Data-driven machine learning and optimization pipelines for real-world applications

Koch, M.

CHAPTER 4

Multivariate Time Series Classification

In recent publications TSC plays an important and challenging role. Most of the proposed approaches for TSC can be categorized into univariate or multivariate techniques. In many cases, only results on univariate data sets are shown due to the limited public availability of multivariate data (Fawaz et al., 2018; Fu, 2011; Esmael et al., 2012; Aggarwal, 2015b). However, many applications, e.g., in engineering or medicine, produce more than one time series. In such cases, techniques dealing with multivariate data sets can significantly improve the performance. Some multivariate data sets were recently published, which has enabled the scientific study of multivariate techniques for TSC at a large scale (Bagnall et al., 2018). In the following section 4.1, the most relevant earlier contributions in the field of Multivariate Time Series Classification (MTSC) are summarized. Then several approaches for MTSC are introduced.

4.1 Related Work

Hidden Markov models showed improved results on two multivariate data sets when compared to an approach proposed in Moghaddam and Pentland (1997) and to neural networks (Bashir et al., 2005). A two kernel approach with a vector autoregressive model outperformed other kernel-based approaches in three out of five tested data sets (Cuturi and Doucet, 2011). Based on industrial time series from drilling sensors a trend- and value approach was proposed in Esmael et al. (2012). This approach relies on and extends the symbolic aggregate approximation (SAX) (Lin et al., 2003), i.e., a segment of the time series is represented by (v, t) where $v \in \{low, normal, high\}$ describes the relative magnitude and $t \in \{up, down, straight\}$ identifies the so-called slope. These extracted symbolic

sequences are used as a feature matrix for a classification with, e.g, k-nearest neighbors or decision trees. As an extension of dynamic time warping, the so-called correlation-based dynamic time warping was developed for multivariate time series (Bankó and Abonyi, 2012). These authors combined similarity measures from dynamic time warping and principal component analysis. In 2014, based on structural engineering applications the use of genetic programming for multivariate time series classification was investigated (Harvey, 2014). The author computed 47 features based on using a sliding window. These features were processed with genetic programming techniques. In this benchmark study, however, the methods were tested mainly on univariate datasets. In 2009, shapelets were proposed, which are considered to be subsequences of the time series (Ye and Keogh, 2009). Most discriminating shapelets of different classes are identified and conglomerated in a decision tree. Therefore, the final decision of the tree is based on the comparison of those shapelets. Next to this, a so-called fast classification approach with shapelets was used on multivariate data sets resulting in enhanced accuracy and reduced computation time compared to previous shapelet methods (Grabocka et al., 2016). An autoregressive tree-based ensemble approach was proposed in Tuncel and Baydogan (2018) to model nonlinear behavior of multivariate time series. A novel approach considered different dimensions by feature interplay (Schäfer and Leser, 2017). The authors used a sliding window on each dimension in order to extract features. Based on similarity measures (learned pattern similarity) a generalized autoregression method was proposed to find local patterns in time series (Baydogan and Runger, 2016). A recent novel similarity-based method uses a time series cluster kernel with a Gaussian mixture model (Mikalsen et al., 2018). This approach can deal with missing data and avoids time consuming hyperparameter tuning. Recent contributions show the successful use of neural networks (NN) for MTSC. In this context, a Fully Convolutional Network (FCN) and a Residual Network (ResNet) are considered as baseline results (Wang et al., 2016; Geng and Luo, 2018). FCNs as well as ResNets are developed for object detection in images. ResNets consist, like FCNs, of convolutional layers but with a deeper structure. An overview of other deep learning methods for TSC can be found in Fawaz et al. (2018).

The methods presented above, however, are highly specialized and use a sophisticated modeling approach. They are neither developed for automated modeling nor tested on a significant number of data sets.

4.2 Overview of Approaches for MTSC

In section 4.3, a Plain Hand-Crafted Pipeline (PHCP) is proposed, which aims to be a practical approach using basic methods. Then, in section 4.4, parts of the pipeline are replaced by tree-based genetic programming components to additionally consider the combinations of features from different time series. In section 4.5, the embedding of state-of-the-art automated machine learning approaches in the pipeline is shown. Furthermore, in section 4.6, state-of-the-art neural networks for time series classification, as well as a hand-crafted neural network design are introduced.

4.3 General Approach and Plain Hand-Crafted Pipeline

Most of the approaches to classify time series are based on computationally expensive methods which are nontransparent in the choice of the features. In contrast, one of our hand-crafted approaches is considered to be a plain approach with using basic methods. It aims at being numerically efficient in order to solve classification problems with good performance and acceptable computational effort. The PHCP contains elements of feature-based techniques which have the effect of creating interpretable features.

The approach consists of seven steps, which are each explained in more detail in the following subsections. The first two steps, namely

1. Preprocessing,
2. Exploratory Data Analysis,

require some hand work due to the variety of raw time series characteristics from different domains. This indicates that different data sets require individual preprocessing methods, which have to be identified in a manual process once for each data set. To get an understanding of the data set, it is strongly advised to also execute the exploratory data analysis phase. The next four phases, namely

3. Feature Extraction from Time Series,
4. Feature Selection,

5. Training of a Classifier,

6. Hyperparameter Optimization

form our so-called pipeline. These four steps are completely automated with the input consisting of the preprocessed time series data and the output being the performance scores after the hyperparameter optimization. Our plain pipeline is presented in Algorithm 1: It takes as input repeated measures $\mathcal{X}$ containing N recordings and corresponding target labels Y and starts by extracting time series features (line 3, using the tsfresh package, see Algorithm 2) for each recording in $\mathcal{X}$. Given d-dimensional time series and p predefined feature functions, $d \cdot p$ features are generated for each recording, resulting in a feature matrix $\mathbf{\Phi}^0 \in \mathbb{R}^{N \times dp}$.

The feature matrix then undergoes the feature selection procedure (line 6), where the so-called Boruta algorithm is adopted (see Algorithm 3). Prior to the model training, the feature matrix is then split into the training and test matrices $\mathbf{\Phi}_{\text{train}}, \mathbf{\Phi}_{\text{test}}$ respectively (the split of the matrix is done row-wise). To tune the hyperparameters of the modeling algorithm $\mathcal{A}$, the empirical performance of a candidate hyperparameter vector $\theta \in \Theta$ is assessed through a N_{CV}-fold cross validation (CV) that is conducted on the training matrix (lines 10-15), based on a performance metric f, e.g., accuracy or F1 (see chapter 3.1).

The performance values are obtained and averaged over all validation data sets $(\mathbf{\Phi}_{\text{val}}, Y_{\text{val}})$, which is then passed into a hyperparameter optimizer $\mathcal{H}$ for proposing new candidate hyperparameter settings (line 18). Finally, the best hyperparameter vector θ_{best} is used to train the algorithm $\mathcal{A}$ on the entire training data (line 20) and the generalization ability of the model is measured on the test set (line 22).

The last phase, namely

7. Evaluation of Results

is the manual assessment of the results to evaluate whether the results match the hypotheses.

The seven phases mentioned above are described in detail below. Note that the emphasis of this work is the pipeline itself. Therefore, the preprocessing and the exploratory data analysis step are only discussed from a general perspective.

Algorithm 1 Plain hand-crafted pipeline

Require: $\mathcal{X} = \{\mathbf{X}^{(1)}, \ldots, \mathbf{X}^{(N)}\} \subseteq \mathbb{R}^{L \times d}$ input time series, $Y \in \mathcal{C}^N$ target labels, $\mathcal{F} = \{F_i\}_{i=1}^{p}$ feature functions, performance metric f, a ML algorithm $\mathcal{A}$, its configuration space Θ and an algorithm configurator $\mathcal{H}$.

procedure PLAIN-PIPELINE
  for $i = 1$ to N **do**
    $\boldsymbol{\phi}^{(i)} \leftarrow$ FEATURE-EXTRACTION $\left(\mathbf{X}^{(i)}, \mathcal{F}\right)$ ▷ Alg. 2
  end for
  $\boldsymbol{\Phi}^0 \leftarrow \left(\boldsymbol{\phi}^{(1)}, \boldsymbol{\phi}^{(1)}, \ldots, \boldsymbol{\phi}^{(N)}\right)^\top \in \mathbb{R}^{N \times dp}$
  $\boldsymbol{\Phi} \leftarrow$ FEATURE-SELECTION$(\boldsymbol{\Phi}^0, Y)$ ▷ Alg. 3
  $(\boldsymbol{\Phi}_{\text{train}}, Y_{\text{train}}), (\boldsymbol{\Phi}_{\text{test}}, Y_{\text{test}}) \leftarrow (\boldsymbol{\Phi}, Y)$ ▷ train-test split
  Sample $\theta \in \Theta$ randomly
  while stopping criteria are not fulfilled **do**
    $\{(\boldsymbol{\Phi}^{(i)}, Y^{(i)})\}_{i=1}^{N_{\text{CV}}} \leftarrow (\boldsymbol{\Phi}_{\text{train}}, Y_{\text{train}})$ ▷ cross validation split
    for $i = 1$ to N_{CV} **do**
      $\boldsymbol{\Phi}_{\text{val}} \leftarrow \boldsymbol{\Phi} \setminus \boldsymbol{\Phi}^{(i)}, Y_{\text{val}} \leftarrow Y \setminus Y^{(i)}$
      $\mathcal{M} \leftarrow \mathcal{A}.\texttt{train}(\boldsymbol{\Phi}^{(i)}, Y^{(i)}, \theta)$
      $\widehat{Y} \leftarrow \mathcal{M}.\texttt{predict}(\boldsymbol{\Phi}_{\text{val}})$
      $f_i \leftarrow f(Y_{\text{val}}, \widehat{Y})$
    end for
    $\bar{f} \leftarrow \sum_i f_i / N_{\text{CV}}$
    $\theta \leftarrow \mathcal{H}(\theta, \bar{f})$ ▷ Learn new hyperparameters
  end while
  $\mathcal{M} \leftarrow \mathcal{A}.\texttt{train}(\boldsymbol{\Phi}_{\text{train}}, Y_{\text{train}}, \theta_{\text{best}})$
  $\widehat{Y} \leftarrow \mathcal{M}.\texttt{predict}(\boldsymbol{\Phi}_{\text{test}})$
  $f_{\text{test}} \leftarrow f(Y_{\text{test}}, \widehat{Y})$
  return $\boldsymbol{\Phi}, \mathcal{M}, \theta_{\text{best}}, f_{\text{test}}$
end procedure

4.3.1 Preprocessing

Dealing with time series usually implies extensive data preparation. Such raw time series often contain noise and have different lengths, formats and units. Especially time series data from sensors requires certain signal processing techniques like, e.g., filters or transformations (time domain to frequency domain). However, the individual techniques are strongly depending on the domain and the data set at

Algorithm 2 Feature extraction with tsfresh package

Require: $\mathbf{X} \in \mathbb{R}^{L \times d}$ an input time series and $\{F\}_{i=1}^{p}$ feature functions.

procedure FEATURE-EXTRACTION($\mathbf{X}, \{F_i\}_{i=1}^{p}$)
  for $i = 1$ to d **do**
    for $k = 1$ to p **do**
      $n \leftarrow (i-1)d + k$
      $\phi_n \leftarrow F_k\left(\mathbf{X}_{\cdot j}\right)$ ▷ j-th column of $\mathbf{X}$
    end for
  end for
  return $(\phi_1, \phi_2, \ldots, \phi_{dp})^\top$
end procedure

hand. After having processed the data, the exploration phase starts to gain an understanding of the data.

4.3.2 Exploratory Data Analysis

The exploration phase serves the purpose of becoming familiar with the data. This phase is mostly done manually and essential for building comprehensible prediction models, because a good data understanding helps building high performing models. Based on visualization and statistical methods the data set is analyzed in order to find, e.g., outliers and to handle missing values. Especially on small data sets, outliers and missing values can have a significant impact. Therefore, treating those is important for enhancing the model performances (Lissandrini et al., 2018; Zuur et al., 2010; Kantardzic, 2011). In this phase many different techniques can be important to ideally identify first patterns in the data.

4.3.3 Feature Extraction from Time Series

This work aims at creating comprehensible and computationally efficient classification models by automatically generating and selecting time series features. Depending on the domain and the data set, very different features are significant. Traditionally, such features are constructed by hand from the time series, which requires good knowledge from experts. Constructing such significant features in a manual process, however, is usually very time consuming. In order to create a

Algorithm 3 Feature selection with Boruta algorithm

Require: $\mathbf{\Phi} \in \mathbb{R}^{N \times m}$ feature matrix with N samples and m features, $Y \in \mathcal{C}^N$ target, B maximal iterations, $\mathrm{CDF}_{B(k,1/2)}$ cumulative distribution function of a binomial random variable $B(k, 1/2)$ with k trials and $1/2$ success probability, a significance level α and a random forest algorithm RF with its hyperparameter θ

procedure FEATURE-SELECTION($\mathbf{\Phi}, Y$)
  $\{c_1, c_2, \ldots, c_m\} \leftarrow \{0, 0, \ldots, 0\}$ ▷ counters
  $S \leftarrow \emptyset, R \leftarrow \emptyset$ ▷ selections and rejections
  for $k = 1$ to B and $S \cup R \neq [1..m]$ **do**
    $I \leftarrow [1..m] \setminus R$ ▷ remaining feature indices
    $\mathbf{\Phi}' \leftarrow (\mathbf{\Phi}_{\cdot i})_{i \in I}$
    for $i \in I$ **do**
      $\mathbf{\Phi}^{\text{shadow}}_{\cdot i} \leftarrow$ RANDOM-SHUFFLE($\mathbf{\Phi}'_{\cdot i}$)
    end for
    $\mathcal{M} \leftarrow$ RF.`train` $\left((\mathbf{\Phi}', \mathbf{\Phi}^{\text{shadow}}), Y, \theta\right)$
    $(\mathbf{z}, \mathbf{z}^{\text{shadow}}) \leftarrow$ FEATURE-IMPORTANCE($\mathcal{M}$)
    $\tau \leftarrow \max \mathbf{z}^{\text{shadow}}$ ▷ threshold
    $\alpha' \leftarrow \alpha / k$ ▷ Bonferroni correction
    for $i \in I$ **do**
      if $z_i > \tau$ **then**
        $c_i \leftarrow c_i + 1$
      end if
      $p_i \leftarrow 1 - \mathrm{CDF}_{B(k,1/2)}(c_i - 1)$ ▷ right tail
      $p'_i \leftarrow \mathrm{CDF}_{B(k,1/2)}(c_i)$ ▷ left tail
      if $p_i < \alpha'$ **then**
        $S \leftarrow S \cup \{i\}$ ▷ select feature i
      else if $p'_i < \alpha'$ **then**
        $R \leftarrow R \cup \{i\}$ ▷ reject feature i
      end if
    end for
  end for
  return $(\mathbf{\Phi}_{\cdot i})_{i \in S}$
end procedure

generic and automated approach to generate features for time series, a massive number of time series features can be computed in order to subsequently select

the most relevant features for the overall classification task.

For the feature extraction phase, we adopt 63 parametric feature functions[1] (e.g., autocorrelation, kurtosis or skewness) that are pre-defined in the tsfresh package (Christ et al., 2016, 2018). In total, by taking multiple different parameterizations (e.g., different time lags when calculating the autocorrelation) for each feature function, 794 features (tsfresh features) are computed by tsfresh for each time series. In this work, tsfresh was used with its default settings (see Chapter 3.4.1).

From this automatically generated feature space, the most significant tsfresh features are selected in the next step.

4.3.4 Feature Selection

Various feature selection methods have been proposed and applied, e.g., forward, backward and recursive selection (Witten et al., 2011). Importantly, all those feature selection methods require a measure of the feature importance with respect to the modeling task at hand. For instance, in random forests, the importance of a particular feature can be defined as the mean decrease of impurity (e.g., Gini impurity) over all the nodes that split this feature (Breiman, 2001). In the PHCP, a recent feature selection algorithm called Boruta (Kursa and Rudnicki, 2010) is adopted due to the fact that: 1) it is computationally relatively cheap and 2) it has shown best performances when compared to other alternatives (Koch et al., 2018). The Boruta algorithm is summarized in Algorithm 3: in each iteration, it starts with creating so-called "shadow" features by randomly shuffling the values of each feature (line 7-9). A random forests model is trained on the combination of original and shadow features, from which the feature importance is determined (line 11). A feature is considered for selection if its importance dominates the maximum importance of all shadow features (line 12, 15). The number of times such a dominance occurs is counted for each feature (line 16), on which a binomial test is performed to determine the statistical significance (line 18 - 23).

[1] Please see `https://tsfresh.readthedocs.io/en/latest/text/list_of_features.html` for the detailed list of features.

4.3.5 Training of a Classifier

In this phase the selected Boruta features are used to train a random forest classifier. Of course, any other classifier can be implemented here but due to its simplicity and its efficiency we have used a random forest in our pipeline. Random forests belong to the class of ensemble learning methods and are considered to achieve good performance in different problem domains. A random forest is the conglomeration of many decision trees and the resulting decision is the average outcome of all those decision trees (Hastie et al., 2009).

4.3.6 Hyperparameter Optimization

Setting up hyperparameters properly is vital to the performance of a machine learning model. To boost up the performance of the random forest model on the data, it is necessary to conduct a hyperparameter optimization. The list of hyperparameters under consideration and their ranges are shown in Table 4.1. Note that the resulting search space for hyperparameter optimization contains integer variables as well as categorical ones. There are a variety of well-established algorithms for this task, including grid search, evolutionary algorithms and Bayesian optimization (Geron, 2017).

Table 4.1: Hyperparameter search space for optimizing the random forest classifier.

Parameter	**Range**
Max depth of each tree	$\{1, 2, \ldots, 100\}$
Number of trees	$\{1, 2, \ldots, 100\}$
Max number of features when splitting a node	{`auto`, `sqrt`}
Min number of samples required to split a node	$\{2, 3, \ldots, 20\}$
Min number of samples required in the leaf node	$\{1, 2, \ldots, 10\}$
Use bootstrap training samples?	{`True`, `False`}

In the PHCP, the Bayesian optimization algorithm is chosen due to its efficiency when optimizing expensive problems. In detail, a recent variant of it, called Mixed-integer Parallel Efficient Global Optimization (MIP-EGO) (Wang et al., 2017, 2018) is adopted as it handles the mixed-integer categorical variables in an efficient way. The MIP-EGO algorithm is executed for 200 iterations, where in each iteration a candidate hyperparameter setting is suggested and its goodness is

determined by the quality of the corresponding model on a validation set. Using N_{CV}-fold cross validation on the training data, the hyperparameter optimizer uses the average classification quality measure $\bar{f}$ across the N_{CV} validation sets as the overall quality measure for each configuration θ under consideration.

4.3.7 Evaluation of the Results

In the last step of this approach the outcome has to be evaluated. In this context, it is essential to run the computation of each method multiple times and use its average to reduce noise. Sometimes machine learning models tend to overfit / underfit, therefore it is important to evaluate the training performance and the validation performance in detail. Often used evaluation metrics are accuracy and the $F1$-score for class-imbalanced data sets (see chapter 3.1).

Furthermore, visualization methods like, e.g., learning curves, the ROC or the precision / recall curve can provide additional performance insights (see Chapter 3.1).

4.4 Hand-Crafted Approach with Genetic Programming

In the Hand-Crafted Pipeline with Genetic Programming (HCPGP) a feature generation module is implemented in the pipeline. This feature generation is a tree-based genetic programming approach which conglomerates many so-called less significant features (low level features) into several significant features (high level features). In detail, the search domain is defined by symbolic expressions for function approximation (Koza and Rice, 1992; Poli et al., 2008). These symbolic expressions are typically represented as parse trees, such as illustrated in Figure 4.1, where $x_0, x_1, \ldots$ describe the features. The genetic operators are applied to those trees, for example by subtree exchanges between different trees in a population (crossover), or by mutation of subtrees or terminal nodes in the tree.

However, this requires iterations over many generations to find combinations with adequate classification qualities. The large tsfresh feature space limits the efficiency of the computation. As mentioned, this work focuses on competitive

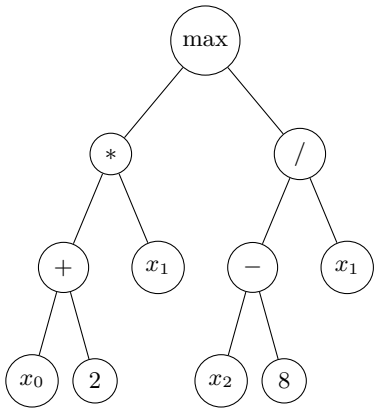

Figure 4.1: An example of a tree-based representation of symbolic expressions, as used in Genetic Programming.

methods regarding performance and computation times. Testing large numbers of combinations is not very efficient here.

Therefore, the dimensionality in terms of the number of initial low level features has to be reduced either way. Promising ways to obtain a smaller set of new features without removing any information would be by using CNNs or a PCA. It is key not to drop information at this point, which is more or less granted in both methods. In this work we are using PCAs mainly due to its lower computation time when compared to CNNs. PCA represents the data in a lower dimension with linear principal components whereas the so-called kernel Principal Component Analysis (kPCA) transforms the data into a higher dimensional feature space and applies PCA on this feature space. Compared to standard PCA, kPCA allows to describe also non-linear behavior (Bishop, 2009).

With an increased prediction scores of approximately 2.5% over all data sets our first experiments showed a slight increase in performance when using kPCA instead of PCA. Therefore, we used kPCA in this work.

Furthermore, in this study we set the number of components of the kPCA to a relatively small number (here 15). This means that the initial tsfresh feature space is reduced to 15 components. Setting the number of components to such a small number enormously improves the efficiency of the genetic algorithm regarding computation time. Especially this boost in efficiency allowed us to investigate the

genetic algorithm and its parameters in this study in a computationally feasible way.

When dealing with these algorithms we were facing issues with overfitting, because from our experience such genetic programming algorithms tend to find perfect solutions on the training data by overfitting the data. Therefore, the fitness scores are based on a cross validation approach (see chapter 3.1). Furthermore, as fitness function of our HCPGP approach a performance metric like accuracy is used to maximize the overall performance. This is used together with a minimization of the difference between validation score and training score of the cross validation. This setting aims at reducing the overfitting tendency of the algorithm on the training set of the HCPGP approach.

Table 4.2 shows the parameters of the genetic programming approach. This parameter setting aims at generating a single complex feature based on 15 basic features which can be used as variables in the symbolic expressions generated by genetic programming.

Table 4.2: Parameter of the genetic feature generation algorithm.

Parameter	**Value**
Population	10
Generation	1500
Crossover	0.5
Mutation	0.1
Tree depth	1-10
Number of features	15

The basic features are obtained from applying the tsfresh and kPCA preprocessing algorithms. The performance of the complex features generated by the genetic programming approach is measured by using a random forest and cross validation (see above).

The whole pipeline with genetic programming is shown below:

3. Feature Extraction from Time Series,
4. Kernel PCA,
5. Feature Generation with Genetic Programming,

6. Training of a Classifier and
7. Hyperparameter Optimization.

The pipeline based on genetic programming is shown in Algorithm 4: the input is similar to Algorithm 1 except that this algorithm requires genetic operators ψ and arithmetic operators ζ which are used to construct a parse tree Ω. The feature extraction is executed as presented in Algorithm 1 (line 2-5). The kPCA is applied on the feature matrix (line 3). Then, the feature matrix is split into the training and test matrices $\mathbf{\Phi}_{\text{train}}, \mathbf{\Phi}_{\text{test}}$ respectively (the split of the matrix is done row-wisely). Based on the genetic operator ψ and the arithmetic operators ζ an initial parse tree is constructed for the given population size (here 10).

From line 6-19 the parse tree, which achieves the best cross-validated score of the highest validation performance subtracted by the absolute difference of validation score and training score, is optimized over generations (here 1500). Then, this optimized parse tree Ω_{best} is used to compute its depending features of the split (line 20). These features are considered in the same hyperparameter optimization procedure and validation method as shown in Algorithm 1 (line 8-22).

4.5 AutoML

AutoML generates machine learning pipelines in an automated approach. In this work the following AutoML methods are considered: Auto-Sklearn, TPOT, AMLPA, GAMA, H2O, MLBOX, ATM and RECIPE (see Chapter 3.8). All of these AutoML approaches require stationary features. Therefore, it is implemented after phase 3 (feature extraction):

3. Feature Extraction from Time Series,
4. AutoML,

where phase 4 now replaces the steps 4 - 7 in the pipeline definition given before.

Algorithm 4 Hand-crafted pipeline with genetic programming

Require: $\mathcal{X} = \{\mathbf{X}^{(1)}, \ldots, \mathbf{X}^{(N)}\} \subseteq \mathbb{R}^{L \times d}$ input time series, $Y \in \mathcal{C}^N$ target labels, $\mathcal{F} = \{F_i\}_{i=1}^{p}$ feature functions, parse tree Ω, its genetic operator ψ, its arithmetic operators ζ, performance metric f, a ML algorithm $\mathcal{A}$, its configuration space Θ, an algorithm configurator $\mathcal{H}$ and a genetic programming algorithm GP.

procedure GENETIC-PIPELINE
  Execute lines 2-5 of Algorithm 1
  $\mathbf{\Phi} \leftarrow \text{KPCA}(\mathbf{\Phi}^0, Y)$
  $(\mathbf{\Phi}_{\text{train}}, Y_{\text{train}}), (\mathbf{\Phi}_{\text{test}}, Y_{\text{test}}) \leftarrow (\mathbf{\Phi}, Y)$ ▷ train-test split
  Sample initial parse tree Ω with ψ and ζ
  $\Omega_{\text{best}} \leftarrow \Omega,\ \bar{f}_{\text{best}} \leftarrow -\infty$
  while the stop criteria are not fulfilled **do**
    $\{(\mathbf{\Phi}^{(i)}, Y^{(i)})\}_{i=1}^{N_{\text{CV}}} \leftarrow (\Omega(\mathbf{\Phi}_{\text{train}}), Y_{\text{train}})$ ▷ cross validation split
    for $i = 1$ to N_{CV} **do**
      $\mathbf{\Phi}_{\text{val}} \leftarrow \mathbf{\Phi} \setminus \mathbf{\Phi}^{(i)}, Y_{\text{val}} \leftarrow Y \setminus Y^{(i)}$
      $\mathcal{M} \leftarrow \mathcal{A}.\texttt{train}(\mathbf{\Phi}^{(i)}, Y^{(i)}, \theta)$
      $\widehat{Y}_{\text{train}} \leftarrow \mathcal{M}.\texttt{predict}(\mathbf{\Phi}^{(i)})$
      $\widehat{Y}_{\text{val}} \leftarrow \mathcal{M}.\texttt{predict}(\mathbf{\Phi}_{\text{val}})$
      $f_{\text{train,i}} \leftarrow f(\mathbf{Y}^{(i)}, \hat{\mathbf{Y}}_{\text{train}})$
      $f_{\text{val,i}} \leftarrow f(\mathbf{Y}_{\text{val}}, \hat{\mathbf{Y}}_{\text{val}})$
    end for
    $\bar{f}_{\text{train}} \leftarrow \sum_i f_{\text{train,i}}/N_{\text{CV}}$
    $\bar{f}_{\text{val}} \leftarrow \sum_i f_{\text{val,i}}/N_{\text{CV}}$
    $\bar{f} \leftarrow \bar{f}_{\text{train}} - |\bar{f}_{\text{train}} - \bar{f}_{\text{val}}|$
    if $\bar{f} > \bar{f}_{\text{best}}$ **then**
      $\bar{f}_{\text{best}} \leftarrow \bar{f},\ \Omega_{\text{best}} \leftarrow \Omega$
    end if
    $\Omega \leftarrow \text{GP}(\Omega, \psi, \bar{f})$ ▷ Learn a new parse tree from GP.
  end while
  $(\mathbf{\Phi}_{\text{train}}, Y_{\text{train}}), (\mathbf{\Phi}_{\text{test}}, Y_{\text{test}}) \leftarrow (\Omega_{\text{best}}(\mathbf{\Phi}), Y)$ ▷ train-test split
  Execute lines 8-22 of Algorithm 1
  return $\mathbf{\Phi}, \mathcal{M}, \Omega_{\text{best}}, \theta_{\text{best}}, f_{\text{test}}$
end procedure

4.6 Neural Network Architectures

In this work three neural network architectures for MTSC are considered, namely the hand-crafted architecture Convolutional Neural Network + Long Short-Term Memory (CNN+LSTM), the ResNet (He et al., 2015) and the FCN (Long et al., 2014). Figure 4.2 illustrates these architectures.

The FCN and the ResNet architectures were proposed for time series classification with 2000 epochs (Wang et al., 2016). Initially, both architectures were developed for object detection on images and modified for time series classification with the result of being very competitive regarding other state-of-the-art approaches. Compared to FCNs, a ResNet has a deeper structure by using so-called shortcut connections between layers. ResNets are considered as state-of-the-art in image detection. In a recent publication the time series modification of FCNs showed better results on time series classification tasks compared to ResNets and other architectures (Wang et al., 2016).

Next to FCNs and ResNets, based on mainly real-world time series data (automotive) within this work a hand-crafted neural network has been developed for MTSC by combining CNNs and LSTMs. CNNs can extract important information from the time series and exclude unimportant parts, i.e. CNNs transform the time series into a shorter, high level representation. Furthermore, CNNs learn local patterns, i.e., after learning a pattern this can be found independently of its location on the time axis (see Chapter 3.6). In this setting we are using 2 times 2 CNNs with 128 units or neurons, each followed by a Max Pooling. The number of units can be adapted to the need of the certain data set. However, our real-world data set has 701 time steps and in the optimization phase 128 units showed favorable results. As activation function the Rectified Linear Unit (ReLU) function is used. In detail, the most informative sections are extracted by the CNNs and then, the Max Pooling extracts the maximum value of every 3 successive values. Hence Pooling removes unnecessary information like the exact position of a certain pattern, but the information of a pattern itself still exists, i.e. Pooling thins the data out. This again results in a higher level representation of the data.

However, CNNs do not consider the order of appearance of certain patterns in the data. Therefore, in this setting CNNs are only used to shorten the sequence and to provide this shortened data into a LSTM layer. LSTMs are computationally more

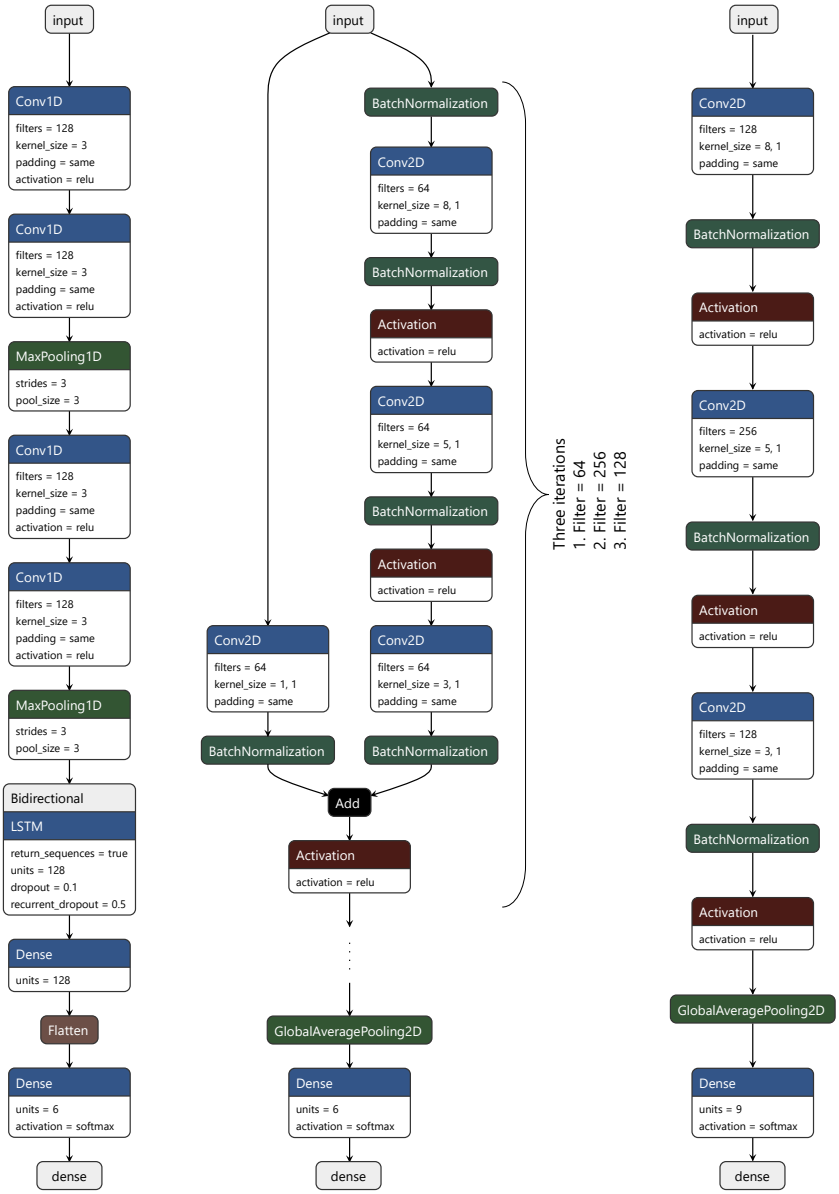

Figure 4.2: In this study three different neural network architectures are used, namely CNN+LSTM (left), ResNet (center) and FCN (right).

expensive than CNNs, but LSTMs do consider the order of values. Therefore, the CNN+LSTM uses CNNs as preprocessor to reduce the long sequence with keeping only high-level features. In this way, the sequence can be shortened into a highly informative sequence, which is processed by the LSTM. The LSTM returns the extracted sequence to a dense layer which is a fully connected layer. The output of the dense layer is flattened in order to be processed in the last dense layer. This last layer contains a softmax activation function. For each class in the data set exists one unit. The output of all units must add up to 1.0, i.e. the output of each individual class unit shows the probability of this class to be the target.

The hyperparameter of this Neural Network has been tuned for the real-world automotive data set. The LSTM has 128 units, a dropout of 0.1 and a recurrent dropout of 0.5. Dropout is a so-called regularization method. It randomly drops units to prevent overfitting and the lower number of units allows faster model training. The *normal* dropout regularizes the inputs between layers, whereas the recurrent dropout drops units of the recurrent state, i.e. it regularizes between different time steps. Furthermore, in this setting 16 data points are used at once to update the model parameters, i.e. the batch size is 16. All data are passed up to 2000 times through the network. This are the so-called epochs. The learning rate is set initially to 0.001 and when the objective function stagnates it is reduced down to 0.0001. In order to reduce computation time, an early stopping criteria is defined to stop training when experiencing no further improvement in the objective function.