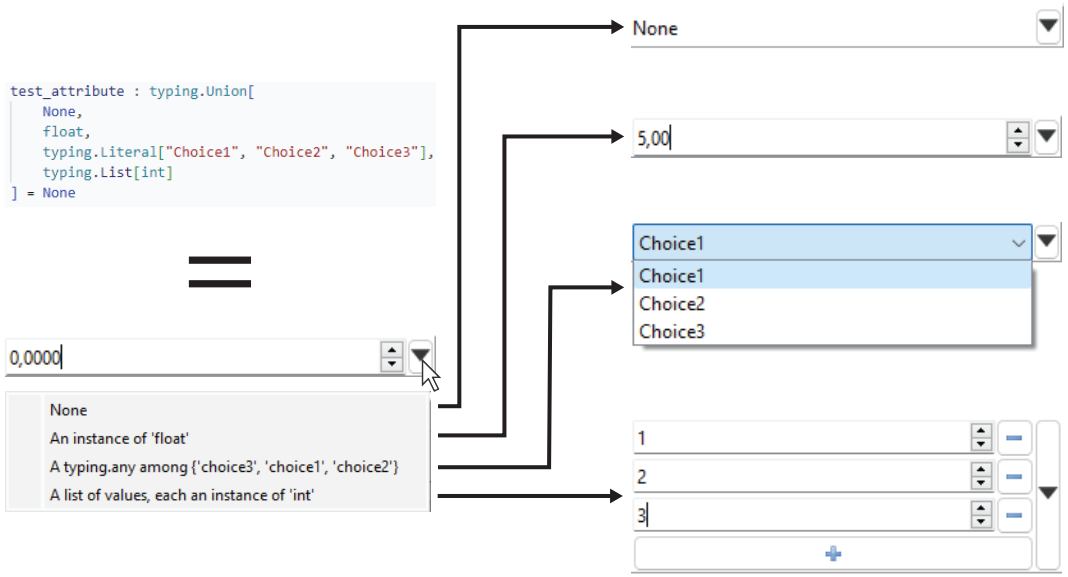

Figure 8: An example of how a single attribute in a python `@dataclass` is presented to the user. The dataclass is automatically converted into a UI in which editing-widgets are provided based on the type-hints.

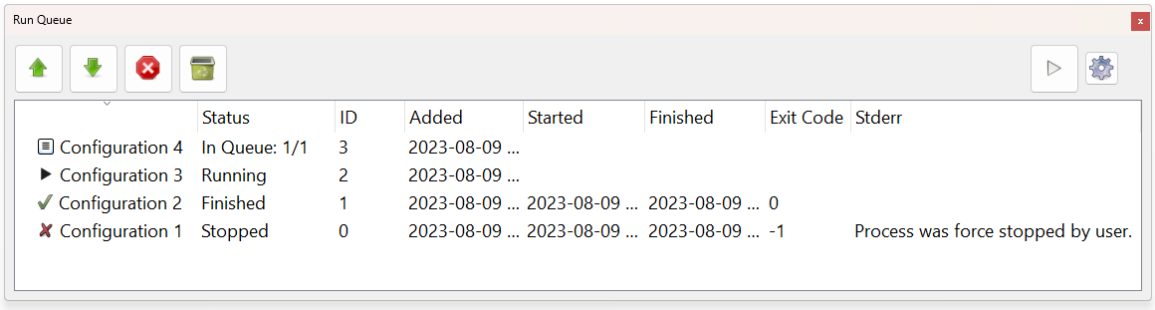

Figure 9: An example of several configurations in the Run-Queue.

the current configuration, allowing the user to quickly generate intuitive testing-environments for any existing python framework.

The main configurun-editor window displays a sub-window with a workspace-explorer in which we can easily save, load and edit configurations. In this way, we can keep track of the configurations we have run and create new configurations based on previous ones. Parameters that have changed from the default values are highlighted, allowing the user to keep track of what changes have been made to large configurations.

Configurun saves the workspace-states when quitting, allowing the user to resume work at a later time. Queued, running and finished jobs are kept in the Run-Queue. In the same window, we can start, stop, remove and edit runs, as depicted in Figure 9. Auto-running-mode can also be enabled here, which will automatically start the next job when a process-space opens up. We can run multiple configurations simultaneously by specifying the amount of processes to use.

The Configurun-app can either be run entirely locally, or in a client-server setup. In the latter case, the user can start a server-instance on the remote machine. A local client-instance can then be used to interface with the remote server-instance as if the user is running the normal local app, adding jobs to the queue and monitoring the output of running, queued and finished jobs. Multiple clients can connect to the same server-instance, so multiple users can work on the same remote machine at once.

4.1.3 MVTS-Trainer

On top of Configurun, we implement *MVTS-Trainer: an open source framework for time-series classification tasks*[8].

This implementation is based on the transformer-based framework for multivariate time-series representation learning as presented by Zerveas et al.[9], and combines this existing framework with the Rocket-feature extraction method and a multitude of classification algorithms from the Scikit-learn [29] library. We also integrate XGBoost, using the implementation made by its authors [10]. We have furthermore implemented our own Convolutional-Transformer architecture(s) (see Section 7), which we have also made available in this framework.

All combination of models, datasets and preprocessing methods can be easily selected using the Configurun-UI, at which point the appropriate sub-options for model, dataset and training are automatically presented to the user.

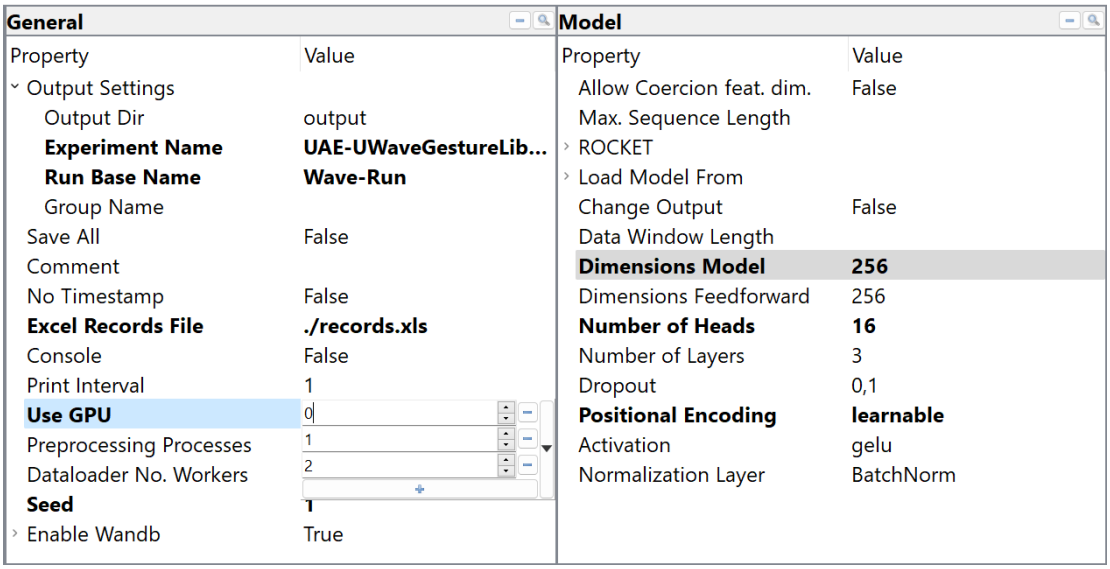

Figure 10: An example of the general and model sub-options. The applicable options are automatically selected and shown based on the model selected in the main-options sub-window and allow for easy customization of the training/testing parameters.

An example of the general-options and model-options can be seen in Figure 10. The model-options are selected automatically based on the currently selected model-architecture. The framework makes it very easy to run a variety of experiments on a multitude of datasets. We have split the options into main-, general-, model-, dataset- and training-options. All sub-option choices are then loaded dynamically based on the selected main-options. As noted before, we can save, load and edit these configurations and add them to the run-queue to be run (remotely). Options that have changed from their default values are printed in bold, allowing us keep track of what changes are made to each configuration.

All target-algorithms from the sklearn-library are automatically loaded into the UI and every argument of each of the individual algorithms is then automatically converted into a UI-element based on the class-signatures and type-hints. The description of each of the arguments is loaded automatically from the Scikit-learn documentation and is displayed on-hover. Adding more Scikit-learn algorithms to the framework can be done by simply adding the desired class to the list of algorithms to be loaded.

[8] https://github.com/Woutah/MVTS-Trainer
[9] https://github.com/gzerveas/mvts_transformer
[10] https://github.com/dmlc/xgboost/tree/master

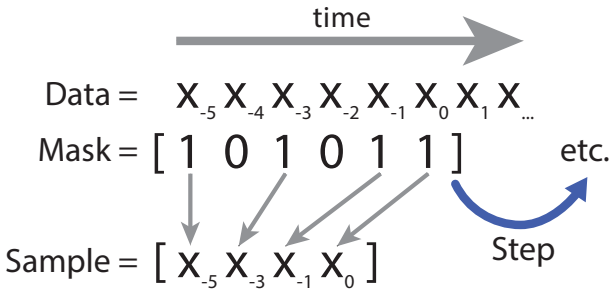

Figure 11: The time-series window sampler with a frame with of 6. We can specify the boolean mask and step size to customize our sampling method.

The framework is designed in such a way that users can also add their own custom models, datasets and preprocessing methods. By creating a python-`@dataclass`, all options for the new implementation can then be presented to the user in an intuitive manner.

As we have integrated the framework by Zerveas et al., we should be able to load any of the datasets reported in the paper in question (including time-series regression datasets/tasks). We note that, in this thesis, we have primarily made use of the time-series-classification type datasets. These datasets were provided in the ts-format.

We have additionally implemented a general-use data-loader for *Pandas*-dataframes, compatible with the output of MVTS-Analyzer. This means that any dataset which is loaded (and annotated) in MVTS-Analyzer, can also be loaded into the framework for training and testing. This dataloader provides a variety of functions to split the data into train, validation and test sets, to extract the actual samples from the data and to validate the data. The core feature for converting continuous time-series data into multiple samples, is the time-window-sampler. This sampler allows the user to select a window-size, a step-size and a boolean mask to determine which time-steps should be included in the sample. The sampler will create a sample for each window-step in the data, using the provided window-size, step-size and padding settings. The boolean mask additionally allows the user to use a variable step-size in the actual sample as depicted in Figure 11.

Using this window, we can resample large continuous datasets using a fixed-window size with a boolean mask and the desired step-size. Depending on the problem at hand, we can decide to what extent long-term dependencies should be taken into account. In Section 7, we experiment with these settings on the Radial-Drilling dataset.

Finally, we have integrated Wandb [5][11] and Tensorboard into the framework, allowing the user to quickly log all training and testing results to either the (online) Wandb-platform or to a local Tensorboard-instance.

4.2 Classification Methods: (M)CTST and XGBR

The following subsections describe the contributions made in terms of novel classification methods, all methods are evaluated in Section 7.

4.2.1 Convolutional Transformer - (M)CTST

Certain time-series patterns evolve over time. Canonical transformer-encoder layers, however, do not immediately take into account local context when computing the self-attention vector if the feature-vectors are directly inserted into the Transformer-Encoder - as this means that the attention vector is calculated in a point-wise manner, as is also noted by Li et al. This means we might miss such evolving patterns.

[11] https://wandb.ai/

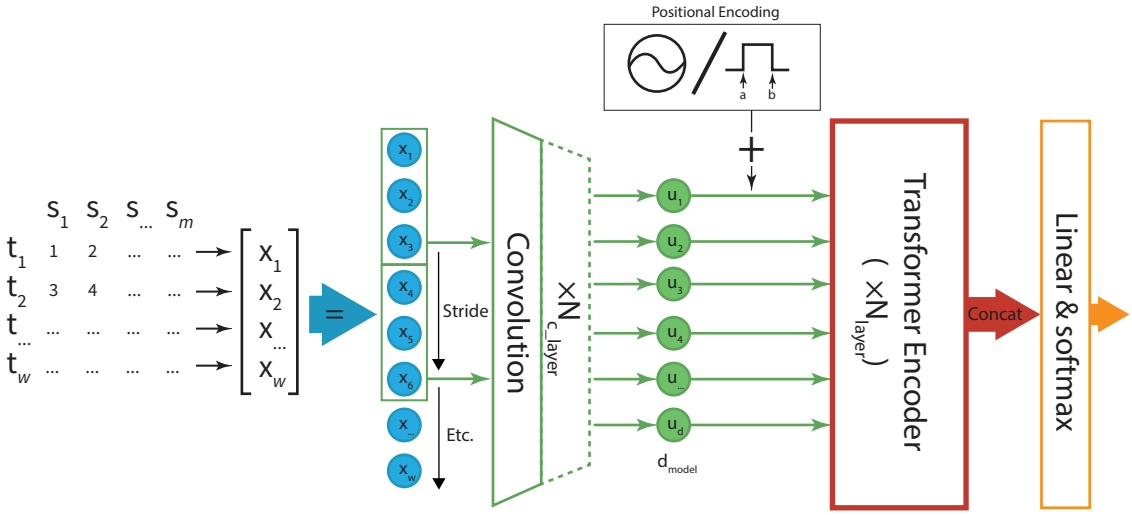

Figure 12: The (M)CTST-architecture, based on the TST architecture, self-attention is computed over features using either one (CTST) or multiple (MCTST) convolutional layers.

Convolutional layers have previously been used in combination with Transformers in time-series sequence-to-sequence models [27]. Zerveas et al. propose a similar improvement for future work, and note that from their initial tests, the convolutional transformer showed promising results on datasets consisting of longer (lower-dimensional) time series.

Based on the work of Li et al. in time-series forecasting tasks, and based on the suggestions for future work provided by Zerveas et al., we design and implement our own architecture using the transformer-encoder architecture.

A schematic overview of this architecture is depicted in Figure 12. In accordance with the naming scheme of Zerveas et al. for their Transformer-based convolutional classifier Time Series Transformer (**TST**). We refer to the models with a single convolutional layer as Convolutional Time Series Transformer models (**CTST**) and to the models with multiple convolutional layers as Multi-Convolutional Time Series Transformer models (**MCTST**).

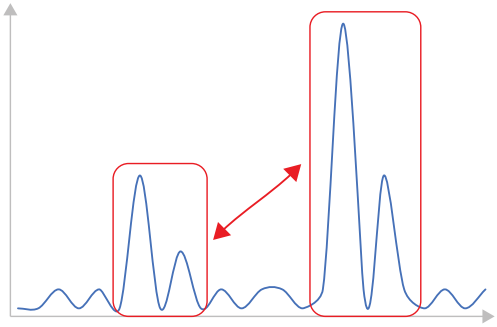

Figure 13: Convolutional self-attention allows for local context to be taken into account, extracting (local) features/patterns before self-attention is applied.

The intended use of the convolutional layers is to capture local dependencies, allowing for feature-extraction on a local level before self-attention is applied. An example of this principle is shown in Figure 13. A fully connected layer would have trouble capturing these local dependencies in a repeatable manner (especially for longer sequences), and using self-attention directly on the time-sequence would result in point-wise self-attention, which might miss such patterns.

Using one or more 1D convolutional layers theoretically allows these patterns to be extracted more easily.

Additionally, by using sub-sampling or dilation (Section 9), the receptive field of the convolutional layers can be increased. This might allow the model to learn to create a more condensed representation of long input sequences.

4.2.2 XGBR

In addition to evaluating the performance of Rocket (in combination with a RidgeCV-Classifier) and XGBoost individually, we also experiment with a combination of the two. We use the output features of Rocket as an input to the XGBoost-classifier and denote this method as **XGBR** (XGBoost-Rocket).

At the time of writing, a concurrent work has applied this combination of classifier and feature extraction on a 3D-printer fault monitoring problem [31]. As far as we know, this is the first time this combination of feature-extractor and classifier has been applied to time-series classification tasks.

This method aims to make use of both the feature extraction capabilities of Rocket and the classification capabilities of XGBoost, while also taking advantage of the computational effectiveness of both methods. We compare the performance of this method to the default RidgeCV-classifier proposed by the creators of Rocket.

5 Datasets

5.1 Radial-Drilling

We operate on data provided by Radial-Drilling Europe B.V.. We have been given access to a dataset consisting of anonymized well operations with a couple of hours of data per operation. The sensor-data consists of 1-second measurement intervals of the four most important sensors:

1. Depth - The hose length that has currently been inserted into the well.
2. Load - The load on the hose-mechanism.
3. Fluid-Flow - The fluid-flow rate through the hose, which determines the speed of the milling and jetting-nozzle.
4. High-Pressure - The pressure of the fluid in the hose.

Using MVTS-Analyzer, all individual well-operations were given their own ID, after which process-labels were assigned according to the day reports provided by Radial-Drilling Europe B.V.. The dataset differentiates between six different classes in the radial-drilling process: waiting/standby, pull-out-of-hole (POOH), run-in-hole (RIH), tagging/run-in-shoe (RIS), jetting and milling.

The datasets were then sampled by sliding a sampling-window of a fixed size over the time-steps in each well-operation as described in section 4.1.3, and depicted in Figure 11. To select the optimal size and sampling-method for this window, we experiment with four different settings denoted by a version number from 1 to 4:

- **DrillingV1** A sliding window with a width of 60 seconds, each sample consisting of all 60 time-steps in this window.
- **DrillingV2** A sliding window with a width of 6000, steps between each sampled index increase exponentially as we go further back in time, from 1 to 256, in total, the sample consists of around 1200 time-steps.
- **DrillingV3** Analogous to V2 - but steps increase more quickly, resulting in a total size of 600 time-steps per sample.
- **DrillingV4** A sliding window of width 600, of which all indexes are sampled - corresponding to the last 10 minutes of data for each sample.

Every dataset is sub-sampled such that the classes are approximately balanced. The dataset is then split into 70% train- and 30% test-set similar to the used public-datasets with a larger amount of samples. During hyperparameter tuning, the train-set is further split into the actual train- (80% of total train-set) and validation-set (20% of train-set), following the example of Zerveas et al.

We note that due to the severely limited amount of distinct wells present in the dataset, and due to the fact that the well-locations were not annotated, we were unable to create a test-set consisting of completely distinct well locations, which would be most desirable as this would come closest to a real-world performance evaluation in which all (geological) well-parameter could differ from the train-set (see Section 9).

We make use of Stratified Group K-Fold splitting to try to preserve the class-distributions as much as possible, while ensuring that the same well-operation is never present in both the train- and val- or test-set. The fact that well-operations are kept distinct, makes it so that the model is never trained on samples from "in between" samples in the validation- or test-set, from which information could be leaked. Splitting the dataset in this manner, results in a more realistic performance evaluation.

5.2 UAE-Datasets

In addition to the novel classification task on the Radial-Drilling dataset, we experiment on a variety of datasets from the UEA Time Series Classification Repository [3]. We follow Zerveas et al. by example and use 10 datasets with a diverse amount of training samples, dimensions, sequence lengths and classes. We note that we did not use the InsectWingbeat-dataset because Zerveas et al. do not take this dataset into account for their calculation of the weighted accuracy due to the fact that they could not run the *Rocket*-experiments on this dataset. Additionally, we noted that loading-times for this dataset were prohibitively long.

A summary of the dataset properties is shown in Table 1.

Table 1: Overview of the used datasets from the UEA Time Series Classification Repository.

Dataset	Train size	Test size	Dimensions	Length	Classes
EthanolConcentration	261	263	3	1751	4
FaceDetection	5890	3524	144	62	2
Handwriting	150	850	3	152	26
Heartbeat	204	205	61	405	2
JapaneseVowels	270	370	12	29	9
PEMS-SF	267	173	963	144	7
SelfRegulationSCP1	268	293	6	896	2
SelfRegulationSCP2	200	180	7	1152	2
SpokenArabicDigits	6599	2199	13	93	10
UWaveGestureLibrary	120	320	3	315	8

As can be seen, several of these datasets contain very few training samples. The predefined test- and train-splits are used during training and testing. For hyperparameter tuning, the train-set is split into a 20%-80% validation and train-split respectively in accordance with the training procedure used by Zerveas et al. A more in-depth description of the training procedure can be found in Section 6.

In Table 2, a short description of the classification-objective of each dataset is given.

Table 2: Summary of the classification objectives per dataset.

Dataset	Objective	Balanced
EthanolConcentration	Determine alcohol concentration of a sample using spectroscope data [26].	Yes
FaceDetection	Determine whether subject is looking at a picture of a person or scrambled data using MEG data.	Yes
Handwriting	Determine handwritten letters of the alphabet based on x, y, z motion data.	2-7 samples per class
Heartbeat	Determine whether a heart sound-recording is normal or abnormal.	75%/25% resp.
JapaneseVowels	Determine the person based on utterances of the vowels 'e' and 'a'.	Yes
PEMS-SF	Determine day of the week based on car occupancy rate in car lanes.	Yes
SelfRegulationSCP1	Determine mouse movements made by subject based on cortical potential measurements.	Yes
SelfRegulationSCP2	Determine mouse movements made by patient with ALS based on cortical potential measurements.	Yes
SpokenArabicDigits	Determine spoken Arabic digit based on MFCC.	Yes
UWaveGestureLibrary	Determine gesture based on x, y, z data.	Yes

6 Pre-Review of related work

Zerveas et al. provide the model architecture hyperparameters for their proposed TST-model for the supervised learning tasks. The authors have run hyperparameter optimization on each individual dataset in the UAE-archive to determine the number of transformer-encoder blocks (*n_blocks*), the number of attention heads (*n_heads*), the dimensionality of the self-attention in/output (*d_model*) and the dimensionality of the linear feed-forward layers in each transformer-encoder block (*d_FFW*). In practice, however, we were not immediately able to reach the reported results in the paper. After reaching out with questions about the training and testing procedure, the authors have remarked that (at least) both the subsampling-rate and batch-sizes were also tuned separately for each dataset. The authors, in their paper, noted that the amount of epochs trained was treated as a hyperparameter: 20% of the training data was used as a validation set for hyperparameter tuning, after which the full train-set was used to train the model a pre-determined amount of epochs.

Unfortunately, the authors were no longer able to provide us with the exact hyperparameter settings for each dataset. This made it difficult to reproduce the reported results.

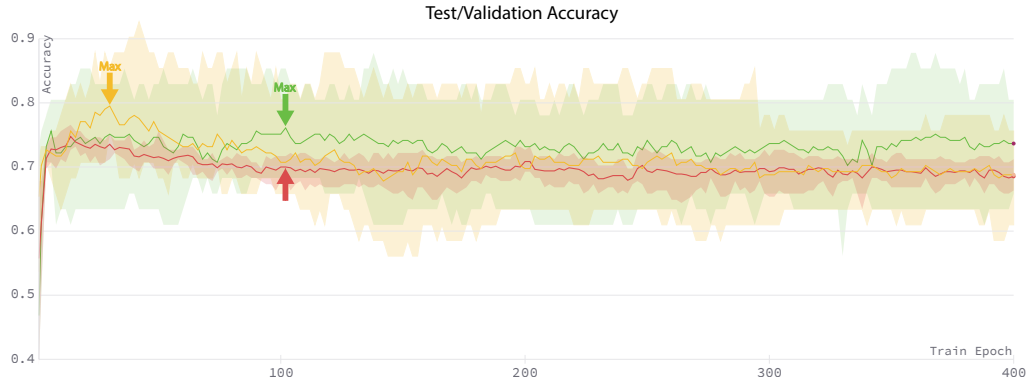

Figure 14: Test-accuracy of provided hyperperameters, Validation-accuracy of provided hyperparameters, Validation-accuracy of own hyperparameter optimization - all over 5 folds.

We illustrate this point using figure 14. For this example, we picked the Heartrate-dataset, for which the most hyperparameter-settings were known, as this is one of the few datasets for which the batch-size was given due to the authors' extra experiment regarding the difference between Batch-Normalization and Layer-normalization. We set all model-hyperparameters to the reported optimal settings (*n_blocks*=1, *n_heads*=8, *d_model*=64, *d_FFW*=256), after which we trained the model for 400 epochs 5 times using the default settings provided in the framework, with *BatchNormalization* and a Learnable positional encoding - as recommended by the authors.

In green, we show the validation-accuracy on the Heartbeat dataset over 5 folds, with 20% of train-set used as the validation set in each fold. In red, we show the test-accuracy over 5 runs, when using 100% of the training data for training - note that this is only done for illustrative purposes.

For the validation-accuracy we can see that the maximum mean-validation-accuracy of 0.761 is reached at epoch 102 - indicated by the green arrow. According to the authors' testing procedure, we would then evaluate model at the same epoch. At this epoch, (as denoted by the red arrow) the test accuracy is only 0.700 ± 0.017. The reported accuracy on this dataset is 0.776, which is significantly higher.

Because this pattern was observed for many of the datasets, we have decided to re-run the hyperparameter optimization for each dataset to be able to also compare our found optimal

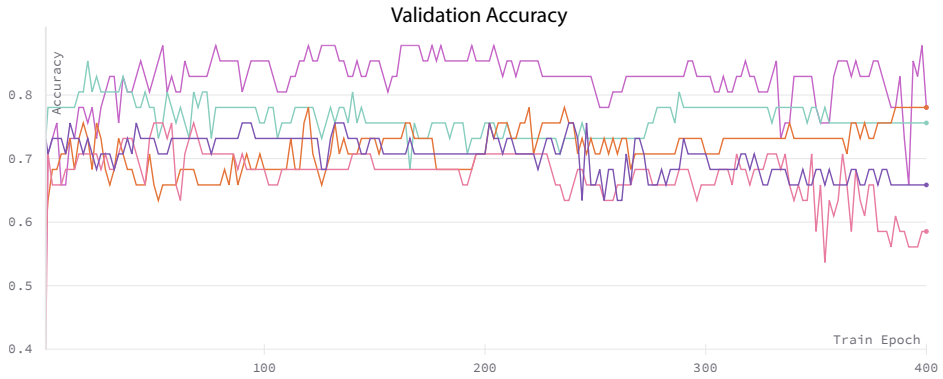

Figure 15: Heartrate validation accuracy per training epoch - over 5 folds of the validation set. All runs use the same hyperparameter settings. All folds have a 20%-size validation set. Each fold in a different color.

hyperparameters to the reported optimal hyperparameters. We not only vary the subsampling rate and batch-size, but also compare the architectural hyperparameters to the reported optimal settings. Due to the very small training-sizes of several of the datasets, we have decided to use 5-fold or 10-fold validation for each hyperparameter-optimization-run (based on the size of the dataset in question) to be able to evaluate the performance of the hyperparameters in a more meaningful way. We illustrate this using 5-fold validation on the same dataset, using the same hyperparameter settings, as depicted in Figure 15. We can see that, depending on the split, the validation accuracy can vary greatly. At the absolute maximum of almost 90% accuracy, the minimum of one of the other folds has an accuracy of around 68% - a difference of more than 20%, which is only caused by how the validation-split was made. Furthermore, the amount of epochs needed to reach the optimal accuracy also varies between folds.

For many of the datasets, such as for the Heartbeat-dataset used in our example, we were able to find a set of hyperparameters which performed better on the validation set. An example of this can be seen in Figure 14. In orange, we show a run with custom hyperparameter settings for which a maximum accuracy of 0.795 was reached at epoch 30. In this example we used the following settings which differ from the authors: fixed positional encoding, n_blocks=4, $batch\_size$=64 and d_FFW=128.

At epoch 30, using our found optimal hyperparameters, we reached a mean accuracy of 0.699 ± 0.017 on the test-set - which is in line with our earlier test that resulted in about the same accuracy - though still significantly lower than the reported accuracy of 0.776. For the Heartrate-dataset, we were unable to find a set of hyperparameters which both outperformed the abovementioned hyperparameter settings on the validation-set, and also matched the reported test-accuracy of the authors.

We specifically note that we also did some precursory investigation in the reported performance improvements when pre-training the Transformer-classification architecture on the UAE dataset in an unsupervised manner. We found that tuning the hyperparameters played a much larger role in the final performance of the model. As the reported accuracy-improvement over non-pretrained models was only 0.6%, we decided to limit the scope of our experiments to not include this pre-training procedure.

The final results on the Heartbeat- and the other UAE-datasets using the author proposed architecture are investigated further in Section 7.

7 Experiments & Results

7.1 Computing Devices

All experiments with CPU-intensive tasks (in particular Rocket and the gradient-boosting methods) were done on the Leiden-University CPU DSLAB-servers with a 16 Intel Xeon E5-2630v3 CPU @2.40Ghz. GPU-intensive tasks (TST, CTST, MCTST) were either done on a desktop-pc with a RTX-3080TI, or on the Leiden University DSLAB-servers using either an NVidia GTX 980TI, an NVidia Titan X or an Nvidia RTX 2080.

7.2 Summary Tested Methods & Contributions

We note that the reported performance of *TapNet* (Zhang et al. [42]) offered no improvement over any of the mentioned methods on this selection of the UAE-datasets. As the average accuracy of 0.678 would rank it as the second worst performing (see Section 7), we forego implementing it in our framework and/or including the results in the results-table.

We train, test and evaluate the following existing methods:

- **Gradient Boosting (GBoost)** - using the scikit-learn implementation of the gradient boosting classifier (Section 3.1)[12].
- **XGBoost** - Using the XGBoost-library implementation(Section 3.1.1)[13].
- **Rocket** - using the implementation for multivariate time-series data provided by the authors [35] in combination with a Ride-CV classifier (Section 3.2)[14].
- **Time Series Transformer (TST)** - as presented in *A transformer-based Framework for Multivariate Time Series Representation Learning* [41] (Section 3.3.4).

In addition to these methods, we propose and implement the following novel contributions to the classification-framework, and train and test them on the datasets in question:

- **Convolutional Time Series Transformer (CTST)** - a transformer-based classification architecture with a single convolutional layer used for feature extraction (Section 4.2.1).
- **Multi-Convolutional Time Series Transformer (MCTST)** - a transformer-based classification architecture with multiple convolutional layers used for feature extraction (Section 4.2.1).
- **XGBR** - a combination of Rocket and XGBoost, using the output of Rocket as input to the XGBoost-classifier (Section 4.2.2).

7.3 Training and Testing Procedure

For the Radial-Drilling datasets, a single validation set consisting of 20% of the train-set is used. The best performing (as determined using the method described below) model is retrained using 3 folds of the train-set, every time the best performing model of that fold (as determined by the accuracy on the validation set) is then evaluated on the testset. For the non-NN methods, for each fold, the final found model is evaluated on the test-set.

Hyperparameter optimization on the Radial-Drilling datasets was performed for TST, CTST, MCTST and Rocket. Both GBoost and XGBoost were tested using the default settings. XGBR uses the default settings for both XGBOOST and Rocket.

[12] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html
[13] https://xgboost.readthedocs.io/
[14] https://github.com/ChangWeiTan/TS-Extrinsic-Regression/blob/master/models/rocket.py

For all UAE-datasets, we used either 5 or 10 fold validation-evaluation during hyperparameter tuning depending on the size of the train-set to get a more stable evaluation for each of the settings (also see Section 6). The transformer-based methods (TST, CTST, MCTST) were optimized by doing between 10 and 15 hyperparameter optimization training-runs on each of the 10 datasets, with the model-hyperparameters reported by the authors as the starting point for the optimization process. For Rocket and GBoost, around 8 runs were done to try to optimize the number of kernels and the number of estimators respectively. For XGBoost and XGBR, the default settings were used. The top-10 hyperparameter-runs and rankings can be found in Section A in the Supplementary Material.

After training, all hyperparameter runs are objectively ranked by first obtaining the maximum reached (mean) accuracy (averaged over all folds) on the validation set. For that epoch, the average accuracy over the last and next 10 epochs is then calculated to also weigh the stability of the chosen method. Both the maximum and "10-average"-accuracies are then ranked, at which point an average ranking is calculated from the two rankings, which serves as the final parameter-ranking-method.

To obtain the test-results, the model is then trained using the best found hyperparameters on 100% of the training set, for the amount of epochs at which the found maximum validation accuracy was reached.

The evaluation metric used is the accuracy. This is also the metric used by the creators of the TST-model, to which we compare our found results.

7.4 Results

We optimize the hyperparameter settings of the models according to Section 7.3. In the following subsections, the found performance is first compared to the reported performance by Zerveas et al., after which we compare the performance of the novel approaches to the authors' method. Lastly, the performance of all methods on the novel Radial-Drilling dataset is examined.

7.4.1 UAE Results compared to Zerveas et al.

We firstly compare our found results for TST to two of the main baselines that the same authors use in their paper (XGBoost & Rocket). In Table 3, our (left) found results are shown next to the results reported by the authors (right) for each of the datasets, for each of the methods.

	TST		**Rocket**		**XGBoost**	
Dataset	**Our**	**Author**	**Our**	**Author**	**Our**	**Author**
Eth. Concentration	0.280	0.337	0.381	0.452	0.449	0.437
FaceDetection	0.671	0.681	0.618	0.647	0.589	0.633
Handwriting	0.308	0.305	0.555	0.588	0.167	0.158
Heartbeat	0.699	0.776	0.710	0.756	0.693	0.732
JapaneseVowels	0.976	0.994	0.985	0.962	0.908	0.865
PEMS-SF	0.822	0.919	0.761	0.751	0.983	0.983
SelfRegulationSCP1	0.886	0.925	0.883	0.908	0.826	0.846
SelfRegulationSCP2	0.520	0.589	0.602	0.533	0.467	0.489
SpokenArabicDigits	0.990	0.993	0.997	0.712	0.970	0.696
Wave G. Library	0.866	0.903	0.918	0.944	0.759	0.759
Mean Accuracy	0.702	0.742	0.741	0.725	0.681	0.659

Table 3: Our (left) results compared to Zerveas et al. (right) found results for 3 of their used methods, on the 10 UAE-datasets. TST is the novel method proposed by Zerveas et al.

From these results, we firstly conclude that we were unable to exactly reproduce the accuracy-scores reported by Zerveas et al. for their proposed TST-model. With an average accuracy of 0.702, we fall short 4% of the reported average accuracy of 0.742 - which would, also according to self reported accuracies, put the TST-model's performance behind the state-of-the-art Rocket-method. Presumably due to our precursory hyperparameter optimization of the Rocket/Ridge-method, we were also able to outperform the reported accuracy of Rocket from 0,725 to 0,741. We observed that, even though the average accuracy of Rocket was higher for us, for several of the datasets, we nevertheless performed slightly below the reported accuracy. For example, Zerveas et al. report an accuracy of 0.452 on the EthanolConcentration dataset. We observed optimal performance on the validation set when the hyperparameter settings were set to default (10,000 kernels). Nevertheless, we observed an accuracy of only 0.381 on the test-set. For SpokenArabicDigits we can see a significant difference between the reported and found accuracy - whereas the authors reported an accuracy of 0.712 - our test-runs resulted in an average accuracy of 0.997, an increase of 0.285. This same extreme difference is also visible for XGBoost for the same dataset, where we observed an average accuracy of 0.970, compared to the reported accuracy of 0.696 on the SpokenArabicDigits dataset.

The largest differences between the authors' observations and our observations for the TST-model can be seen for the Heartbeat and the PEMS-SF datasets. For the Heartbeat dataset, we observed an accuracy of 0.699, whereas the authors report to have reached an accuracy of 0.776. This difference is even more clear for PEMS-SF, as we were unable to approach the reported accuracy of 0.919 - the maximum achieved accuracy was 0.831.

For XGBoost, our other found accuracies were mostly in line with the performance reported by the authors. Our average accuracy for the XGBoost-method on the UAE-dataset was 0.681, compared to 0.659 reported by the authors, so its performance was shown to be slightly higher from our experiments - mostly due to the aforementioned difference for the SpokenArabicDigits dataset.

We have reached an average accuracy of 0.741 using Rocket, and the author-reported accuracy for the TST-model was 0.742 (or 0.748 for the pre-trained version of TST), our found accuracy for the TST method is 0.702. We are thus unable to conclude that the TST-model outperforms the other state-of-the-art methods as mentioned by the authors, as we have found that both baselines performed better than reported, and TST performed worse than reported.

7.4.2 CTST & MCTST Results

	Ours						Ours
Dataset	**MCTST**	**CTST**	**TST**	**Rocket**	**GBoost**	**XGBoost**	**XGBR**
Eth. Concentration	0.398	0.324	0.280	0.381	0.439	**0.449**	*0.418*
FaceDetection	**0.679**	*0.668*	0.671	0.618	0.661	0.589	0.573
Handwriting	*0.327*	0.371	0.308	**0.555**	0.105	0.167	0.321
Heartbeat	0.658	0.678	0.699	*0.710*	**0.754**	0.693	0.717
JapaneseVowels	0.970	*0.975*	0.976	**0.985**	0.905	0.908	0.953
PEMS-SF	0.753	0.831	0.822	0.761	**0.996**	0.983	*0.949*
SelfRegulationSCP1	*0.885*	**0.889**	0.886	0.883	0.819	0.826	0.811
SelfRegulationSCP2	0.479	*0.524*	0.520	**0.602**	0.478	0.467	0.531
SpokenArabicDigits	0.992	0.988	*0.990*	**0.997**	0.970	0.970	0.986
Wave G. Library	0.880	0.862	0.866	**0.918**	0.659	0.759	*0.875*
Mean Accuracy	0.702	*0.711*	0.702	**0.741**	0.679	0.681	0.713

Table 4: Only our found test accuracies on the 10 publicly UAE-datasets are listed here. The best result for each dataset is highlighted in **bold**. The second-best result is underlined. The third-best result is printed in *italic*.

We run and test the CTST, MCTST and XGBR methods on the UAE-datasets. The results can be seen in Table 4.

As expected, the default Gradient-Boosting implementation performed very similarly to XGBoost. From our testing on the validation sets, we noted that on several datasets, some gain in accuracy was displayed when the amount of estimators was varied. In practice, however, this gain seems to have been negligible, as XGBoost outperforms GBoost - if only slightly. We note that from our experiments, we indeed observed a significant computational advantage of the XGBoost-method over the GBoost-method. In some instances, we could observe a speedup of more than a factor ten on the tested datasets.

The average accuracy of the CTST-model shows an average improvement of about 1% over the baseline TST-model. In particular, we see an improvement for the Handwriting-dataset, for which an accuracy of 0.371 was achieved using CTST, compared to 0.308 using TST (or 0.305 according to the results of the authors).

MCTST offers a small improvement over the TST and CTST-models for the EthanolConcentration and FaceDetection datasets. For EthanolConcentration, the observed accuracy of 0.398 is quite high compared to the other TST-methods. This result was achieved using quite a high subsampling rate (4) in combination with 2 convolutional layers of size 64 and 16 respectively. Due to the large scope of our optimization experiments, we were unable to also experiment with dilation rates for the convolutional layers, which, from this result, might have been fruitful to further explore. Especially for the longer-time-series data, capturing long sequences into a more compact representation in this way would be interesting. Even though we have simulated this behavior using subsampling, using this method means that, in some cases, more information might be lost than necessary.

For the performance of both the CTST and MCTST we must remark, however, that Zerveas et al. report an average accuracy of 0.742 for the canonical TST-model (see previous Section), which is still 0.031 higher than the average accuracy of 0.711 that we have found for CTST. Even though our results on the TST model showed an overall improvement when introducing convolutional layers, it is therefore difficult to conclude that the predictive performance is actually better than the canonical TST-model.

For the UAE-datasets, we can see that XGBR performs relatively well compared to the other baseline methods. With an average accuracy of 0.713, it comes in at second place in the accuracy ranking, just below the Rocket-method. Although XGBR never comes in first place for any of

the accuracies on the UAE-datasets, it does strike a good balance between the performance of Rocket and XGBoost. For most datasets, this balance is in favor of Rocket, which (on average) performs considerably better than XGBoost for the UAE-datasets. However, the opposite also holds true for the PEMS-SF-dataset, on which Rocket achieves an accuracy of 0.761 compared to the 0.949 achieved by XGBR - and on the EthanolCencentration-dataset, for which XGBR is able to achieve an accuracy of 0.418, compared to 0.381 for Rocket.

7.4.3 Radial Drilling Results

	Ours						**Ours**
Dataset	**MCTST**	**CTST**	**TST**	**Rocket**	**GBoost**	**XGBoost**	**XGBR**
DrillingV1	0.706	*0.732*	0.719	0.770	<u>0.744</u>	<u>0.744</u>	**0.820**
DrillingV2	0.732	0.727	*0.750*	<u>0.780</u>	0.737	0.733	**0.801**
DrillingV3	0.745	0.741	*0.764*	<u>0.791</u>	0.749	0.755	**0.819**
DrillingV4	<u>0.755</u>	<u>0.755</u>	0.737	*0.754*	0.751	*0.754*	**0.804**
Mean Accuracy	0.734	0.739	0.743	<u>0.774</u>	0.745	*0.746*	**0.811**

Table 5: Test accuracies using the 4 sampling-methods on the Radial-Drilling dataset. The best result for each dataset is highlighted in **bold**. The second-best result is <u>underlined</u>. The third-best result is printed in *italic*.

For the Radial-Drilling datasets, we can see from Table 5 that the transformer-based approaches (TST, CTST & MCTST) perform better on the datasets in which samples contain data from a longer time ago. Especially DrillingV3 seems to perform, on average, quite well. In this dataset, samples are included from up to an hour ago (compared to 60 seconds for the V1-dataset). This suggests that the transformer-based models make use of the longer-term dependencies in the data. DrillingV2 - which includes samples from the same amount of time before the prediction-point - but includes more data in each sample - seems to perform a bit worse than V3 in all cases, which suggests that a balance between the amount of time-steps further in the past and the amount of more recent time-steps is important for optimal performance of the transformer-based models. We note that all 4 dataset sampling methods perform reasonably well. This can be explained by the fact that the performance for DrillingV1 is already very strong, implying that the last 60 seconds of data contain most of the information needed to predict the current state of the drilling process. As the last 60 seconds are included in all 4 variants, this would explain the good performance across the board.

Although the performance of XGBR is quite robust on the UAE datasets, we can see from Table 4 that the resulting accuracies of the same method on the Radial-Drilling dataset are surprisingly high, outperforming both the Rocket and XGBoost methods individually. Most notably, the performance on the DrillingV1 dataset was the highest, with an accuracy of 82%, indicating that it is sufficient to use only the last 60 seconds of data to accurately predict the current main-state of the drilling process. In second place, we find the Rocket-classification method, which, similar to the UAE-dataset experiments, performs quite well compared to the other methods. XGBoost comes in third, though its average performance is quite close to that of the other remaining methods.

8 Conclusion

In this study, we have designed and implemented a UI-based machine learning pipeline for multivariate time-series classification tasks, which we use on our novel classification task: determining the current main-process step in the Radial-Drilling process based on the previous outputs of the 4 most important sensors. Using our implementation of MVTS-Analyzer - a tool for visualizing, analyzing and labeling multivariate time-series data, the raw dataset was annotated. Using our implementation of the MVTS-Trainer platform, we have fine-tuned several state-of the art methods on 10 public multivariate time series datasets across several hundreds of optimization runs. We have compared the results of our finding to those reported by the authors but were unable to definitively confirm that their proposed TST-model outperformed the other state-of-the-art methods. We furthermore implement and test two novel transformer-based classification architectures, CTST and MCTST, which make use of convolutional layers to perform self attention on local-features in the data. From our results, we find that the CTST-model offers a small overall improvement over the original classification-transformer model. On all but one dataset, MCTST does not seem to offer a significant improvement over the same baseline. We additionally combine the state-of-the-art Rocket-method with XGBoost (XGBR), and conclude that this classifier is quite robust for the public multivariate time-series datasets, and offers a good balance between the performance of Rocket and XGBoost. Using XGBR, we were able to achieve a maximum accuracy of 82% on the novel Radial-Drilling dataset.

9 Discussion & Future work

The hyperparameter-settings play a large part in the test-performance of the models on the UAE-datasets. We found it very difficult to reproduce the results reported by Zerveas et al., because the authors evidently performed quite a high degree of hyperparameter-optimization on each individual dataset, without explicitly mentioning the optimal settings found. The found difference in performance on several on the datasets could be explained with certain hyperparameters that we have missed while optimizing our configurations, though we believe that our search was quite broad.

In our opinion, to get a more accurate indication of the performance on the smaller UAE-datasets, it would be better to use a basic set of hyperparameters that is used across all datasets to evaluate the performance. Alternatively, if hyperparameter tuning is performed, a set (objective) procedure should be included when reporting on these relatively small datasets to ensure reproducibility of the results. We have, for transparency, included the top-10 hyperparameter optimization experiments in the supplementary material (Section A) for each of the 3 transformer models, on each of the 10 UAE-datasets, such that any future research using these models or datasets can quickly reproduce our results.

For the Radial-Drilling dataset, we must reiterate that the amount of distinct well locations in the provided dataset was very limited. As the limited data for the few well-operations that were from different locations, often did not contain all available labels, this means that, although the well-operations in the train and test-set were distinct, it was impossible to create a sizeable test-set where all well locations were also completely distinct from the train-set. To get a better estimate of the real-world performance of the model, we would like to see our models tested on a variety of completely unseen wells at unseen locations such that (geological) well parameters are completely different. We particularly would like to confirm whether the performance of the XGBR-method indeed remains as high as we have found it to be. Additionally, we would suggest data augmentation by introducing noise to the train-data to make better use of available labeled data, as to make the resulting model more robust.

For future research on our proposed CTST and MCTST classification-architectures, we propose applying multiple parallel convolutional layers to the input (in a Rocket- or inception-

module like fashion), to create a more robust model without the need to fine-tune the kernel and step-sizes. We furthermore would like to explore the effect on performance of using varying degrees of dilation on the introduced convolutional layers. This might allow the model to learn to create a more condensed representation of long input sequences, which could be useful for longer time-series data. Lastly, we would like to test our methods on more large datasets.

References

[1] A. Bagnall, J. Lines, A. Bostrom, J. Large, and E. Keogh. "The Great Time Series Classification Bake Off: a Review and Experimental Evaluation of Recent Algorithmic Advances". In: *Data Mining and Knowledge Discovery* 31 (3 2017), pp. 606–660.

[2] A. Bagnall, Jason Lines, Jon Hills, and Aaron George Bostrom. "Time-Series Classification with COTE: The Collective of Transformation-Based Ensembles". In: *IEEE Transactions on Knowledge and Data Engineering* 27 (2015), pp. 2522–2535.

[3] Anthony J. Bagnall, Hoang Anh Dau, Jason Lines, Michael Flynn, James Large, Aaron Bostrom, Paul Southam, and Eamonn J. Keogh. "The UEA multivariate time series classification archive, 2018". In: *CoRR* abs/1811.00075 (2018). arXiv: 1811.00075. URL: http://arxiv.org/abs/1811.00075.

[4] Y. Bengio, P. Frasconi, and P. Simard. "The problem of learning long-term dependencies in recurrent networks". In: *IEEE International Conference on Neural Networks*. 1993, 1183–1188 vol.3. DOI: 10.1109/ICNN.1993.298725.

[5] Lukas Biewald. *Experiment Tracking with Weights and Biases*. Software available from wandb.com. 2020. URL: https://www.wandb.com/.

[6] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 785–794. ISBN: 9781450342322. DOI: 10.1145/2939672.2939785. URL: https://doi.org/10.1145/2939672.2939785.

[7] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2014. DOI: 10.3115/v1/d14-1179. URL: https://doi.org/10.3115%2Fv1%2Fd14-1179.

[8] F. Chollet. *Deep Learning with Python*. Manning, 2017. ISBN: 9781638352044. URL: https://books.google.nl/books?id=wzozEAAAQBAJ.

[9] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. 2014. arXiv: 1412.3555 [cs.NE].

[10] Razvan-Gabriel Cirstea, Darius-Valer Micu, Gabriel-Marcel Muresan, Chenjuan Guo, and Bin Yang. "Correlated Time Series Forecasting Using Multi-Task Deep Neural Networks". In: *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. CIKM '18. Torino, Italy: Association for Computing Machinery, 2018, pp. 1527–1530. ISBN: 9781450360142. DOI: 10.1145/3269206.3269310. URL: https://doi.org/10.1145/3269206.3269310.

[11] Angus Dempster, François Petitjean, and Geoffrey I. Webb. *ROCKET: Exceptionally fast and accurate time series classification using random convolutional kernels*. cite arxiv:1910.13051Comment: 27 pages, 23 figures. 2019. URL: http://arxiv.org/abs/1910.13051.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423.

[13] Maha Elbayad, Laurent Besacier, and Jakob Verbeek. "Pervasive Attention: 2D Convolutional Neural Networks for Sequence-to-Sequence Prediction". In: *Proceedings of the 22nd Conference on Computational Natural Language Learning*. Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 97–107. DOI: 10.18653/v1/K18-1010. URL: https://aclanthology.org/K18-1010.

[14] Hassan Ismail Fawaz, Benjamin Lucas, Germain Forestier, Charlotte Pelletier, Daniel F. Schmidt, Jonathan Weber, Geoffrey I. Webb, Lhassane Idoumghar, Pierre-Alain Muller, and François Petitjean. "InceptionTime: Finding AlexNet for time series classification". In: *Data Mining and Knowledge Discovery* 34.6 (Sept. 2020), pp. 1936–1962. DOI: 10.1007/s10618-020-00710-y. URL: https://doi.org/10.1007%2Fs10618-020-00710-y.

[15] Andrejs Fedjajevs, Willemijn Groenendaal, Carlos Agell, and Evelien Hermeling. "Platform for Analysis and Labeling of Medical Time Series". In: *Sensors* 20 (Dec. 2020), p. 7302. DOI: 10.3390/s20247302.

[16] Jean-Yves Franceschi, Aymeric Dieuleveut, and Martin Jaggi. "Unsupervised Scalable Representation Learning for Multivariate Time Series". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/53c6de78244e9f528eb3e1cda69699bb-Paper.pdf.

[17] Jerome H. Friedman. "Greedy Function Approximation: A Gradient Boosting Machine". In: *Annals of Statistics* 29 (2000), pp. 1189–1232.

[18] Jerome H. Friedman. "Stochastic gradient boosting". In: *Computational Statistics & Data Analysis* 38.4 (2002). Nonlinear Methods and Data Mining, pp. 367–378. ISSN: 0167-9473. DOI: https://doi.org/10.1016/S0167-9473(01)00065-2. URL: https://www.sciencedirect.com/science/article/pii/S0167947301000652.

[19] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. "Learning to Forget: Continual Prediction with LSTM". In: *Neural Computation* 12 (1999), pp. 2451–2471.

[20] Ruichang Guo, Gensheng Li, Zhongwei Huang, Shouceng Tian, Xiaoning Zhang, and Wei Wu. "Theoretical and experimental study of the pulling force of jet bits in radial drilling technology". In: *Petroleum Science* 6.4 (Nov. 2009), pp. 395–399. DOI: 10.1007/s12182-009-0060-6. URL: https://doi.org/10.1007%2Fs12182-009-0060-6.

[21] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162%2Fneco.1997.9.8.1735.

[22] Ahmad Kh. Al-Jasmi, Ali Alsabee, Ahmad Al-Awad, Adel Attia, Abdou Elsayed, and Ahmed El-Mougy. "Improving Well Productivity in North Kuwait Well by Optimizing Radial Drilling Procedures". In: *Day 2 Thu, February 08, 2018*. SPE, Feb. 2018. DOI: 10.2118/189516-ms. URL: https://doi.org/10.2118%2F189516-ms.

[23] Tung Kieu, Bin Yang, Chenjuan Guo, and Christian S. Jensen. "Outlier Detection for Time Series with Recurrent Autoencoder Ensembles". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, July 2019, pp. 2725–2732. DOI: 10.24963/ijcai.2019/378. URL: https://doi.org/10.24963/ijcai.2019/378.

[24] Aleksandr Kochnev, Sergey Galkin, Sergey Krivoshchekov, Nikita Kozyrev, and Polina Chalova. "Application of Machine Learning Algorithms to Predict the Effectiveness of Radial Jet Drilling Technology in Various Geological Conditions". In: *Applied Sciences* 11.10 (May 2021), p. 4487. DOI: 10.3390/app11104487. URL: https://doi.org/10.3390%2Fapp11104487.

[25] John F. Kolen and Stefan C. Kremer. "Gradient Flow in Recurrent Nets: The Difficulty of Learning Long Term Dependencies". In: *A Field Guide to Dynamical Recurrent Networks*. 2001, pp. 237–243. DOI: 10.1109/9780470544037.ch14.

[26] James Large, E. Kemsley, Nikolaus Wellner, Ian Goodall, and Anthony Bagnall. "Detecting Forged Alcohol Non-invasively Through Vibrational Spectroscopy and Machine Learning". In: June 2018, pp. 298–309. ISBN: 978-3-319-93033-6. DOI: 10.1007/978-3-319-93034-3_24.

[27] Shiyang Li, Xiaoyong Jin, Yao Xuan, Xiyou Zhou, Wenhu Chen, Yu-Xiang Wang, and Xifeng Yan. "Enhancing the Locality and Breaking the Memory Bottleneck of Transformer on Time Series Forecasting". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[28] Jason Lines, Sarah Taylor, and Anthony Bagnall. "HIVE-COTE: The Hierarchical Vote Collective of Transformation-Based Ensembles for Time Series Classification". In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. 2016, pp. 1041–1046. DOI: 10.1109/ICDM.2016.0133.

[29] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[30] *Radial Drilling Technique for Improving Well Productivity in Petrobel-Egypt*. Vol. All Days. SPE North Africa Technical Conference and Exhibition. Apr. 2013, SPE-164773–MS. DOI: 10.2118/164773-MS. eprint: https://onepetro.org/SPENATC/proceedings-pdf/13NATC/All-13NATC/SPE-164773-MS/1587091/spe-164773-ms.pdf. URL: https://doi.org/10.2118/164773-MS.

[31] Gabriel Sampedro, Made Adi Paramartha Putra, and Mideth Abisado. "3D-AmplifAI: An Ensemble Machine Learning Approach to Digital Twin Fault Monitoring for Additive Manufacturing in Smart Factories". In: *IEEE Access* PP (Jan. 2023), pp. 1–1. DOI: 10.1109/ACCESS.2023.3289536.

[32] Patrick Schäfer and Mikael Högqvist. "SFA: A symbolic fourier approximation and index for similarity search in high dimensional datasets". In: Mar. 2012, pp. 516–527. DOI: 10.1145/2247596.2247656.

[33] Guizhu Shen, Qingping Tan, Haoyu Zhang, Ping Zeng, and Jianjun Xu. "Deep Learning with Gated Recurrent Unit Networks for Financial Sequence Predictions". In: *Procedia Computer Science* 131 (2018), pp. 895–903. DOI: 10.1016/j.procs.2018.04.298. URL: https://doi.org/10.1016%2Fj.procs.2018.04.298.

[34] Li Shen and Yangzhu Wang. "TCCT: Tightly-coupled convolutional transformer on time series forecasting". In: *Neurocomputing* 480 (2022), pp. 131–145. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2022.01.039. URL: https://www.sciencedirect.com/science/article/pii/S0925231222000571.

[35] Chang Wei Tan, Christoph Bergmeir, Francois Petitjean, and Geoffrey I Webb. "Time Series Extrinsic Regression". In: *Data Mining and Knowledge Discovery* (2021), pp. 1–29. DOI: https://doi.org/10.1007/s10618-021-00745-9.

[36] Wensi Tang, Lu Liu, and Guodong Long. "Few-shot Time-series Classification with Dual Interpretability". In: 2019.

[37] Wensi Tang, Lu Liu, and Guodong Long. "Interpretable Time-series Classification on Few-shot Samples". In: *CoRR* abs/2006.02031 (2020). arXiv: 2006.02031. URL: https://arxiv.org/abs/2006.02031.

[38] Maxim Tkachenko, Mikhail Malyuk, Andrey Holmanyuk, and Nikolai Liubimov. *Label Studio: Data labeling software*. Open source software available from https://github.com/heartexlabs/label-studio. 2020-2022. URL: https://github.com/heartexlabs/label-studio.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[40] Qingsong Wen, Tian Zhou, Chaoli Zhang, Weiqi Chen, Ziqing Ma, Junchi Yan, and Liang Sun. *Transformers in Time Series: A Survey*. 2023. arXiv: 2202.07125 [cs.LG].

[41] George Zerveas, Srideepika Jayaraman, Dhaval Patel, Anuradha Bhamidipaty, and Carsten Eickhoff. "A Transformer-Based Framework for Multivariate Time Series Representation Learning". In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. KDD '21. Virtual Event, Singapore: Association for Computing Machinery, 2021, 21142124. ISBN: 9781450383325. DOI: 10.1145/3447548.3467401. URL: https://doi.org/10.1145/3447548.3467401.

[42] Xuchao Zhang, Yifeng Gao, Jessica Lin, and Chang-Tien Lu. "TapNet: Multivariate Time Series Classification with Attentional Prototypical Network". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.04 (Apr. 2020), pp. 6845–6852. DOI: 10.1609/aaai.v34i04.6165. URL: https://ojs.aaai.org/index.php/AAAI/article/view/6165.

[43] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. *Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting*. 2021. arXiv: 2012.07436 [cs.LG].

A Supplementary Material

Standard Deviations

Table 6 denotes the standard deviation of the test-accuracy for each of the methods, for each of the UAE-datasets. All methods were tested 5-10 times according to the procedure described in Section 7.3.

Dataset	MCTST	CTST	TST	Rocket	GBoost	XGB	XGBR
EthanolConcentration	0.027	0.024	0.012	0.003	0.011	0.000	0.022
FaceDetection	0.002	0.005	0.006	0.005	0.004	0.000	0.006
Handwriting	0.016	0.008	0.008	0.003	0.005	0.000	0.016
Heartbeat	0.078	0.010	0.017	0.010	0.009	0.000	0.023
JapaneseVowels	0.024	0.006	0.012	0.001	0.006	0.000	0.009
PEMS-SF	0.042	0.051	0.011	0.016	0.007	0.000	0.028
SelfRegulationSCP1	0.035	0.007	0.017	0.007	0.008	0.000	0.036
SelfRegulationSCP2	0.014	0.032	0.013	0.028	0.012	0.000	0.036
SpokenArabicDigits	0.004	0.006	0.002	0.001	0.001	0.000	0.003
UWaveGestureLibrary	0.011	0.003	0.011	0.002	0.005	0.000	0.021

Table 6: The standard deviations of the reported accuracies of each of the methods on the UAE-datasets.

For the radial drilling datasets, all classifiers were trained from scratch at least 3 times, according to the methods described in Section 7.3. Table 7 denotes the standard deviation of the found test-accuracies of each of the methods, on each of the dataset-variants.

	MCTST	CTST	TST	Rocket	Gboost	XGB	XGBR
DrillingV1	0.034	0.023	0.014	0.003	0.004	0.008	0.027
DrillingV2	0.037	0.019	0.030	0.011	0.012	0.006	0.003
DrillingV3	0.031	0.024	0.020	0.009	0.022	0.019	0.019
DrillingV4	0.016	0.015	0.030	0.012	0.006	0.012	0.002

Table 7: The standard deviations of the reported accuracies of each of the methods on the radial-drilling datasets.

Hyperparameter optimization

The following (sub-)sections contain the top-10 best performing hyperparameter optimization runs on the UAE-datasets, for the original TST models, CTST-models, and MCTST-models. Any settings in the names of the configurations are the deviations from the author-provided hyperparameters and/or the default values of the parameters in question, as shown in table 8. *bs* denotes the used batch-size, *ss* corresponds to the subsampling-rate. For CTST and MCTST, the hyperparameter setting names start with: `conv<1>_<2>x<3>`, where:

- **<1>** denotes the amount of convolutional layers used in the model.
- **<2>** denotes the kernel sizes for each layer
- **<3>** denotes the step sizes for each layer

E.g. `conv2_4+2x2cstep_bs32_16model_ss2` describes the following hyperparameters: an MCTST-model with 2 convolutional layers with a kernel size of 4 and 2 respectively and a

step size of 2 for both, `d_model` is 16. The model is trained using a batch-size of 32, and a sub-sampling rate of 2 is applied to the input.

Dataset	**n_blocks**	**n_heads**	**d_model**	**d_FFW**
EthanolConcentration	1	8	64	256
FaceDetection	3	8	128	256
Handwriting	1	8	128	256
Heartbeat	1	8	64	256
JapaneseVowels	3	8	128	256
PEMS-SF	1	8	128	512
SelfRegulationSCP1	3	8	128	256
SelfRegulationSCP2	3	8	128	256
SpokenArabicDigits	3	8	64	256
UWaveGestureLibrary	3	16	256	256

Table 8: Default author-provided hyperparameter settings for the TST-model.

The tables are split per architecture (TST, CTST, MCTST) and per dataset. The high-score is the maximum accuracy achieved by the model on the validation set. To take stability of the tested model into account, we also average the accuracy of the previous and next 10 epochs (denoted by *10-accuracy* and *10-rank*), as mentioned in Section 7.3. The best performing hyperparameter-settings are shown in **bold**, these are the hyperparameters that were tested on the test-set.

Hyperparameter Optimization Runs - TST, CTST & MCTST

Table 9: TST - EthanolConcentration

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
TST_2ss_bs32_5fold	**106.0**	**0.374**	**1**	**0.328**	**2**	**1.5**
TST_bs16	146.0	0.358	2	0.326	3	2.5
TST_0,0001lr_2ss_bs32_5fold	344.0	0.343	7	0.331	1	4.0
TST_relu_bs16	218.0	0.351	3	0.320	5	4.0
TST_4ss_bs45_5fold	190.0	0.347	4	0.310	6	5.0
TST_128ffw_32model_4ss_bs45_5fold	306.0	0.343	7	0.322	4	5.5
TST_128ffw_4ss_bs45_5fold	146.0	0.347	4	0.300	9	6.5
TST_128dim_4ss_bs45_5fold	58.0	0.343	7	0.303	8	7.5
TST_3layer_8ss_bs128	42.0	0.343	9	0.306	7	8.0
TST_4ss_bs64_5fold	134.0	0.340	10	0.300	9	9.5

Table 10: TST - FaceDetection

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
TST_bs512	**232.0**	**0.783**	**1**	**0.772**	**1**	**1.0**
TST_fixedencoding_bs512	112.0	0.768	4	0.763	3	3.5
TST_bs256_64model_128ffw	292.0	0.767	6	0.763	2	4.0
TST_bs64_64model_fixedencoding	72.0	0.770	2	0.762	7	4.5
TST_bs128	28.0	0.768	4	0.760	8	6.0
TST_bs512	66.0	0.767	7	0.762	5	6.0
TST_bs96	22.0	0.768	3	0.759	9	6.0
TST_bs256	46.0	0.767	9	0.762	4	6.5
TST_bs64	350.0	0.767	8	0.762	6	7.0
TST_bs256_64model	248.0	0.762	10	0.756	10	10.0

Table 11: TST - Handwriting

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
TST_bs64_ss2	**116.0**	**0.287**	**1**	**0.283**	**2**	**1.5**
TST_bs64_16seed1	216.0	0.283	3	0.283	2	2.5
TST_bs64_3layer	66.0	0.283	3	0.283	2	2.5
TST_bs128_ss2	72.0	0.283	3	0.276	5	4.0
TST_bs32	68.0	0.283	3	0.276	5	4.0
TST_bs64	286.0	0.277	6	0.277	4	5.0
TST_bs128	560.0	0.277	6	0.274	7	6.5
TST_fixedencoding_bs64_ss2	346.0	0.270	8	0.270	8	8.0
TST_0.25drop_bs64_ss2	348.0	0.270	8	0.265	9	8.5
TST_adam_0.5drop_bs64_ss2	186.0	0.267	10	0.244	11	10.5

Table 12: TST - Heartbeat

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
TST_bs64_400epoch_128feedf_4heads_5fold	**144.0**	**0.795**	**1**	**0.786**	**1**	**1.0**
TST_fixedencoding_bs64_400epoch_128feedf_4heads_5fold	30.0	0.795	1	0.778	2	1.5
TST_bs64_400epoch_128feedf_5fold	86.0	0.785	4	0.768	3	3.5
TST_bs64_400epoch_4heads_5fold	82.0	0.785	4	0.767	4	4.0
TST_bs64_llr0_0001_800epoch_5fold	336.0	0.780	7	0.767	4	5.5
TST_b64_400epoch_5fold	126.0	0.780	7	0.765	6	6.5
TST_bs64_127400epoch_5fold	12.0	0.785	4	0.757	9	6.5
TST_bs64_400epoch_5fold	126.0	0.780	7	0.765	6	6.5
TST_bs128_400epoch_5fold_ss2	46.0	0.766	10	0.759	8	9.0
TST_bs48	120.0	0.771	9	0.747	11	10.0

Table 13: TST - JapaneseVowels

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
TST_bs64_128moded2subs_5fold	**476.0**	**0.996**	**1**	**0.995**	**1**	**1.0**
TST_bs64_2subs_5fold	140.0	0.993	3	0.990	2	2.5
TST_bs64_16heads_5fold	248.0	0.993	3	0.989	3	3.0
TST_bs64_5fold	284.0	0.993	3	0.989	4	3.5
TST_bs128_16head_5fold	354.0	0.989	6	0.989	4	5.0
TST_64model_128_ffw_bs64_2subs_5fold	218.0	0.991	5	0.986	6	5.5
TST_fixedencoding_bs64_2subs_5fold	234.0	0.989	6	0.984	7	6.5
TST_bs128_5fold	386.0	0.985	8	0.984	7	7.5
TST_64_model_128_ffw_bs64_5fold	496.0	0.983	10	0.981	9	9.5
TSTbs64_16heads_2ss_5fold	32.0	0.985	8	0.979	11	9.5

Table 14: TST - PEMS-SF

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
TST_bs64_3layers_16heads_256ff_5fold_1.5kepoch	**1264.0**	**0.867**	**2**	**0.857**	**1**	**1.5**
TST_bs64_3layers_5fold_1500epoch	796.0	0.874	1	0.824	4	2.5
TST_fixedencoding_bs64_3layers_16heads_256ff_5fold_1.5kepoch	1384.0	0.859	3	0.840	2	2.5
TST_bs64_3layers_16heads_256ff_5fold	412.0	0.844	4	0.829	3	3.5
TST_bs64_2ss_5fold_1.5kepoch	1434.0	0.830	5	0.813	5	5.0
TST_bs64_12785fold	18.0	0.822	6	0.799	7	6.5
TST_bs64_5fold1.5kepoch	18.0	0.822	6	0.799	7	6.5
TST_bs128_5fold_1.5kepoch	30.0	0.815	9	0.802	6	7.5
TST_bs32_5fold	12.0	0.819	8	0.767	10	9.0
TST_bs64_2ss_5fold	60.0	0.793	10	0.770	9	9.5

Table 15: TST - SelfRegulationSCP1

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
TST_64model_layernorm_128bs_2ss	**56.0**	**0.867**	**1**	**0.854**	**1**	**1.0**
TST_64model_128ffw_layernorm_128bs_2ss	54.0	0.848	2	0.837	2	2.0
TST_layernorm_128bs_2ss	50.0	0.848	2	0.836	3	2.5
TST_64bs_4ss	84.0	0.844	4	0.834	4	4.0
TST_64model_128ffw_128bs_2ss	42.0	0.844	4	0.831	6	5.0
TST_128bs_4ss	170.0	0.837	9	0.834	4	6.5
TST_32bs_2ss	190.0	0.841	7	0.831	6	6.5
TST_32bs	30.0	0.841	6	0.825	8	7.0
TST_128bs_2ss	10.0	0.841	7	0.760	10	8.5
TST_32bs_3ss	8.0	0.833	10	0.794	9	9.5

Table 16: TST - SelfRegulationSCP2

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
TST_64model_16batch_2ss	**104.0**	**0.555**	**5**	**0.542**	**1**	**3.0**
TST_32bs_2ss_0,3dropout_1layer	102.0	0.560	1	0.514	7	4.0
TST_64model_64batch_128ffw_1layer_4ss_4heads	32.0	0.555	5	0.531	3	4.0
TST_32bs_2ss_0,4dropout	56.0	0.560	1	0.514	8	4.5
TST_bs64_3ss_5fold	170.0	0.555	3	0.519	6	4.5
TST_bs64_5fold	92.0	0.550	8	0.532	2	5.0
TST_64model_64batch_4ss	38.0	0.535	9	0.527	4	6.5
TST_64model_64batch_128ffw_1layer_4ss	74.0	0.535	9	0.523	5	7.0
TST_bs64_2ss_5fold	24.0	0.555	5	0.512	9	7.0
TST_bs64_2ss_5fold	18.0	0.555	5	0.507	10	7.5

Table 17: TST - SpokenArabicDigits

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
TST_bs256_2ss	**124.0**	**0.997**	**2**	**0.996**	**1**	**1.5**
TST_bs384	350.0	0.996	6	0.996	2	4.0
TST_fixedencoding_bs256_2ss	140.0	0.997	2	0.996	6	4.0
TST_bs256_128model_6layer	108.0	0.997	2	0.995	7	4.5
TST_bs256_128model	246.0	0.996	6	0.996	3	4.5
TST_bs256	92.0	0.996	6	0.996	4	5.0
TST_bs512	148.0	0.996	9	0.996	5	7.0
TST_bs256_128ffw	86.0	0.997	4	0.994	12	8.0
TST_bs256_3ss	144.0	0.996	9	0.995	8	8.5
TST_bs256_128ffw_4heads	138.0	0.996	11	0.995	9	10.0

Table 18: TST - UWaveGestureLibrary

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
TST_bs64_ss2	**154.0**	**0.904**	**1**	**0.904**	**1**	**1.0**
TST_bs48_ss2	600.0	0.900	2	0.900	2	2.0
TST_bs128	40.0	0.896	5	0.896	4	4.5
TST_bs48	42.0	0.896	5	0.896	4	4.5
TST_bs64	582.0	0.896	5	0.896	4	4.5
TST_fixedencoding_bs64	192.0	0.896	5	0.896	4	4.5
TST_bs32_ss2	60.0	0.896	5	0.894	7	6.0
TST_bs32	10.0	0.896	5	0.810	10	7.5
TST_bs64_512ffw	500.0	0.892	9	0.890	8	8.5
TST_bs128_512ffw	596.0	0.887	10	0.888	9	9.5

Table 19: CTST - EthanolConcentration

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv1_128x1cstep_4ss_bs45	**716.0**	**0.453**	**1**	**0.414**	**1**	**1.0**
conv1_128x4cstep_bs45_128model	222.0	0.415	3	0.379	3	3.0
conv1_128x4cstep_bs45	743.0	0.408	4	0.387	2	3.0
conv1_8x1cstep_4ss_bs45	304.0	0.396	6	0.372	4	5.0
conv1_256x4cstep_bs45_128model	376.0	0.400	5	0.370	5	5.0
conv1_256x4cstep_bs45	562.0	0.421	2	0.357	10	6.0
conv1_256x4cstep_bs45	343.0	0.396	6	0.357	9	7.5
conv1_8x1cstep_4ss_bs64	292.0	0.392	9	0.364	6	7.5
conv1_256x4cstep_bs20	471.0	0.392	9	0.358	7	8.0
conv1_8x1cstep_128ffw_4ss_bs45	300.0	0.385	12	0.358	8	10.0

Table 20: CTST - FaceDetection

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv1_8x1cstep_64dmodel_bs512	**46.0**	**0.784**	**1**	**0.775**	**1**	**1.0**
conv1_8x1cstep_fixed_bs512	54.0	0.776	3	0.771	2	2.5
conv1_16x1cstep_bs512_fixed	28.0	0.774	4	0.770	3	3.5
conv1_16x1cstep_fixed_bs512	78.0	0.778	2	0.767	7	4.5
conv1_32x1cstep_fixed_bs512fixed_	30.0	0.771	7	0.769	4	5.5
conv1_32x1cstep_fixed_bs512	30.0	0.771	7	0.769	4	5.5
conv1_4x1cstep_bs512_fixed	76.0	0.771	6	0.768	6	6.0
conv1_32x1cstep_bs512_128model	20.0	0.774	5	0.766	8	6.5
conv1_8x1cstep_bs512_128model	52.0	0.769	9	0.764	9	9.0
conv1_8x1cstep_64dmodel_bs512	332.0	0.768	10	0.761	11	10.5

Table 21: CTST - Handwriting

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv1_16x1clayer_bs128	**92.0**	**0.327**	**1**	**0.327**	**1**	**1.0**
conv1_16x1clayer_bs128_32model_128ffw	268.0	0.327	1	0.327	1	1.0
conv1_8x1clayer_bs64	28.0	0.320	3	0.318	3	3.0
conv1_32x1clayer_bs128	216.0	0.313	4	0.313	4	4.0
conv1_64x1clayer_bs128	106.0	0.307	5	0.307	5	5.0
conv1_16x1clayer_bs128_32model	80.0	0.307	5	0.298	6	5.5
conv1_16x1clayer_bs128_64model	296.0	0.293	7	0.293	7	7.0
conv1_16x1clayer_bs128_32model_128ffw_4heads_2layers	164.0	0.293	7	0.293	8	7.5
conv1_16x1clayer_bs64_ss2	48.0	0.280	9	0.280	9	9.0
conv1_16x1clayer_bs64_ss2	48.0	0.280	9	0.280	9	9.0

Table 22: CTST - Heartbeat

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv1_8x2cstep_bs128_ss2	**388.0**	**0.795**	**1**	**0.773**	**1**	**1.0**
conv1_8x2cstep_bs128_64ffw	176.0	0.790	2	0.773	2	2.0
conv1_4x1cstep_bs128_32model_32ffw	54.0	0.780	4	0.767	3	3.5
conv1_8x2clayer_bs64_64model_128ffw_1layer	286.0	0.785	3	0.766	4	3.5
conv1_8x4cstep_bs128	152.0	0.780	4	0.761	6	5.0
conv1_16x1cstep_bs96	362.0	0.766	8	0.763	5	6.5
conv1_8x1cstep_bs128_32model_32ffw	74.0	0.771	6	0.751	7	6.5
conv1_8x1clayer_bs64_64model_128ffw_1layer	72.0	0.766	8	0.747	8	8.0
conv1_4x1cstep_bs64_ss2	72.0	0.766	8	0.740	9	8.5
conv1_8x2cstep_bs128	132.0	0.761	10	0.739	10	10.0

Table 23: CTST - JapaneseVowels

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv1_8x1cstep_fixedpos_64_model_bs64_2subs	**450.0**	**0.993**	**1**	**0.991**	**1**	**1.0**
conv1_4x1cstep_fixedpos_64_model_bs64_2subs	202.0	0.991	2	0.987	3	2.5
conv1_8x1cstep_fixedpos_64_model_128_ffw_bs64_2subs	260.0	0.989	4	0.987	2	3.0
conv1_4x2cstep_64_model_bs64	246.0	0.991	3	0.983	4	3.5
conv1_16x2cstep_128model_bs64	220.0	0.980	6	0.978	5	5.5
conv1_4x2cstep_64_model_128ffw_bs64	212.0	0.985	5	0.976	7	6.0
conv1_2x4cstep_64model_bs256	404.0	0.978	7	0.978	6	6.5
conv1_16x2cstep_64_model_256_ffw_bs64	128.0	0.974	8	0.971	8	8.0
conv1_16x4cstep_64model_bs128	464.0	0.970	9	0.970	9	9.0
conv1_16x2cstep_64_model_bs64	292.0	0.969	10	0.966	10	10.0

Table 24: CTST - PEMS-SF

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv1_16x1cstep_256ffw_16heads_3layer_64bs	**426.0**	**0.867**	**1**	**0.834**	**1**	**1.0**
conv1_16x1cstep_256ffw_16heads_3layer_128bs	260.0	0.852	2	0.813	3	2.5
conv1_4x2cstep_256ffw_16heads_3layer_64bs	280.0	0.844	3	0.817	2	2.5
conv1_8x1cstep_bs32	1432.0	0.826	4	0.812	4	4.0
conv1_16x1cstep_default_64bs	18.0	0.815	5	0.791	7	6.0
conv1_2x1cstep_64model_64bs	30.0	0.811	8	0.798	5	6.5
conv1_64x1cstep_default_64bs	18.0	0.815	5	0.790	9	7.0
conv1_4x1cstep_default_64bs	22.0	0.811	8	0.791	8	8.0
conv1_2x1cstep_default_64bs	386.0	0.796	11	0.795	6	8.5
conv1_4x2cstep_64model_128ffw_64bs	26.0	0.811	8	0.773	11	9.5

Table 25: CTST - SelfRegulationSCP1

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv1_8x1cstep_layernorm_128bs_2ss	**94.0**	**0.844**	**2**	**0.844**	**1**	**1.5**
conv1_32x4cstep_bs64	144.0	0.848	1	0.837	3	2.0
conv1_8x1cstep_64model_128ffw_layernorm_128bs_2ss	120.0	0.841	5	0.841	2	3.5
conv1_8x1cstep_32model_layernorm_16bs	204.0	0.844	2	0.824	7	4.5
conv1_8x1cstep_64model_layernorm_64bs_2ss	496.0	0.837	7	0.836	4	5.5
conv1_8x2cstep_256model_layernorm_64bs	40.0	0.841	5	0.825	6	5.5
conv1_8x4cstep_64	98.0	0.833	10	0.828	5	7.5
conv1_8x2cstep_64model_layernorm_128bs	42.0	0.837	7	0.819	10	8.5
conv1_32+16x2cstep_128+64cchannel_bs45	10.0	0.841	5	0.788	13	9.0
conv1_4x4cstep_bs64	128.0	0.830	12	0.821	8	10.0

Table 26: CTST - SelfRegulationSCP2

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv1_4x2cstep_bs27	**296.0**	**0.560**	**3**	**0.547**	**1**	**2.0**
conv1_4x2cstep_bs32_512model	76.0	0.565	1	0.526	6	3.5
conv1_4x2cstep_bs32	148.0	0.550	6	0.541	2	4.0
conv1_4x1cstep_bs32	78.0	0.555	5	0.533	4	4.5
conv1_6x3cstep_bs64	204.0	0.545	8	0.540	3	5.5
conv1_8x1cstep_bs27_1024model	160.0	0.560	3	0.516	10	6.5
conv1_4x2cstep_bs32_1024model	128.0	0.560	3	0.503	13	8.0
conv1_32+16x1cstep_4ss_bs45	40.0	0.540	10	0.521	7	8.5
conv1_8x1cstep_4xbs27_16model	130.0	0.535	12	0.528	5	8.5
conv1_32x1cstep_bs27	0.0	0.545	8	0.509	12	10.0

Table 27: CTST - SpokenArabicDigits

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv1_8x2cstep_fixedencoding_bs256	**238.0**	**0.998**	**2**	**0.998**	**1**	**1.5**
conv1_24x4cstep_bs128	58.0	0.998	1	0.997	3	2.0
conv1_16x1cstep_32model_bs256	190.0	0.998	3	0.998	2	2.5
conv1_8x1cstep_fixedencoding_bs256_2ss	248.0	0.998	4	0.997	5	4.5
conv1_24x4cstep_bs128_32model	248.0	0.998	5	0.997	6	5.5
conv1_8x1cstep_bs256	230.0	0.997	7	0.997	4	5.5
conv1_2x2cstep_bs128	100.0	0.997	6	0.995	10	8.0
conv1_8x1cstep_fixedencoding_bs128_2ss	242.0	0.997	8	0.996	8	8.0
conv1_24x2cstep_bs128	50.0	0.997	10	0.997	7	8.5
conv1_4x2cstep_bs128	132.0	0.997	8	0.995	11	9.5

Table 28: CTST - UWaveGestureLibrary

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv1_4x1cstep_bs64	**578.0**	**0.958**	**1**	**0.958**	**1**	**1.0**
conv1_8x1cstep_3ss_bs64	594.0	0.933	2	0.928	2	2.0
conv1_8x2cstep_2ss_bs64	466.0	0.933	2	0.925	3	2.5
conv1_8x2cstep_bs64	284.0	0.925	4	0.925	3	3.5
conv1_16x2cstep_bs64	312.0	0.908	6	0.908	5	5.5
conv1_8x1cstep_bs64_128model	450.0	0.908	6	0.908	5	5.5
conv1_16x1cstep_bs64	436.0	0.896	8	0.896	7	7.5
conv1_64x1cstep_bs64	6.0	0.917	5	0.696	11	8.0
conv1_32x1cstep_bs64	594.0	0.892	9	0.892	8	8.5
conv1_4x1cstep_fixedpos_bs64	10.0	0.885	10	0.735	10	10.0

Table 29: MCTST - EthanolConcentration

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv2_64+16x1cstep_4ss_bs45	**464.0**	**0.468**	**1**	**0.443**	**1**	**1.0**
conv2_32+16x1cstep_4ss_bs45	538.0	0.419	3	0.408	2	2.5
conv3_64+32+16x1cstep_4ss_bs45	396.0	0.442	2	0.404	3	2.5
conv2_128+16x1cstep_4ss_bs45	436.0	0.404	4	0.360	6	5.0
conv2_16+8x1cstep_4ss_bs45	296.0	0.396	7	0.368	4	5.5
conv2_16+8x1cstep_4ss_bs45	626.0	0.404	4	0.353	7	5.5
conv2_64+16x4+2cstep_bs45	784.0	0.400	6	0.367	5	5.5
conv2_128+32x4cstep_128+64cchannel_bs45_ss2	750.0	0.396	7	0.346	8	7.5
conv2_128+32x4+1cstep_128+32cchannel_bs64	719.0	0.385	9	0.331	9	9.0
conv2_128+32x4cstep_128+64cchannel_bs45	252.0	0.384	10	0.329	11	10.5

Table 30: MCTST - FaceDetection

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv2_clayer_16x1cstep_fixed_bs512	**18.0**	**0.776**	**1**	**0.769**	**1**	**1.0**
conv2_clayer_32+16x1cstep_fixed_bs1024	30.0	0.772	2	0.767	2	2.0
conv2_clayer_32+16x1cstep_fixed_bs512	34.0	0.766	4	0.763	3	3.5
conv2_8x1cstep_bs512	24.0	0.766	4	0.761	5	4.5
conv2_8x1cstep_64dmodel_bs512	84.0	0.769	3	0.760	7	5.0
conv3_clayer_16x1cstep_fixed_bs512	16.0	0.766	6	0.762	4	5.0
conv2_8x1cstep_256+128cchannel_bs512	24.0	0.766	7	0.760	6	6.5
conv3_clayer_32+16+8x1cstep_fixed_bs512	16.0	0.764	8	0.759	8	8.0
conv2_32x1cstep_bs512	200.0	0.762	9	0.754	9	9.0
conv3_8+8+1x1cstep_256+128+128cchannel_bs512_2ss	18.0	0.745	10	0.740	10	10.0

Table 31: MCTST - Handwriting

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv2_32+4x1clayer_bs128_32model_128ffw_3layer	**254.0**	**0.333**	**1**	**0.333**	**1**	**1.0**
conv2_32+4x1clayer_128x64cchannel_bs128_64model	136.0	0.300	3	0.300	2	2.5
conv2_32+4x1clayer_bs128_64model	136.0	0.300	3	0.300	2	2.5
conv2_16+2x1clayer_bs128_64model	66.0	0.300	3	0.295	4	3.5
conv2_4x2+1cstep_64model_128ffw_3blocks_64	72.0	0.293	6	0.291	5	5.5
conv2_16x1clayer_bs64_fixed_3layer	256.0	0.293	6	0.288	6	6.0
conv2_32+4x1clayer_128x32cchannel_bs128_32model	88.0	0.287	8	0.282	8	8.0
conv3_4x1clayer_bs64	30.0	0.293	6	0.276	10	8.0
conv2_16x1clayer_bs64	34.0	0.287	10	0.284	7	8.5
conv2_32+4x1clayer_bs128_32model_128ffw_16heads_6layer	104.0	0.287	8	0.278	9	8.5

Table 32: MCTST - Heartbeat

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv2_32x2cstep_bs64_64model_3layer_8heads_256ffw	**382.0**	**0.795**	**1**	**0.769**	**1**	**1.0**
conv2_16x2cstep_bs64_128model_1layer_4heads_128ffw	148.0	0.780	3	0.767	2	2.5
conv2_8x2cstep_bs64_128model_1layer_4heads_128ffw	212.0	0.785	2	0.765	3	2.5
conv2_clayer_8x2cstep_bs64_128model_2layer	118.0	0.776	4	0.754	4	4.0
conv3_clayer_8x2+1+1cstep_bs64_128model_2layer	588.0	0.766	5	0.753	5	5.0
conv2_8x2cstep_bs64_128model_1layer	100.0	0.766	5	0.742	7	6.0
conv2_clayer_8x2+1cstep_bs64_128model_2layer	282.0	0.751	8	0.746	6	7.0
conv2_clayer_16x1cstep_bs64	46.0	0.751	8	0.739	8	8.0
conv2_clayer_16x1cstep_bs64_128model_2layer	18.0	0.751	8	0.739	9	8.5
conv2_8x2clayer_bs64_64model_128ffw_1layer	46.0	0.746	10	0.732	10	10.0

Table 33: MCTST - JapaneseVowels

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv2_8+1x1+2cstep_bs64	**140.0**	**0.989**	**1**	**0.988**	**1**	**1.0**
conv2_8+1x1+2cstep_256x128cchannel_bs64	232.0	0.989	1	0.978	3	2.0
conv2_4+1x1+2cstep_bs64	396.0	0.981	4	0.979	2	3.0
conv2_8+2x1cstep_bs64	80.0	0.985	3	0.974	4	3.5
conv2_4x2cstep_64model_128ffw_bs64	298.0	0.980	5	0.973	5	5.0
conv2_16+1x1+2cstep_bs64_64model	270.0	0.978	6	0.972	6	6.0
conv2_16+1x1cstep_bs64_128model	208.0	0.978	6	0.969	9	7.5
conv2_8+1x2cstep_512x128cchannel_bs64	278.0	0.974	8	0.970	8	8.0
conv2_4x2cstep_128ffw_4head_2layer_bs128	332.0	0.970	10	0.970	7	8.5
conv2_8x1cstep_256x128cchannel_bs64	90.0	0.974	8	0.962	10	9.0

Table 34: MCTST - PEMS-SF

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv2_16x1cstep_256ffw_16heads_3layer_128bs	**40.0**	**0.833**	**1**	**0.811**	**1**	**1.0**
conv2_16x8cstep_default_64bs	478.0	0.815	2	0.803	3	2.5
conv3_32+16+8x2+1+1cstep_default_64bs	56.0	0.811	4	0.811	1	2.5
conv2_8x1cstep_256ffw_16heads_3layer_128bs	42.0	0.811	4	0.790	4	4.0
conv2_32x1cstep_256ffw_16heads_3layer_128bs	262.0	0.811	4	0.773	7	5.5
conv2_128x1cstep_256ffw_16heads_3layer_128bs	298.0	0.807	6	0.756	9	7.5
conv2_64x1cstep_256ffw_16heads_3layer_128bs	258.0	0.804	7	0.758	8	7.5
conv2_64x1cstep_32model_128ffw_16heads_3layer_128bs	74.0	0.796	10	0.781	5	7.5
conv4_16x1cstep_32model_64bs	34.0	0.796	9	0.780	6	7.5
conv2_64x1cstep_64model_256ffw_16heads_3layer_128bs	244.0	0.800	8	0.745	10	9.0

Table 35: MCTST - SelfRegulationSCP1

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv3_64+32+16x1cstep_4ss_bs45	**388.0**	**0.856**	**1**	**0.844**	**1**	**1.0**
conv3_64+32+16x1cstep_128+64+64cchannel_4ss_bs45	300.0	0.856	1	0.841	3	2.0
conv2_32+16x1cstep_4ss_bs45	334.0	0.852	3	0.842	2	2.5
conv2_8x1cstep_64model_layernorm_64bs_2ss	44.0	0.852	3	0.828	6	4.5
conv3_64+32+16x2cstep_2ss_bs45	142.0	0.848	6	0.838	4	5.0
conv3_64+32+16x2cstep_128+64+64cchannel_2ss_bs45	62.0	0.848	6	0.835	5	5.5
conv2_32x4cstep_bs64	96.0	0.848	6	0.810	10	8.0
conv2_4x4cstep_bs64	176.0	0.837	9	0.826	7	8.0
conv2_8x4cstep_64	166.0	0.841	8	0.824	8	8.0
conv3_64+32+16x1cstep_bs64	12.0	0.833	10	0.817	9	9.5

Table 36: MCTST - SelfRegulationSCP2

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv2_4x1cstep_bs27_ss4_32model_128ffw_4heads_1layer	**162.0**	**0.565**	**2**	**0.558**	**1**	**1.5**
conv3_16+8+4x2+1+1cstep_128x64cchannels_bs20	22.0	0.570	1	0.534	3	2.0
conv3_8+8+4x2+1+1cstep_128x64cchannels_bs20	118.0	0.560	3	0.549	2	2.5
conv2_6x3cstep_bs64	150.0	0.545	6	0.533	4	5.0
conv3_16+8+4x2+1+1cstep_128x64cchannels_bs20_4ss	34.0	0.550	4	0.528	8	6.0
conv3_4x1cstep_bs27_ss4_32model_128ffw_4heads_1layer	262.0	0.545	6	0.530	7	6.5
conv2_32x1cstep_bs27	40.0	0.540	9	0.532	5	7.0
conv3_12x3+2+1cstep_bs64	52.0	0.545	8	0.531	6	7.0
conv2_64x1cstep_bs27	4.0	0.550	4	0.508	11	7.5
conv2_8x1cstep_512x64cchannels_bs27_16model	290.0	0.525	11	0.525	9	10.0

Table 37: MCTST - SpokenArabicDigits

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv2_8+8x2+1cstep_fixedencoding_bs256	**208.0**	**0.998**	**1**	**0.997**	**1**	**1.0**
conv2_8+8x2+1cstep_128model_bs128	22.0	0.998	2	0.996	2	2.0
conv2_2x2cstep_bs128	62.0	0.997	3	0.996	3	3.0
conv2_2x1cstep_bs128	62.0	0.997	5	0.996	4	4.5
conv2_8+8x2+1cstep_128model_bs_ss2	70.0	0.997	4	0.995	5	4.5
conv3_32+16+8x2+1+1cstep_bs128_32model	158.0	0.996	6	0.994	6	6.0
conv3_2x2cstep_bs128	170.0	0.995	7	0.994	7	7.0
conv2_8+8x2+1cstep_128model_bs128_ss2	36.0	0.995	8	0.992	8	8.0
conv2_8+8x2+1cstep_128model_bs128_ss2	14.0	0.995	9	0.989	10	9.5
conv2_8x2+1cstep_128model_bs128_ss2	32.0	0.995	10	0.990	9	9.5

Table 38: MCTST - UWaveGestureLibrary

Configuration	Epoch	High-score	High-rank	10-accuracy	10-rank	Total-Rank
conv2_16x1cstep_128+64cchannel_bs64	**492.0**	**0.912**	**4**	**0.913**	**1**	**2.5**
conv2_4x2cstep_256+128cchannel_bs64	252.0	0.912	4	0.913	1	2.5
conv2_16x2+1cstep_bs64	582.0	0.912	4	0.909	3	3.5
conv2_16x2+1cstep_bs64	288.0	0.912	4	0.909	4	4.0
conv2_64+32x2+1cstep_bs128_32model	570.0	0.908	7	0.908	5	6.0
conv2_8x1cstep_bs64	196.0	0.908	7	0.908	5	6.0
conv2_16x2+1cstep_bs64	28.0	0.938	1	0.896	12	6.5
conv2_16x2+1cstep_bs128	596.0	0.904	9	0.904	7	8.0
conv2_4x2cstep_256+128cchannel_bs64	12.0	0.917	2	0.847	16	9.0
conv2_16x2+1cstep_bs128_32model	596.0	0.900	11	0.900	8	9.5