UNIVERSITÀ DEGLI STUDI DI TORINO

Dipartimento di Informatica

Scuola di Scienze della Natura

Corso di laurea magistrale in Informatica

Curricula INTELLIGENZA ARTIFICIALE E SISTEMI INFORMATICI

Tesi di laurea magistrale in informatica

Incomplete time-series classification methods

Relatore
Prof. Esposito Roberto

Corelatore
Dott. Ienco Dino

Candidato
Shtjefni Donald

Anno Accademico 2021/2022

Abstract

This thesis investigates the problem of classification for incomplete time series. Time series data, characterised by a sequence of observations taken over time, are abundant in a wide range of fields such as finance, healthcare and Earth satellite observations. However, the presence of missing values in these series can make the classification task more complex and challenging.

To address this issue, we implemented several machine learning models and compared their performance on a dataset of Satellite Image Time Series (SITS) to study how natural and semi-natural areas evolve over time. Various models have been employed, including Random Forests, Temporal Convolutional Neural Networks (TempCNNs), Recurrent Neural Networks (RNNs), Generative Adversarial Networks (GANs), and Transformers. On the other hand to handle missing data, three approaches have been used: using pre-imputed data, excluding missing data, and imputing missing data. Using pre-imputed data involves using a pre-processing step to fill in the missing values using a method such as mean imputation or linear interpolation. Excluding missing data simply involves discarding the missing values, while imputing missing data involves using components of the neural network to estimate the missing values.

The results of our experiments on incomplete time series classification showed that Temporal Convolutional Neural Networks (TempCNNs) and Transformers outperformed the other models in terms of accuracy, while at the same time keeping the number of parameters low. This indicates that these models were effective in handling missing data and avoiding overfitting.

In conclusion, our study highlights the importance of considering missing data when performing time series classification and the effectiveness of TempCNNs and Transformers in addressing this issue.

Keywords— Time series classification, missing data, Random Forests, TempCNNs, RNNs, Transformers, GANs

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

Vorrei dedicare questo spazio a chi, con dedizione e pazienza, ha contribuito alla realizzazione di questo elaborato.

In particolare, voglio esprimere la mia più sincera gratitudine al mio relatore, il Prof. Esposito Roberto, e al mio correlatore, il Dott. Ienco Dino, per la loro preziosa guida durante i mesi di lavoro. Grazie ai loro consigli pratici e alla loro esperienza, ho potuto approfondire le mie conoscenze e completare questo elaborato con maggiore consapevolezza.

Ringrazio infinitamente i miei genitori, Gjovalin *e Vjollca, e mia sorella Franceska, per il loro costante sostegno e incoraggiamento in ogni mia scelta. Siete stati un sostegno costante durante tutto il mio percorso accademico, e non posso che ringraziarvi per la vostra presenza e il vostro affetto.*

Un ringraziamento speciale va a tutti i miei colleghi di corso, sia dell'università di Torino che dell'università di Klagenfurt, con cui ho condiviso momenti indimenticabili e che mi hanno offerto un ambiente accogliente e stimolante per imparare e crescere.

Desidero inoltre esprimere la mia gratitudine a tutti i miei colleghi di Weconstudio, per aver creato un ambiente di lavoro collaborativo e stimolante che mi ha permesso di crescere professionalmente. Grazie per avermi dato l'opportunità di imparare da voi e di sviluppare le mie competenze.

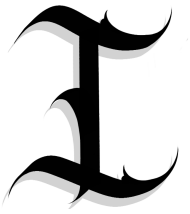
Non posso dimenticare di ringraziare tutti i miei amici, vecchi e nuovi, che mi hanno sempre sostenuto e incoraggiato durante i momenti difficili. La vostra amicizia è stata un faro di luce nei momenti di buio, e mi ha permesso di superare gli ostacoli con maggiore serenità.

Infine, voglio dedicare un pensiero speciale alla mia ragazza Michela, per il suo costante supporto e la sua presenza rassicurante. La tua presenza è stata una fonte di ispirazione e di conforto, e mi ha permesso di affrontare ogni ostacolo con maggior fiducia e determinazione. Sono fortunato ad avere una persona così speciale nella mia vita.

Contents

List of Figures

List of Tables



Introduction

The ability to capture images at a high revisit rate has been made possible by recent advances in satellite imaging technology, such as the European Union's Sentinel program. This has led to a significant increase in the availability of Satellite Image Time Series (SITS) data, which can be used to monitor the evolution of natural and semi-natural areas over time. Remote sensing applications such as land cover classification and change detection, climate monitoring and environmental studies.

However, applying supervised learning techniques to SITS data requires the availability of a reference dataset. This requires manual labelling by humans, which can be a time-consuming and labour-intensive process. Nevertheless, it is crucial to extract valuable information and insights from the data.

This thesis focuses on the development and evaluation of deep learning models for the classification of time series satellite images. The proposed models aim to classify the images accurately and efficiently, while addressing some of the challenges associated with classifying time series data.

One of the challenges we faced during our experiments was dealing with missing values in the dataset. Since our dataset consists of time series data of satellite images, it was expected that there would be some missing values due to weather conditions or technical issues. Missing values can affect the performance of deep learning models because they rely on a complete set of data to learn and make accurate predictions. Since missing values can affect the performance of our models, we approached this challenge in two ways: first, by using imputation techniques to fill in the missing values to ensure that our models could still

perform well with incomplete data; second, by conducting experiments without imputation to evaluate the impact of missing values on the performance of our models.

The models used in our study include Random Forests, Temporal Convolutional Neural Networks (TempCNNs), Recurrent Neural Networks (RNNs), Generative Adversarial Networks (GANs), and Transformers.

The thesis is organized as follows: Chapter 2 provides background information on time series and deep learning models. Chapter 3 describes the dataset used in the experiments, including its characteristics and properties. Chapter 4 provides a thorough exploration of the models we used for our time series classification tasks. Chapter 5 briefly introduces the tools we used to conduct our experiments. Chapter 6 showcase our experiments and the results we obtained. Finally, in Chapter 7, we present our conclusions and suggest possible future work in the area of time series classification.

Background

2.1 Time series

A time series is a sequence of observations taken at regular time intervals. These observations can be numerical values, categorical variables, or even text. Time series data are commonly found in a wide range of fields, including finance, economics, health care, meteorology, and transportation.

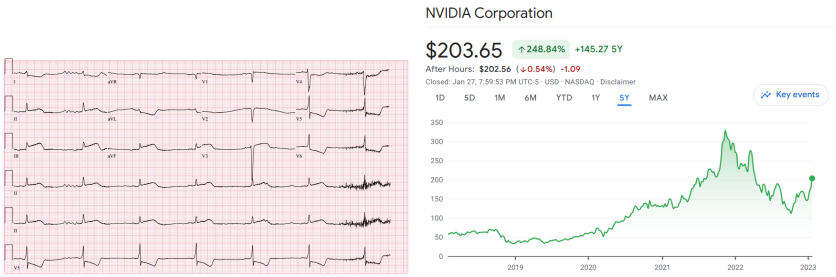


Figure 2.1: Example Time Series: Health Monitoring Records [28] and NVIDIA Stock Price [14]

In finance, for example, time series data can be used to analyze stock prices, interest rates, and currency exchange rates. In healthcare, time series data can be used to analyze patients' vital signs, such as heart rate and blood pressure. In transportation, time series data can be used to analyze traffic patterns and traffic flow.

2.1.1 Univariate and Multivariate

It is important to understand the difference between univariate and multivariate time series in addition to the concept of a time series.

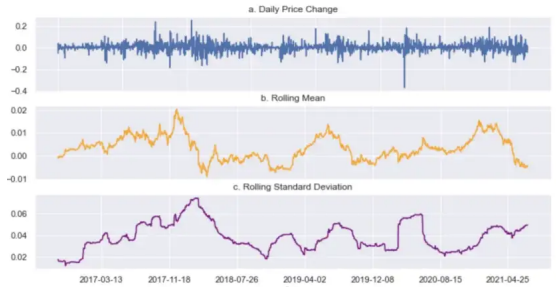


Figure 2.2: Multivariate Time Series Example [42]

Univariate time series This type of time series contains observations of a single variable, such as stock prices or temperature. The focus is on understanding patterns and trends in the single variable over time.

Multivariate time series This type of time series includes observations of multiple variables, also known as features, over time. The main focus is on understanding the relationships and interactions between the multiple variables.

Univariate time series focus on understanding the patterns and trends of a single variable over time. This can be done by analyzing the data using techniques such as time series decomposition, trend analysis, and forecasting. Multivariate time series, on the other hand, focus on understanding the relationships and interactions between multiple variables. It's important to note that the choice of model and the methods used to analyze the data depend on the type of time series data you're working with. Univariate and multivariate time series require different approaches and techniques to analyze.

2.1.2 Satellite Image Time Series (SITS)

Satellite Image Time Series (SITS) refers to a collection of satellite images taken at regular intervals over a specific area of interest. SITS data are obtained using remote sensing satellites that capture images of the Earth's surface at different wavelengths of the electromagnetic spectrum.

The European Union's Sentinel program is an example of a satellite mission that provides high quality SITS data with a long revisit time. The Sentinel satellites acquire images at different spatial resolutions, ranging from 10 meters to 60 meters, depending on the mission. These images are acquired at different time intervals, ranging from a few days to several months, providing a time series of images for a given area.

To obtain SITS data, multiple images are taken at regular intervals over a specific area of interest. After the images are acquired, they go through a pre-processing step where they are aligned to a consistent coordinate system. This step ensures that each pixel in the time series corresponds to the same location on the Earth's surface, allowing accurate analysis and comparison of the data.

2.2 Ensemble learning

Ensemble learning [26, 56] is a technique in machine learning that combines multiple models to create a more powerful model. The idea behind ensemble learning is that by combining the predictions of multiple models, the ensemble model can reduce the variance and bias of the individual models, leading to improved performance on unseen data.

There are several popular ensemble learning methods, including:

Bagging This method involves training multiple models independently on different subsets of the data and then combining their predictions by majority voting or averaging.

Boosting This method involves training multiple models sequentially, with each model attempting to correct the errors of the previous model. The final prediction is made by combining the predictions of all the models.

Random Forest This method is a combination of bagging and decision trees. It involves training multiple decision trees independently on different subsets of the data and then combining their predictions by majority voting.

Stacking This method involves training multiple models independently, and then using the predictions of those models as input features to train a meta-model.

Ensemble learning can be applied to a wide range of machine learning tasks, including classification, regression, and anomaly detection. It has been shown to improve performance on a wide range of datasets and is often used in practice to improve the performance of machine learning models.

2.2.1 Random forests

Random forests [6, 7] are an ensemble learning method used for classification and regression tasks. The name "random forest" comes from the fact that it is a collection of decision trees, where each tree is "trained" on a random subset of the data, and the final predictions are made by combining the predictions of all the trees.

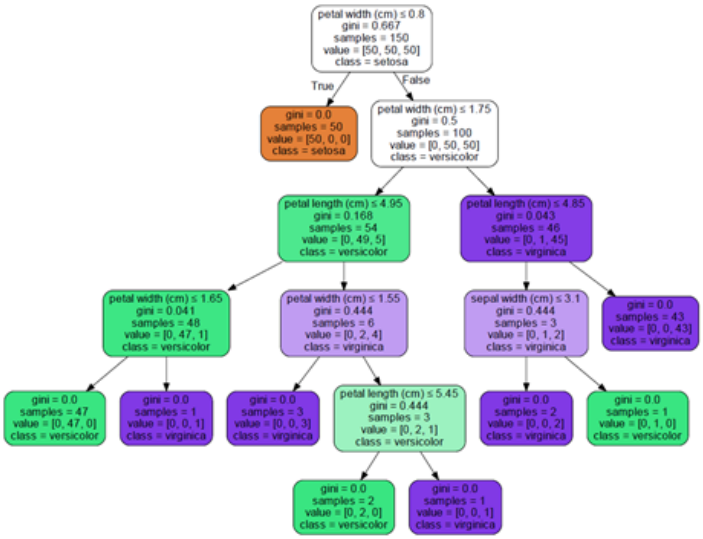


Figure 2.3: Visual representation of a decision tree [53]

A decision tree is a tree-like model that starts with a single node, called the root node, and splits the data into different subsets based on the values of the input features. Each internal node of the tree represents a test on an input feature, and each leaf node represents a prediction. The tree is built by recursively splitting the data until a stopping criterion is met, such as a maximum tree depth or a minimum number of samples in a leaf node.

In a random forest, multiple decision trees are trained on different subsets of the data, where the subsets are obtained by randomly sampling the data with replacement, also known as bootstrapping. Each tree is trained on a different subset of the data, and a different subset of the input features is used at each split of the tree. This is known as feature bagging or random subspace method.

When making a prediction, a random forest takes the average of the predictions of all the trees for regression problems, and takes a majority vote for classification problems. This combination of multiple decision trees reduces the variance of the individual trees and leads to improved performance on unseen data.

In summary, Random Forests are a combination of decision trees, where each tree is trained on a random subset of the data, and the final predictions are made by combining the predictions of all the trees. This combination of multiple decision trees reduces the variance of the individual trees and leads to improved performance on unseen data.

2.3 Deep learning

Deep learning [20] is a subfield of machine learning that utilizes deep neural networks, which are neural networks with multiple hidden layers, to learn from the data. These models are capable of learning abstract features from the data and making predictions based on those features, making them particularly useful for tasks such as image and speech recognition, natural language processing, and anomaly detection.

Deep neural networks are inspired by the structure and function of the human brain, consisting of layers of interconnected nodes, also known as neurons. The input is passed through the layers of neurons, and the output is produced at the last layer. The layers of a neural network can be divided into three main types: the input layer, the hidden layers and the output layer. The input layer receives the input data, and the hidden layers perform computations on the data and pass the results to the next layer. The output layer produces the final predictions.

Deep learning models are trained using large amounts of data and powerful computational resources, such as GPUs. The training process involves adjusting the parameters of the model, also known as weights, to minimize the error between the predictions and the true values. This process is known as supervised learning, where the model learns to map inputs to outputs based on labeled examples.

In recent years, deep learning has made significant advancements in various fields, including computer vision, natural language processing, and speech recognition. The ability of deep neural networks to learn abstract features has led to significant improvements in performance on various tasks, surpassing traditional machine learning methods.

In summary, deep learning is a subfield of machine learning that utilizes deep neural networks, which are neural networks with multiple hidden layers, to model complex patterns in data. These models are trained using large amounts of data and powerful computational resources to minimize the error between predictions and true values and have led to significant advancements in various fields.

2.3.1 Perceptron

The perceptron [49] is a type of artificial neural network first introduced in the 1950s by Frank Rosenblatt. It was designed to replicate the function of a single neuron in the brain, with the ability to learn from data and make decisions based on that learning.

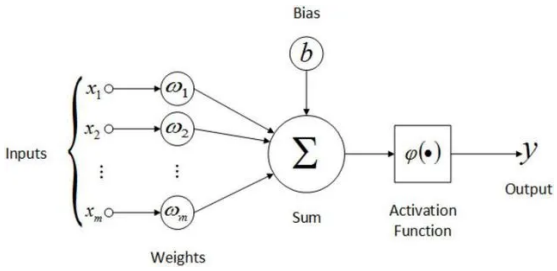


Figure 2.4: Perceptron Diagram [31]

The perceptron consists of one or more input nodes, a single output node, a bias value, and a set of weights that determine the strength of the connections between the input nodes and the output node. The input nodes receive the input data and the weights are adjusted during training in order to make the output node produce the desired output for a given input.

The perceptron uses an activation function to determine the output of the output node. The activation function can be any function that maps the input to an output, but some of the most commonly used activation functions include the step function, ReLU, sigmoid, and tanh.

The perceptron can be used for binary classification problems, where the output node produces a 0 or 1 depending on the input. It can also be used for regression problems, where the output node produces a continuous value. While the perceptron is a relatively simple neural network architecture, it has played an important role in the development of more complex neural networks and deep learning models.

2.3.2 Activation functions

The ReLU (Rectified Linear Unit) function, denoted as $\text{ReLU}(x)$, is defined as:

$$\text{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{2.1}$$

It allows for efficient computation and solves the vanishing gradient problem. ReLU is often used in image classification tasks.

The tanh (hyperbolic tangent) function, denoted as $\tanh(x)$, is defined as:

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \tag{2.2}$$

It is nonlinear and produces a smooth gradient, which can improve training performance. tanh is commonly used in recurrent neural networks for natural language processing tasks.

The sigmoid function, denoted as $\sigma(x)$, is defined as:

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{2.3}$$

It is a commonly used activation function in binary classification tasks because it produces a probability-like output. Sigmoid functions can also be used in deep learning models for tasks such as object detection and speech recognition.

2.3.3 Multilayer perceptrons

Multi-layer perceptrons (MLPs) [2, 20] are neural networks that consist of one or more layers of perceptrons. These networks are also known as fully connected networks because each neuron in a layer is connected to all neurons in the next layer.

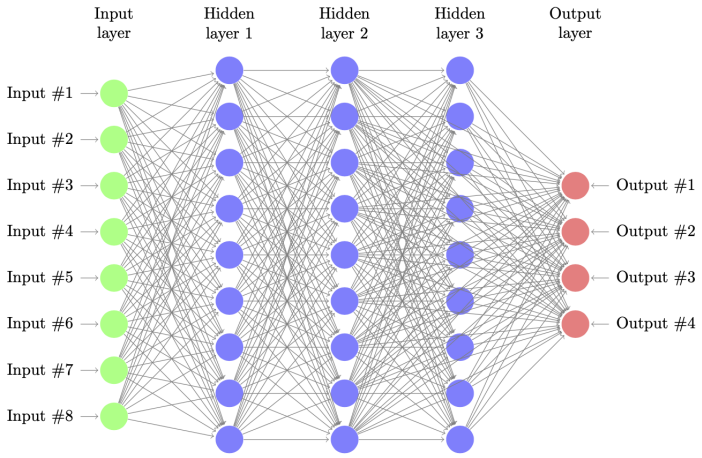


Figure 2.5: Multilayer Perceptron (MLP) Diagram [47]

The hidden layers in an MLP allow for non-linear transformations of the input data, which can help capture complex patterns in the data that would be difficult to capture with a single perceptron. Each hidden layer applies a linear transformation to the inputs, followed by a nonlinear activation function, before passing the output to the next layer.

MLPs can be used for a variety of tasks, including classification, regression, and sequence prediction. They have been successfully applied in areas such as image recognition, natural language processing, and speech recognition. MLPs can also be used as building blocks for more complex neural networks, such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs).

MLPs are trained using a process called backpropagation, which involves computing the gradient of the loss function with respect to the weights of the network, and using this gradient to update the weights through an optimization algorithm, such as stochastic gradient descent (SGD). The training process involves iterating over the training data multiple times, adjusting the weights of the network after each iteration, until the performance of the network on a validation set stops improving or a maximum number of epochs is reached.

2.3.4 Backpropagation

Backpropagation is a popular method for training neural networks. It is a supervised learning algorithm that uses a form of gradient descent optimization to minimize the error between the predicted output and the actual output. The backpropagation algorithm begins with a forward pass through the network, computing the output of each neuron layer by layer until the final output is produced.

It then calculates the error between the predicted output and the true output using a loss function. This error is then propagated back through the network, starting from the output layer and moving backwards. The weights of each neuron in the network are updated in the direction that reduces the error the most. This process is repeated until the error is minimized or a predefined stopping criterion is met.

The backpropagation algorithm requires the use of a differentiable activation function because the error must be propagated through the network to update the weights. The choice of activation function can have a significant impact on the performance of the network, and is often the subject of experimentation and tuning.

2.3.5 Loss functions

Loss functions are used in machine learning to measure the difference between predicted output and actual output. The objective of a machine learning algorithm is to minimize the loss function in order to improve the performance of the model.

There are several types of loss functions, each suitable for different types of problems. For example, mean squared error (MSE) is often used in regression problems where the goal is to predict a continuous output variable.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.4}$$

Cross-entropy loss is used in classification problems, where the output variable is discrete.

$$\text{Cross-entropy loss} = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(\hat{y}_i) \tag{2.5}$$

The choice of loss function is critical in the training process of a neural network, as it can affect the performance of the model. It is important to choose a loss function that is appropriate for the problem at hand and that allows the model to effectively optimize its parameters.

2.3.6 Batch training

Batch training is a technique used to train neural networks using batches of samples, rather than processing the entire data set at once. During training, the data is divided into small batches and the model is updated after each batch is processed. This process helps prevent overfitting and allows for more efficient use of computational resources.

In batch training, the number of iterations over the entire dataset is typically referred to as an epoch. A single epoch consists of processing each batch in the training dataset once. After one epoch, the model has seen each training sample once and has updated weights based on the average loss across all batches in that epoch. Multiple epochs are often used to further refine the weights and biases of the model.

The number of epochs used during training can affect model performance Training on too few epochs can lead to underfitting, where the model has not learned enough from the data. On the other hand, training on too many epochs can lead to overfitting, where the model has learned too much from the training data and is unable to generalize to new data. Choosing the optimal number of epochs is typically done by trial and error or using early stopping techniques.

2.3.7 Learning rate

The learning rate is a hyperparameter that controls the step size of the gradient descent optimization algorithm. It determines how much the weights of the network should be adjusted in response to error. A high learning rate results in larger weight adjustments, while a low learning rate results in smaller weight adjustments.

2.3.8 Overfitting

Overfitting is a common problem in machine learning, where a model is trained to fit the training data too well, resulting in poor performance on new, unseen data.

Regularization is a technique used to prevent overfitting by adding a penalty term to the loss function. This penalty term discourages the model from assigning too much importance to any one feature, and instead encourages it to find

simpler, more generalizable solutions. There are several types of regularization techniques, including L1 and L2 regularization, which add the absolute values and squared values of the weights, respectively, to the penalty term.

Dropout is another regularization technique that randomly drops a certain percentage of the neurons in a layer during training. This forces the remaining neurons to learn more robust features and prevents over-reliance on any one neuron. Dropout has proven to be an effective technique for reducing overfitting in deep neural networks.

2.3.9 Convolutional neural networks

Convolutional Neural Networks (CNNs) [20] are a class of deep neural networks specifically designed to process data with a grid-like structure, such as images. CNNs are composed of multiple layers, including convolutional layers, pooling layers, and fully connected layers.

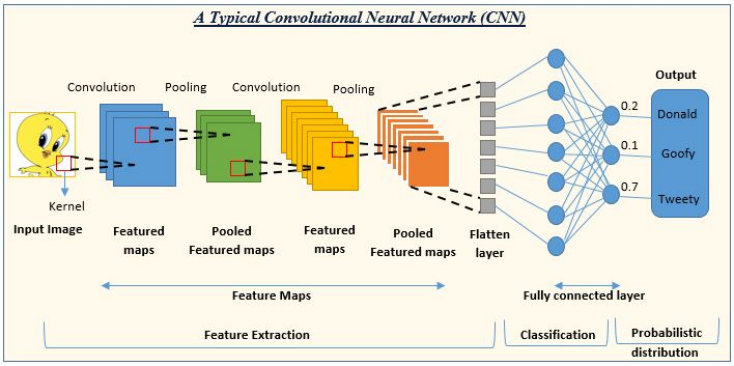


Figure 2.6: Convolutional Neural Network (CNN) Diagram: A visual representation of a deep learning architecture consisting of layers of convolutional, pooling and fully connected layers. [55]

The convolutional layer is the core component of a CNN. It applies a set of learnable filters to the input data, which are used to extract features from the data. These filters are convolved with the input data, resulting in a set of feature maps. These feature maps are then passed on to the next layer for further processing.

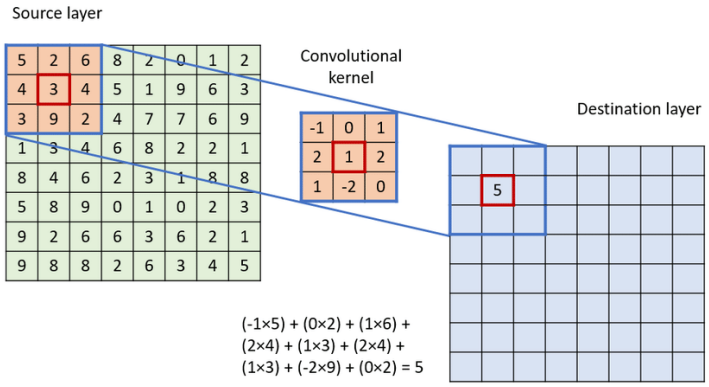


Figure 2.7: Schematic illustration of a convolutional operation. The convolutional kernel shifts over the source layer, filling the pixels in the destination layer. [46]

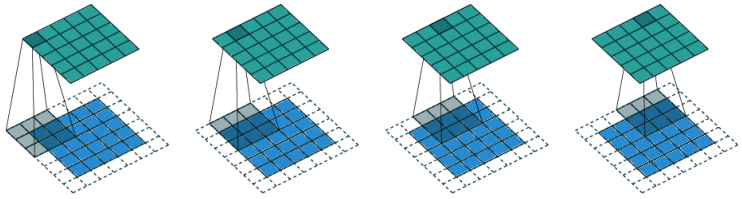


Figure 2.8: Convolving a 3×3 kernel over a 5×5 input using half padding and unit strides (i.e., i=5, k=3, s=1 and p=1) [11]

Pooling layers are used to reduce the dimensionality of the feature maps and extract the most important features to increase the robustness of the network. Max pooling and average pooling are two types of pooling layers that are commonly used.

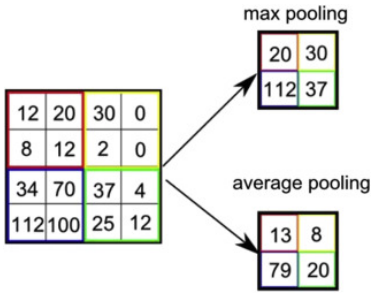


Figure 2.9: Visualization of max and average pooling layers [5]

Max Pooling selects the maximum value from a small region of the input data. This operation is used to capture the most important feature in the region, such as the strongest edge or the highest intensity value. Max-pooling is often used in image recognition tasks because it is able to extract the most salient features from the input image, such as edges and textures.

Average pooling, on the other hand, takes the average value from a small region of the input data. This operation is used to reduce noise and smooth out small variations in the input data. Average pooling is often used in tasks such as image segmentation because it is able to extract more subtle features from the input image, such as shapes and textures.

Both have different advantages and are used in different tasks. Many CNNs use both maximum and average pooling in different layers.

Fully connected layers are used to classify the features extracted by the convolutional layers. They take the output of the pooling layers and use it as input to a traditional feedforward neural network, which makes the final prediction.

CNNs have been successful in a wide range of image classification tasks and have been used to achieve state-of-the-art performance on benchmarks such as the ImageNet dataset. They have also been applied to other domains, such as video analysis and natural language processing.

One of the key advantages of CNNs is their ability to learn spatially hierarchical representations of the data. By using convolutional layers, CNNs can learn local patterns of the data and then compose these local patterns to form more complex patterns. This is in contrast to traditional feedforward neural networks, which use fully connected layers and are not able to take advantage of the spatial structure of the data.

Another advantage of CNNs is their ability to learn translation-invariant representations of the data. By using pooling layers, CNNs can learn representations that are robust to small translations of the input data. This is important for tasks such as image classification, where the object of interest may be in different locations within the image.

In summary, convolutional neural networks (CNNs) are a class of deep neural networks specifically designed to process data with a grid-like structure, such as images. CNNs are composed of multiple layers, including convolutional layers, pooling layers, and fully connected layers. CNNs have been successful in a variety of image classification tasks and have been used to achieve state-of-the-art performance on benchmarks such as the ImageNet dataset. They have also been applied to other domains such as video analysis and natural language processing. CNNs are capable of learning spatially hierarchical representations of the data and translation-invariant representations of the data.

2.3.10 Temporal convolutional neural networks

Temporal Convolutional Networks (TCNs) are a variation of CNNs that are designed to process sequential data, such as time series or videos. They are similar to traditional CNNs in that they use convolutional layers to extract features from the input data, but they also incorporate temporal information by using dilated convolutions.

Dilated convolutions are similar to standard convolutions, but they have a dilation factor that controls the spacing between the elements of the convolutional kernel. This allows TCNs to increase the receptive field of the network without increasing the number of parameters. As a result, TCNs can capture long-term dependencies in the input data.

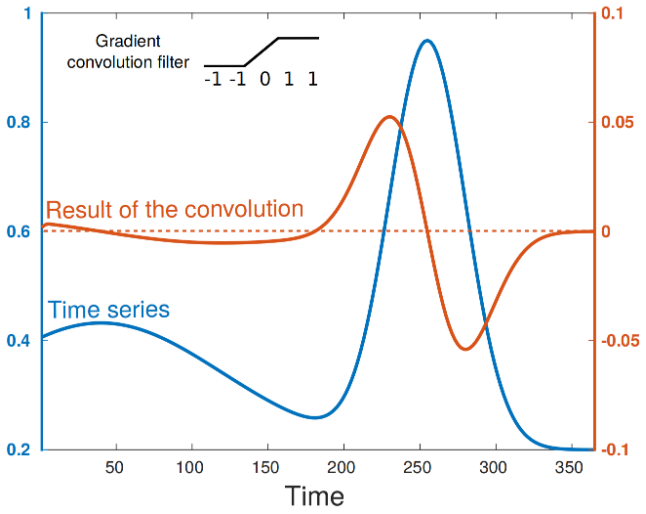


Figure 2.10: Convolution of a time series (blue) with the positive gradient filter [-1 -1 0 1 1] (black).The result (red) takes high positive values when the signal sharply increases, and conversely[44]

TCNs also use causal convolution, which means that the network's output depends only on past inputs. This is important in tasks such as prediction, where the future cannot be influenced by the current output.

TCNs have been used in a variety of tasks, including speech recognition, natural language processing, and time series forecasting. They have been shown to be effective in capturing temporal dependencies and have achieved state-of-the-art performance on benchmarks such as the Wall Street Journal corpus for speech recognition.

In summary, Temporal Convolutional Networks (TCNs) are a variant of CNNs designed to process sequential data, such as time series or video. They incor-

porate temporal information by using dilated convolutions and causal convolution, which allows TCNs to increase the receptive field of the network without increasing the number of parameters and to capture long-term dependencies in the input data. TCNs have been used in a variety of tasks and have been shown to be effective at capturing temporal dependencies, achieving state-of-the-art performance on benchmarks.

2.3.11 Recurrent neural networks

Recurrent Neural Networks (RNNs) [22, 20] are a type of neural network designed to process sequential data, such as time series or natural language. They are called "recurrent" because they have a feedback loop that allows information to flow from one step of the sequence to the next.

RNNs have a hidden state, which is a vector containing information about past inputs. At each time step, the input is combined with the hidden state to produce a new hidden state and an output. The hidden state is then used as the input for the next time step.

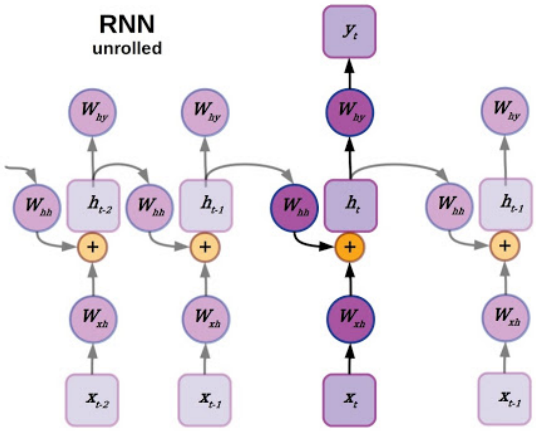


Figure 2.11: An unrolled representation of a Recurrent Neural Network (RNN) showing how it processes input sequences by feeding the output of the previous time step as an additional input to the current time step. This allows the RNN to capture the temporal dependencies and learn patterns in sequential data.[12]

One of the main limitations of traditional RNNs is the vanishing gradient problem, which occurs when the gradients of the parameters tend to decrease exponentially with the number of time steps. This makes it difficult to train RNNs on long sequences. To overcome this problem, variants of RNNs have been developed, such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit

(GRU).

LSTMs and GRUs are architectures that introduce gates that control the flow of information through the hidden state, allowing them to store information for longer periods of time.

The main difference between LSTMs and GRUs is the number and complexity of the gates they use to control the flow of information. LSTMs use three gates: an input gate, an output gate, and a forget gate. The input gate controls the flow of new information into the cell, the output gate controls the flow of information out of the cell, and the forget gate controls the flow of information through the cell.

GRUs, on the other hand, use two gates: an update gate and a reset gate. The update gate controls the flow of new information into the cell, and the reset gate controls the flow of information through the cell. The update gate can be seen as a combination of the input gate and the forget gate in LSTM.

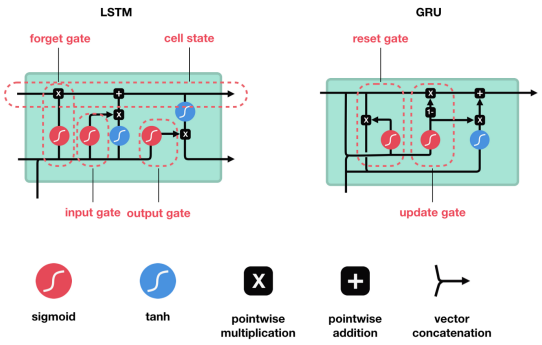


Figure 2.12: Visualization of LSTM and GRU cells [45]

In terms of performance, both LSTMs and GRUs have been shown to be effective at handling long-term dependencies in sequential data. LSTMs have been used in a wide range of applications, such as language modeling, speech recognition, and machine translation, and have been shown to achieve state-of-the-art performance in many tasks. GRUs, on the other hand, are more computationally efficient and have been used in applications such as natural language processing and speech recognition, and have been shown to achieve similar performance to LSTMs with fewer parameters.

In summary, Recurrent Neural Networks (RNNs) are a type of neural network designed to process sequential data. They have a feedback loop that allows information to flow from one step of the sequence to the next, and a hidden state that contains information about past inputs. However, traditional RNNs suffer from the vanishing gradient problem, which makes it difficult to train RNNs on long sequences. To overcome this problem, variants of RNNs such as

Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) have been developed. Both LSTMs and GRUs have been shown to be effective in solving a wide range of problems related to sequential data and have been applied to other areas as well.

2.3.12 Generative adversarial networks

Generative Adversarial Networks (GANs) [21, 40] are a type of neural network architecture that is used for unsupervised learning. They consist of two main components: a generator network and a discriminator network. The generator network is trained to generate new data samples that are similar to a given training dataset, while the discriminator network is trained to distinguish between the generated samples and the real samples from the training dataset.

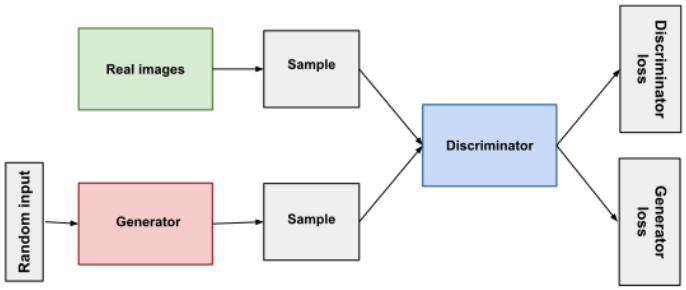


Figure 2.13: Overview of GAN structure [10]

The training process of GANs is based on a game-theoretic approach, where the generator and the discriminator compete against each other. The generator tries to generate samples that are similar to the real samples, while the discriminator tries to correctly identify which samples are real and which are generated. As training progresses, the generator gets better at generating realistic samples, while the discriminator gets better at identifying the generated samples.

One of the main advantages of GANs is that they can be used to generate high-quality images, videos, and other types of data. They have been used in a wide range of applications, including image synthesis, image-to-image translation, style transfer, and video prediction. GANs have also been applied to other areas such as natural language processing, speech synthesis, and 3D object generation.

There are many variations of GANs, such as Wasserstein GANs [3], which use the Wasserstein distance to measure the difference between the generated samples and the real samples, and Cycle GANs [63], which are used for image-to-image translation. There are also GANs that use different architectures, such as convolutional GANs and recurrent GANs.

Conditional GANs [41], also known as cGANs, are a variant of GANs where both the generator and the discriminator are conditioned on some additional input, such as a class label or an image. This allows the model to generate new samples that are conditioned on a specific class or attributes. For example, it can generate images of a particular object or in a particular style.

In summary, Generative Adversarial Networks (GANs) are a type of neural network architecture used for unsupervised learning. They consist of two main components: a generator network and a discriminator network. The generator network is trained to generate new data samples that are similar to a given training data set, while the discriminator network is trained to distinguish between the generated samples and the real samples from the training data set. GANs have many advantages, such as the ability to generate high-quality images, videos, and other types of data. Conditional GANs are a variant of GANs where both the generator and the discriminator are conditioned on some additional input, such as a class label or an image, allowing the model to generate new samples conditioned on a specific class or attributes. They have been used in a wide range of applications and have been applied to other areas such as natural language processing, speech synthesis, and 3D object generation.

2.3.13 Attention is all you need

The "Attention is All You Need" (AIAYN) [58] model, also known as the Transformer, is a neural network architecture introduced by Google researchers in 2017. It is a type of encoder-decoder model that is primarily used for natural language processing tasks such as language translation, text summarization, and question answering.

The key innovation of Transformer is the use of self-attention mechanisms, which allow the model to focus on specific parts of the input while processing it. Traditional encoder-decoder models use recurrent neural networks (RNNs) or convolutional neural networks (CNNs) to process the input sequentially, which can be slow and computationally expensive for long sequences. In contrast, the Transformer uses self-attention mechanisms to process all parts of the input simultaneously, making it much faster and more efficient.

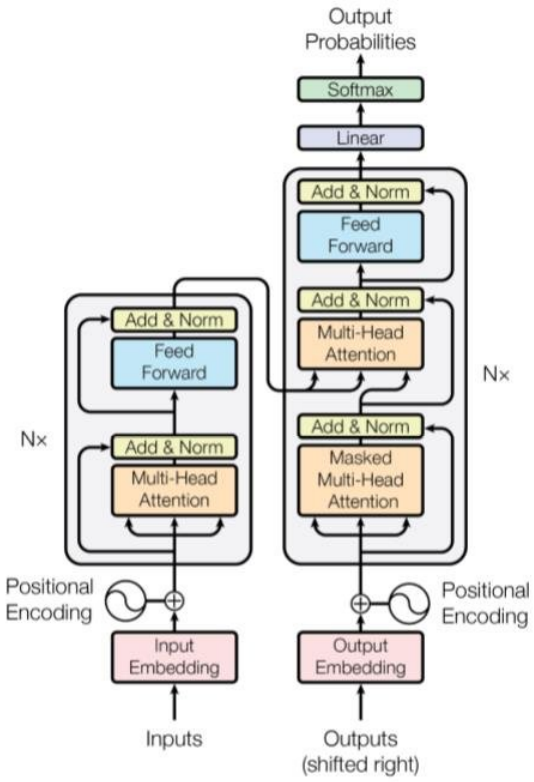


Figure 2.14: The transformer model [58]

The Transformer architecture consists of an encoder and a decoder, which are both made up of multiple layers of self-attention and feed-forward neural networks. The encoder takes in the input sequence and processes it using self-attention mechanisms, which allow the model to learn relationships between different parts of the input. The decoder then takes the encoded representation and generates the output sequence.

The attention mechanism used in the Transformer is a type of dot-product attention, which calculates the similarity between different parts of the input. The attention weights are then used to combine the different parts of the input to create a new representation.

One of the key advantages of the Transformer is its ability to handle input sequences of variable length, which is important for natural language processing tasks where the length of the input can vary widely. In addition, the Transformer can be parallelized, allowing it to be trained and run on multiple GPUs or TPUs, resulting in faster training times.

In conclusion, the Attention is All You Need (AIAYN) model, also known as the Transformer, is a neural network architecture introduced in 2017. The main innovation of the Transformer is the use of self-attention mechanisms, which allow the model to focus on certain parts of the input while processing it. The Transformer architecture consists of an encoder and a decoder, both of which consist of multiple layers of self-attention and feed-forward neural networks. The attention mechanism used in the Transformer is a type of point product attention that computes the similarity between different parts of the input. One of the main advantages of the Transformer is its ability to handle variable-length input sequences, which is important for natural language processing tasks where the length of the input can vary greatly. In addition, the Transformer can be parallelized, allowing it to be trained and run on multiple GPUs or TPUs, resulting in faster training times.

Dataset

3.1 Study Area

The study area, with an area of 100 km^2, is located in southeastern France, centered on the city of Balaruc-les-Bains, bordering the Mediterranean Sea. It is a small and densely populated urban area surrounded by agricultural crops, mainly vineyards, and natural vegetation. Figure 3.1 shows the GT map overlaid on the Sentinel-2 image taken on January 2019.

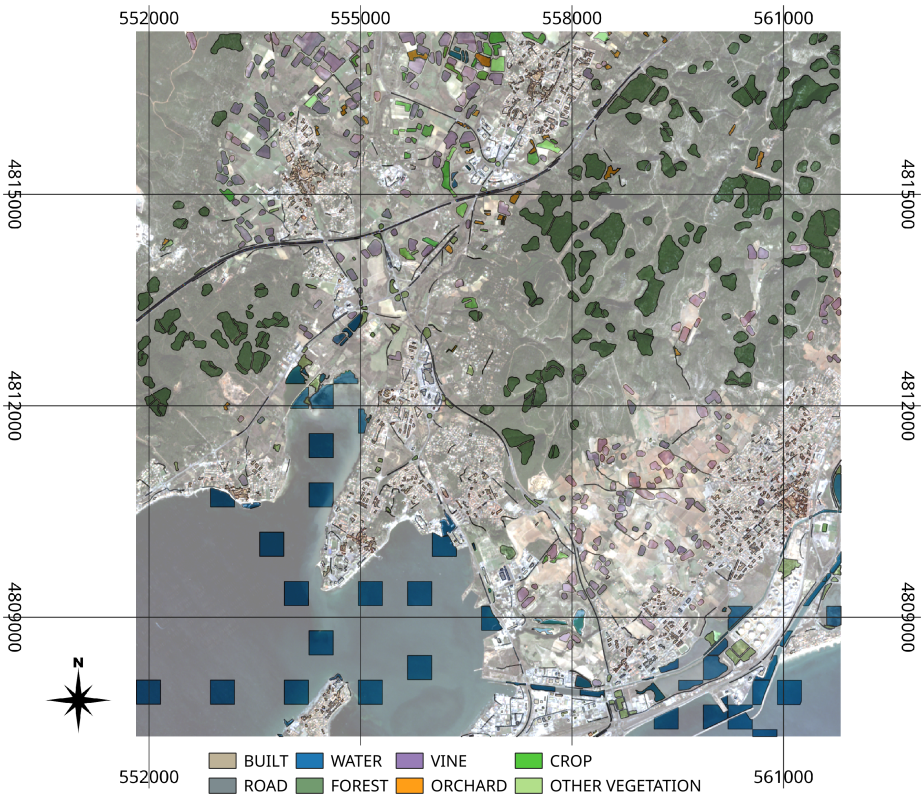


Figure 3.1: The Sentinel-2 image is overlaid with ground truth data.

3.2 Chronology

Satellite Image Time Series: A total of 54 Sentinel-2 image time series covering the year 2019 were collected for the study. The images were selected to represent the temporal (annual) evolution of the land cover associated with the study area and to filter out images that were heavily affected by cloud phenomena. The acquisition dates of the Sentinel-2 satellite image time series are shown in Figure 3.2.

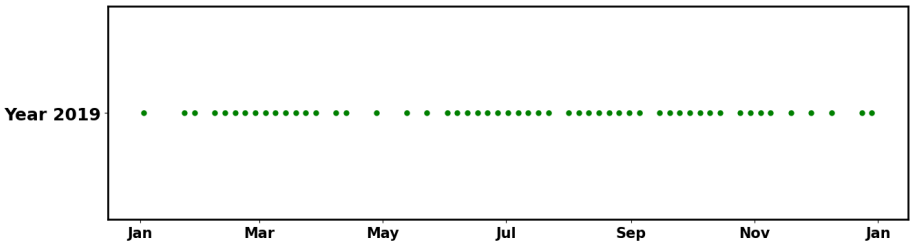


Figure 3.2: Acquisition dates of each Satellite Image Time Series.

3.3 Mask and Bands

The images used in this analysis were obtained from the THEIA pole platform[1] at level-2A in top of canopy reflectance values and with associated cloud masks.

For our classification task, we utilized a total of 16 bands, which included 10 bands with a spatial resolution of 10m and 20m. These bands covered the visible, near-infrared, and short-wave infrared ranges. Additionally, we derived 6 more bands from the initial 10 bands, bringing the total to 16.

Sentinel-2 Band	Description	Spatial resolution (m)
B2	Blue	10
B3	Green	10
B4	Red	10
B5	Vegetation Red Edge 1	20
B6	Vegetation Red Edge 2	20
B7	Vegetation Red Edge 3	20
B8	Near Infrared 1	10
B8A	Near Infrared 2	20
B11	Short Wave Infrared 1	20
B12	Short Wave Infrared 2	20

Table 3.1: Spectral bands for the Sentinel-2 sensors (10m and 20m).[54]

[1] `http://theia.cnes.fr`

Band	Description
$NDVI = \frac{B8-B4}{B8+B4}$	Normalized Difference Vegetation Index [50]
$NDWI = \frac{B3-B8}{B3+B8}$	Normalized Difference Water Index [39]
$BRI = \sqrt[2]{B2^2 + \cdots + B12^2}$	Brightness Index [29]
$MNDWI = \frac{B3-B11}{B3+B11}$	Modified NDWI [60]
$SWDVI = \frac{B11-B8}{B11+B8}$	Short Wave NDWI [16]
$NDRE = \frac{B8-B5}{B8+B5}$	Normalized Difference Red Edge Index [4]

Table 3.2: Derived Bands from Multispectral Data

Derived bands are new spectral bands that are created by combining or transforming the original bands. These derived bands can provide additional information about the surface features of interest that may not be captured by the original bands alone.

To replace cloudy pixel values, a preprocessing step was performed using multitemporal linear interpolation based on the available cloud masks (also known as temporal gap-filling[27]). The study area is located within the Sentinel-2 tile 31TEJ with relative orbit number 8.

The missing ratio in the dataset used for this study is 33.9%. It is important to note that this missing ratio refers specifically to cloud masks.

3.4 Ground Truth Data

The ground truth data was built from various sources: the *Registre Parcellaire Graphique* (RPG) reference data (the French land parcel identification system), the French National Geographic Institute *'BD-Topo & BD Forêt'* and the visual interpretation of a SPOT 6 image (to assess and enrich the GT data) as well. The GT was assembled in Geographic Information System (GIS) vector file, containing a collection of polygons, each attributed with a land cover category. Statistics about the GT data are reported in Table 3.3.

Class Name	Class ID.	# Polygons	# Pixels
Built	1	712	10,412
Road	2	328	5,165
Water	3	82	32,095
Forest	4	172	49,175
Vine	5	223	28,667
Orchard	6	32	2,075
Crop	7	39	5,205
Other Vegetation	8	56	4,812
	Total	1,644	137,606

Table 3.3: Number of polygons and pixels for each class.

Methods

To initiate the process of identifying the most promising models for time series classification, a comprehensive review of the current state of the art in the field was conducted. This involved a meticulous exploration of published research studies and academic papers related to the research question. Subsequently, the different approaches presented in the literature were thoroughly analyzed and compared, following the most promising models were shortlisted for further examination and testing.

Once the model selection phase was completed, I proceeded to conduct an in-depth study of the papers and any available source code. This was done to enhance their understanding of the underlying algorithms and techniques employed in each model, as well as any implementation details that may be relevant for adapting to the specific research needs.

As a result of these efforts, four models - Random Forest (RF), Adversarial Joint-Learning Recurrent Neural Network (AJ-RNN), Temporal Convolutional Neural Network (TempCNN), and Lightweight Temporal Self-Attention Encoder (L-TAE) - were evaluated in this study, and their details are presented in this section.

4.1 Random forest

TensorFlow Decision Forests (TF-DF) [23] is the library used to train and evaluate the random forest model.

A Random Forest [6] is a collection of deep CART decision trees trained independently and without pruning. Each tree is trained on a random subset of the original training dataset (sampled with replacement). The algorithm is robust to overfitting, even in extreme cases e.g. when there are more features than training examples. It is probably the most well-known of the Decision Forest training algorithms.

4.2 TempCNN

The article "Temporal Convolutional Neural Network for the Classification of Satellite Image Time Series" [44] presents a machine learning model for classifying satellite image time series data. The model is based on Convolutional Neural Networks (CNNs) and aims to improve traditional image classification methods by incorporating time series information into the model.

The Temporal Convolutional Neural Network (TempCNN) architecture was used in the following experiments to classify satellite image time series.

The model inputs a series of satellite images and applies a series of convolutional and pooling operations to extract high-level features from the data. The paper presents a novel approach to classify satellite image time series data and highlights the potential applications of the model in areas such as remote sensing and environmental monitoring.

4.2.1 Temporal Convolutions

Convolutional layers have been proposed as a technique to limit the number of weights a network must learn while exploiting structural dimensions in the data, such as spatial, temporal, or spectral dimensions [33]. These layers apply a convolution filter to the output of the previous layer, resulting in an activation map as output, rather than a single activation value per neuron as in dense (fully connected) layers. For example, with a univariate time series as input, the output of the convolutional layer would be a new time series, with each data point generated by the corresponding convolution filter applied to the original series.

Convolutional layers have the property of sharing their parameters across locations. This characteristic involves applying the same linear combination by sliding it over the input, resulting in a significant reduction in the number of weights in the layer. This reduction is based on the assumption that the same convolution can be beneficial in different parts of the time series. Therefore, the number of trainable parameters depends only on the filter size of the convolution and the number of units, but not on the input size.

The output size, on the other hand, depends on the input size, the stride, and the padding. The stride controls the interval between two convolution centers, while padding controls the addition of values (usually zeros) at the beginning

and end of the input series before computing the convolution. Padding can guarantee that the output is the same size as the input.

4.2.2 Model

The baseline architecture of TempCNN used for the experiments, as shown in Figure 4.1, consists of three convolutional layers (64 units), one dense layer (256 units), and one softmax layer. In the experimental section, we will investigate the width (i.e., number of units) of the convolutional layers, the depth (i.e., number of convolutional layers), and the pooling layers of the network.

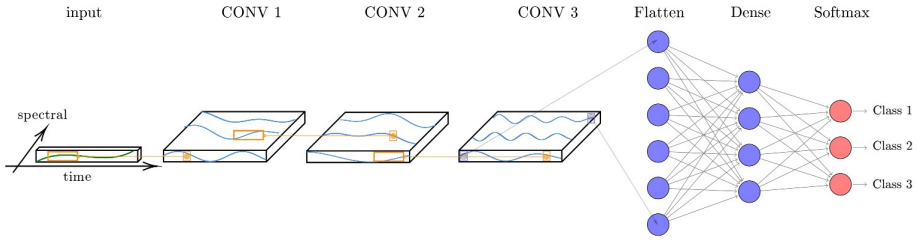


Figure 4.1: Proposed temporal Convolutional Neural Network (TempCNN). The network input is a multi-variate time series. Three convolutional filters are consecutively applied, then one dense layer, and finally the Softmax layer, that provides the predicting class distribution. [44]

To prevent overfitting, we employed the same regularization mechanisms described in [44]:

- dropout rate of 0.5 [57].
- L2-regularization on the weights (also named weight-decay) applied for all the layers with a small rate of 10^{-6}
- batch normalization [30].

To train the network Adam optimization was used with the standard parameter values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $e = 10^{-8}$) [32] We used a batch size of 32, and a maximum number of epochs set to 20, with an early stopping mechanism with a patience of zero on the validation loss.

4.3 AJ-RNN

"Adversarial Joint-Learning Recurrent Neural Network for Incomplete Time Series Classification" [38] is a research paper that proposes a novel approach for classification of incomplete time series data. The authors begin by highlighting the challenges of working with incomplete time series data, particularly the difficulty of extracting features and the need to deal with missing values.

To address these challenges, the authors propose an adversarial joint-learning recurrent neural network (AJ-RNN) that uses a recurrent neural network (RNN) to capture the temporal dependencies in the time series data, and an adversarial learning approach to impute the missing values.

The AJ-RNN is trained using a joint optimization framework that alternates between training the RNN for classification and the imputation network to fill in the missing values. The adversarial component of the imputation network is used to ensure that the imputed values are as close as possible to the true values.

The authors evaluate the performance of the AJ-RNN on several real-world datasets and demonstrate that it outperforms several existing state-of-the-art approaches for time series classification.

4.3.1 Adversarial learning

Previous studies have shown that adversarial approaches outperform traditional methods in generating data that conforms to the distribution of a given dataset [21, 34].

Furthermore, GANs have shown promising results in filling in missing data in time series prediction [62, 35, 37]. Similarly, adversarial techniques have also been applied to tasks such as video captioning [61] and domain adaptation [15]. However, prior to this paper, the question of how to apply adversarial learning in the domain of incomplete time series classification (ITSC) has not been explored.

The integration of adversarial training and joint learning in recurrent neural networks (RNNs) is explored in this paper, resulting in the development of a system called Adversarial Joint learning RNN (AJ-RNN).

The study's results show that employing AJ-RNN, an end-to-end framework integrating adversarial training and joint learning in recurrent neural networks, is an effective solution to tackle the issue of incomplete time series classification. AJ-RNN is trained to predict the value of the next input variable when it is revealed, and to fill in the missing value with its prediction. At the same time, AJ-RNN also learns to classify. Hence AJ-RNN can directly perform classification with missing values.

4.3.2 Model

The following section presents the Adversarial Joint learning Recurrent Neural Network (AJ-RNN) as a solution for time series classification with missing values. The architecture of the model is illustrated in Figure 4.2.

Here we describe how adversarial and joint learning strategies are integrated into an RNN.

The time series X is represented as a sequence vector of T observations, denoted by $X = \{x_1, x_2, ..., x_T\}$. Each observation $x_t \in R^d$ is a d-dimensional vector.

Assume that a time series X has missing values which are represented by a T-dimensional mask vector $M = \{m_1, m_2, ..., m_T\}$. The elements m_t of the mask vector are binary values indicating the presence or absence of the corresponding element x_t in the time series, where m_t takes the value of 1 if x_t is revealed and 0 if x_t is missing.

Each time series X^i in the dataset D is associated with a target label $y^{(i)}$, and a mask vector M^i, where $D = (X^i, M^i, y^i)_{i=1}^N$.

To avoid the traditional two-step approach, imputation and classification are regarded as two tasks in multitask learning.

The AJ-RNN is trained to approximate the value of the next input variable and use it as a target when it is revealed. If the next value is missing, it is filled in with the current prediction. At the same time, the AJ-RNN is trained to perform the classification task.

The missing value problem is addressed through two processes, namely approximation and imputation. As shown in Figure 4.2, two kinds of links enable AJ-RNN to directly model time series in the presence of missing values: dashed blue links (for approximation) and solid blue links (for imputation). The system is trained to approximate the next value x_t using the last hidden state h_{t-1} as follows:

$$\hat{x_t} = W_{imp} h_{t-1} + b_z \tag{4.1}$$

where $W_{imp} \in R^{n \times m}$ is a learned regression matrix and b_z is a bias term. As $\hat{x_t}$ is trained to approximate the next value x_t, it can be employed for imputing it when it is missing. The input value u_t is computed as:

$$u_t = m_t \odot x_t + (1 - m_t) \odot \hat{x_t} \tag{4.2}$$

where m_t is the mask as defined above and $\odot$ is the element-wise product.

The completed input value u_t is used to train the RNN. The update equation of the RNN is:

$$h_t = F_{RNN}(h_{t-1}, u_t; W) \tag{4.3}$$

Where h_t represents the hidden unit vector at time t, W encapsulates the input-to-hidden and hidden-to-hidden parameters, and F_{RNN} represents the update function of the particular RNN variant.

To obtain the probability distribution over each category label, a softmax is applied to the output of the classifier, which takes the last hidden state of the RNN, h_T, as input.

$$P(\hat{y_j}|h_T) = \frac{exp(W_j^T h_T)}{\sum_{l=1}^{K} exp(W_l^T h_T)} \tag{4.4}$$

where K is the number of class labels and $\{W_l\}_{l=1}^{K}$ are the class-specific weights of the softmax layer. More complex networks can be used for the classifier based on the specific task at hand. However, in this study, a simple classifier is used.

During joint learning, imputation and classification tasks are performed. Let D be the time series dataset and let the superscript i denote the i-th sample in the dataset. The imputation task produces an imputation sequence vector $\hat{X}^i = \hat{x}_2^i, ..., \hat{x}_T^i$ for the i-th time series sample. This vector is composed of two parts: approximation values (represented by orange units in Fig. 4.2) and imputation values (represented by purple units in Fig. 4.2). The imputation loss of all time series samples is calculated on the approximation values as follows:

$$\mathcal{L}_{imp}(X, \hat{X}, M) = \frac{1}{N} \sum_{i=1}^{N} \|(X_{2:T}^i - \hat{X}^i) \odot M_{2:T}^i\|_2^2 \tag{4.5}$$

where N represents the number of samples in the dataset. Equation (4.5) measures the mean squared error loss between the approximation and the revealed values. The mask $M_{2:T}^i$ is used to ignore the imputation values in the loss calculation, as there is no ground truth available for these missing values.

The loss for the classification task can be calculated as follows. Let y^i be the true label of the i-th time series sample and $\hat{y}^i$ be the predicted probability distribution given by Equation (4.4).

$$\mathcal{L}_{cls}(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{K} 1\{y^i = j\} \log \hat{y}^i \tag{4.6}$$

where K represents the number of class labels. Equation (4.6) is the softmax cross entropy loss of the predicted label and the true label.

The end-to-end framework is susceptible to the negative impact of missing values, which can propagate errors from the imputation task to the classification task. The imputed values are predictions and are prone to errors, which can quickly amplify as they are fed into the RNN, leading to the problem of exploding bias.

Here, a discriminator D is introduced to alleviate the negative impact of missing values. Unlike the traditional approach that identifies the entire completed

vector, the discriminator D distinguishes whether each value in the completed vector is real or imputed. This direct supervision of the imputed values by the discriminator D can help reduce the error propagation from imputation to classification.

Specifically, for each training sample i in the time series dataset, there exists a completed sequence vector $U^i = u_2^i, ..., u_T^i$ obtained through Equation (4.2), along with its corresponding mask vector $M_{2:T}^i = m_2^i, ..., m_T^i$.

In U^i, some are real values, while the rest are imputed. We know which are which by the mask vector M^i. We can take advantage of this knowledge to provide a supervision signal for the imputation network.

The adversarial learning strategy involves two players in a minimax game: the discriminator D and the AJ-RNN [1]. The parameters of both models are updated alternately. Initially, the discriminator D takes the completed sequence vector as input and is trained to distinguish between the revealed and imputed values in the vector. The discriminator loss can be defined as follows:

$$\mathcal{L}_D(U, M) = -[E \log(D(X_{real})) + E \log(1 - D(\hat{X}_{imp}))] \tag{4.7}$$

$$= -\frac{1}{N} \sum_{i=1}^{N} \left[M_{2:T}^i \odot \log(D(U^i)) + (1 - M_{2:T}^i) \odot \log(1 - D(U^i)) \right] \tag{4.8}$$

Here, the function $D(\cdot)$ takes in the completed sequence vector as input and outputs the estimated mask probability $\hat{P}$ of the discriminator.

In Equation (4.7), the first term is the log output of the discriminator on real values, and the discriminator tries to maximize this to 1. The second term is the loss for imputed values. Hence the discriminator tries to minimize its output for imputed values.

The discriminator is designed as a neural network composed of three fully connected layers to leverage global contextual information from the input sequence. The first hidden layer has T units, which is equal to the length of the input sequence, the second hidden layer has $T/2$ units, and the third hidden layer has T units. The activation function used in each layer is the hyperbolic tangent (tanh) except for the output layer, where the sigmoid activation function is used to obtain the estimated probability of each value being real or fake.

The goal of the RNN is to minimize the difference between the distribution of predicted values and the distribution of revealed ones by deceiving the discriminator D. This provides a supervisory signal to the imputation network to produce better imputed values.

[1] RNN and Classifier

The adversarial loss of the RNN can be defined as follows:

$$\mathcal{L}_{adv}(U, M) = \frac{1}{N}\sum_{i=1}^{N}(1 - M^i_{2:T}) \odot \log(1 - D(U^i)) \tag{4.9}$$

Hence the AJ-RNN tries to maximize the discriminator output for imputed values. This provides a supervisory signal for each imputed value in the completed sequence vector, which can reduce the bias introduced by the imputation operation and thus alleviate the exploding bias problem. Note that we can also feed the whole imputation vector $\hat{X}^i$ into the discriminator, providing supervision for both the approximated and imputed values. However, in practice, using only the imputed values is more computationally efficient and yields similar results as using both approximated and imputed values. Hence, considering computational efficiency, Equation (4.9) was we adopted as the adversarial loss.

Finally, the overall training loss of AJ-RNN is defined as follows:

$$\mathcal{L}_{AJ-RNN} = \mathcal{L}_{cls} + \mathcal{L}_{imp} + \lambda_d \mathcal{L}_{adv} \tag{4.10}$$

where λ_d is hyper-parameter. This forms an end-to-end training framework for incomplete time series classification.

AJ-RNN combines the merits of joint learning and adversarial learning. The discriminator D is trained with the revealed values and the mask vector effectively provides supervision on each imputed value. Therefore, the negative impact of missing values on AJ-RNN is reduced.

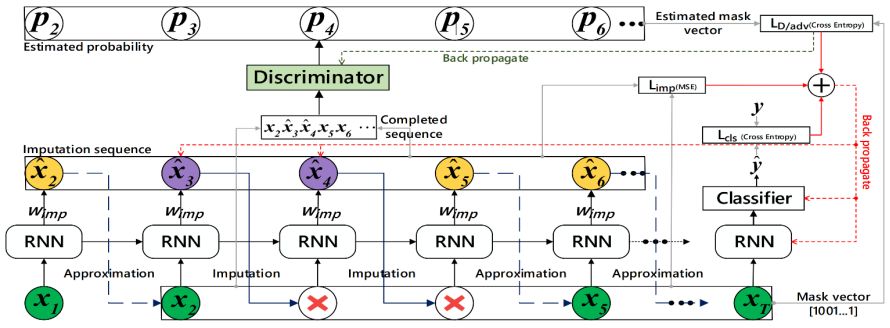


Figure 4.2: The figure shows the proposed AJ-RNN framework. The green units represent revealed inputs, yellow for the output of approximated values, purple for the imputed values, and a red "X" for missing inputs. The dashed links represent the approximation training, while the solid links represent imputation. The discriminator receives a completed vector composed of revealed and imputed values as input, which provides a one-to-one supervisory signal for imputed values $\hat{x_3}$ and $\hat{x_4}$.

4.4 L-TAE

The paper "Lightweight Temporal Self-Attention for Classifying Satellite Image Time Series" [18] was written by Vivien Sainte Fare Garnot and Loic Landrieu and published in 2020.

The authors present a new deep learning model for classifying satellite image time series using a modified version of the Temporal Attention Encoder.

In their proposed network, the channels of temporal inputs are distributed among several attention heads operating in parallel. These heads extract specialized temporal features, which are then concatenated into a single representation. The authors show that their approach achieves superior performance compared to other state-of-the-art time-series classification algorithms on an open-access satellite image dataset, while using significantly fewer parameters and reducing computational complexity.

Overall, the paper presents a novel method for classifying satellite image time series that utilizes a lightweight variant of temporal self-attention and outperforms other state-of-the-art approaches.

4.4.1 Multi-Headed Self-Attention

The original version of self-attention, originally developed for text translation as described in [58], involves three main steps. First, for each position t in the input sequence, a key-query-value triplet is computed, denoted as $k^{(t)}$, $q^{(t)}$, and $v^{(t)}$, respectively, by applying a shared linear layer to the input $e^{(t)}$. Second, attention masks are computed representing the compatibility (dot product) between the queries and the keys of previous elements in the sequence. Finally, an output is assigned to each position in the sequence, which is the sum of the previous values weighted by the corresponding attention mask.

To allow each head to specialize in detecting specific features of the feature vectors, the self-attention process is performed in parallel for H sets of independent parameters or heads, and the outputs are concatenated. This approach is used by Rußwurm et al. [51] to embed sequences of satellite observations by max-pooling the resulting sequence of outputs in the temporal dimension.

Garnot et al. [19] introduce a modified self-attention scheme called the Temporal Attention Encoder (TAE). First, they propose to use the input embeddings directly as values (i.e., $v^{(t)} = e^{(t)}$), which takes advantage of the end-to-end training of the image embedding functions along with the TAE. They also define a single master query $\hat{q}$ for each sequence, which is computed from the temporal average of the queries. The master query is compared to the key sequence to generate a single attention mask of dimension T, which is used to weight the temporal average of the values into a single feature vector.

4.4.2 Model

Their work is an extension of previous efforts to adapt multi-head self-attention for sequence embedding. The primary goal is to optimize efficiency, especially with respect to the number of parameters and the computational load.

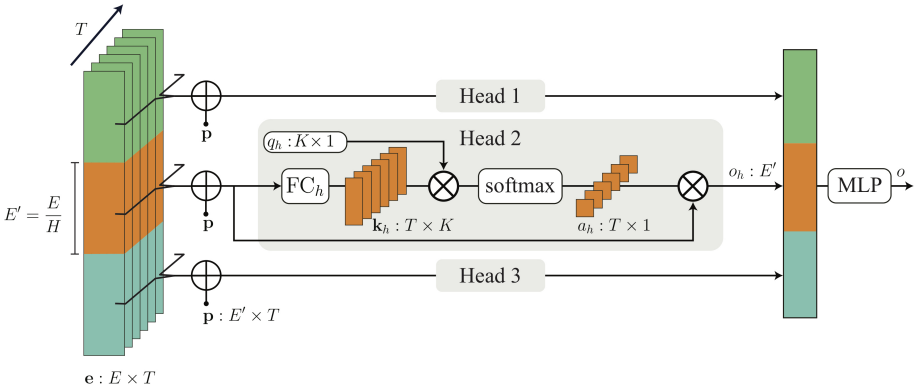


Figure 4.3: Light Temporal Attention Encoder (L-TAE) module processing an input sequence e of T vectors of size E, with $H = 3$ heads and keys of size K. The channels of the input embeddings are distributed among heads. Each head uses a learnt query $\hat{q_h}$, while a linear layer FC_h maps inputs to keys. The outputs of all heads are concatenated into a vector with the same size as the input embeddings, regardless of the number of heads [18]

Channel Grouping The proposal is to divide the E channels of the input elements into H groups of size $E' = E/H$ following the approach of Wu et al. [59], where H refers to the number of heads. The groups of input channels for the h-th group of the t-th element of the input sequence (4.11) are denoted by $e_h^{(t)}$.

To encode the number of days elapsed since the beginning of the sequence, an E'-dimensional positional vector p with a characteristic scale $\tau = 1000$ is used (4.12). To enable each head to access this information, the vector p is replicated and added to every channel group. This approach allows each head to work in parallel on its corresponding group of channels, reducing the computational expense of calculating keys and queries. Additionally, it allows each head to specialize alongside its channel group and avoid redundant operations between heads.

Query-as-Parameter The K-dimensional master query q_h of each head h is defined as a model parameter instead of the results of a linear layer. This approach has the immediate advantage of reducing the number of parameters required. Although this method lacks flexibility, it is compensated by the greater number of heads available.

Attention Masks The approach involves using a learned linear layer (4.13) solely for obtaining the keys, bypassing the values ($v^{(t)} = e^{(t)}$), and employing model parameters for the queries. For each head h, the attention masks $a_h \in [0,1]^T$ are obtained by scaling the softmax of the dot product between the keys and the master query (4.14). The outputs o_h of each head are determined by summing the corresponding inputs weighted by the attention mask a_h along the temporal dimension (4.15). Following this step, the outputs of each head are concatenated to form a vector of size E, which is then processed through a multi-layer perceptron MLP to achieve the desired size (4.16).

A schematic representation of the network is provided in Figure 4.3. In summary, the various steps of the L-TAE method can be summarized by the following operations for $h = 1...H$ and $t = 1...T$:

$$e_h^{(t)} = e^{(t)}[(h-1)E'...hE'] \tag{4.11}$$

$$[p^{(t)}] = sin(day(t)/\tau^{\frac{i}{E'}}) \tag{4.12}$$

$$k_h^{(t)} = FC_h(e_h^{(t)} + p^{(t)}) \tag{4.13}$$

$$a_h = softmax\Big(\frac{1}{\sqrt{K}}\Big[[q_h \cdot k_h^{(t)}\Big]_{t=1}^{T}\Big) \tag{4.14}$$

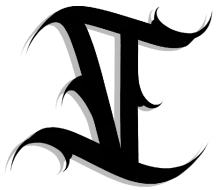
$$o_h = \sum_{t=1}^{T} a_h[t](e_h^{(t)} + p^{(t)}) \tag{4.15}$$

$$o = MLP([o_1, ..., o_H]). \tag{4.16}$$

Spatio-temporal classifier The L-TAE temporal encoder is designed to be trained together with a spatial encoding module and a decoder module in an end-to-end manner (4.17).

The spatial encoder S is responsible for mapping a sequence of raw inputs $X^{(t)}$ to a sequence of learned features $e^{(t)}$, computed independently at each position of the sequence. On the other hand, the decoder D is responsible for mapping the output o of the L-TAE to a target vector y, which can be class logits in the case of a classification task.

$$\Big[X^{(t)}\Big]_{t=1}^{T} \xrightarrow{S} \Big[e^{(t)}\Big]_{t=1}^{T} \xrightarrow{L\text{-}TAE} o \xrightarrow{D} y \tag{4.17}$$



Tools

In this section, we focuses on the tools and frameworks used to implement the models and conduct the experiments.

5.1 Weights and Biases

Weights and Biases (W&B) [36] is a machine learning experiment tracking platform that allows users to log and compare the results of different experiments. In our research, we used W&B to track experiments and compare the results of different models.

W&B provides an intuitive experiment tracking interface that allows users to easily organize and compare results from different runs. It also provides useful features such as visualization tools, automatic logging of metrics and hyperparameters, and integration with popular machine learning frameworks such as TensorFlow and PyTorch.

We used W&B to log metrics and hyperparameters such as accuracy, learning rate, batch size, and more. This allowed us to easily compare the results of different models and identify which hyperparameters had the most significant impact on performance.

In addition to logging metrics and hyperparameters, W&B also allowed us to visualize the results in a variety of ways. We used the platform's dashboard to plot and compare the different metrics across experiments.

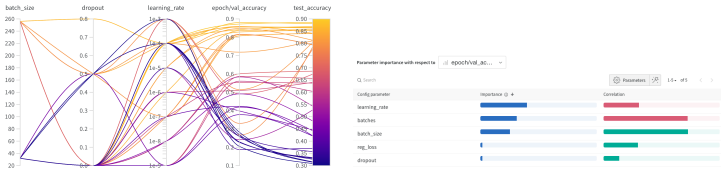


Figure 5.1: W&B panels showing parallel coordinates plot of hyperparameters and metrics on the left, and parameter importance plot on the right.

To facilitate hyperparameter tuning, W&B provides a powerful tool called Sweeps. This feature allows an efficient and automated search of the hyperparameter space, where the user defines a range of values for each hyperparameter and W&B does the rest. You can add multiple agents to run the sweeps in parallel, which can significantly reduce the time required to complete the search. In addition, the results of the sweeps can be easily visualized and compared, providing valuable insight into the impact of different hyperparameters on the performance of the model.

```yaml
method: random
metric:
  goal: maximize
  name: epoch/val_accuracy
parameters:
  G_epoch:
    value: 1
  batch_size:
    value: 32

  # ... more hyper-parameters

  hidden_size:
    value: 128
  learning_rate:
    values:
      - 1e-05
      - 1e-06
      - 1e-07
      - 1e-08
      - 1e-09
program: train.py
```

Figure 5.2: Example of Sweep configuration

Overall, W&B proved to be an essential tool for our project, allowing us to keep track of the experiments and compare the results of the different models. Its intuitive interface and powerful visualization tools made it easy to use and allowed us to quickly gain insight into the performance of the models.

5.2 Frameworks

With the exception of the L-TAE model, which was implemented using PyTorch [43], all other models in this study were implemented using TensorFlow [1].

TensorFlow is a popular and widely used open source machine learning framework developed by Google, which provides an efficient and flexible way to build and train neural networks.

PyTorch, on the other hand, is another popular open-source machine learning framework developed by Facebook that provides dynamic computational graphs and a more python-like way to build and train neural networks, making it particularly suitable for research purposes.

Using both TensorFlow and PyTorch to implement the models was a challenging task, but it provided an opportunity to improve my skills in working with different deep learning frameworks.

Experiments

In this chapter, we delve into the experiments conducted to optimize the performance of our models. To achieve the best possible results, we thoroughly explored the hyperparameter space, testing various parameters such as learning rate, batch size, activation functions, and regularization techniques. Additionally, we experimented with modifications to the architecture of the networks based on the results reported by the original authors. This process allowed us to identify the optimal settings for each model on our specific dataset. Although we encountered challenges such as overfitting and convergence problems, we overcame them through careful experimentation and iteration.

Furthermore, we address the impact of missing data on the performance of our models in time series classification. We implemented two different approaches to handle missing values: one involved replacing the missing values with zeros, while the other used pre-imputed values based on linear multi-temporal interpolation [27]. By comparing the performance of the models with both methods, we gained a better understanding of how missing data affects model accuracy.

6.1 Data preparation

To ensure a comprehensive and unbiased comparison, the same dataset and evaluation metrics were used for all machine learning models in this thesis. Specifically, the dataset of choice was the Satellite Image Time Series (SITS) dataset, which contains a total of 137,606 time series of satellite images covering natural and semi-natural areas. Since the data available for this task is

relatively limited, k-fold cross validation with $k = 5$ and varying seeds was implemented to ensure that the results were not affected by the partitioning of the dataset. Additionally, to ensure a balanced distribution of classes within each fold, the dataset was partitioned into training/validation/test sets with equal representation of each class. In order to prevent spatial correlation, we avoided placing identical polygon pixels within the same sets. This precaution was taken because disregarding spatio-temporal autocorrelation [17] may result in biased and unreliable outcomes.

The distribution of polygons and pixels for each class in the training, validation, and test sets can be observed in Table 6.1. The splitting was done in a ratio of 60:20:20 for the training, validation, and test sets respectively. The table provides a comprehensive summary of the number of polygons and pixels for each class in each of the sets.

	Train			Validation			Test		
Class	Pol.	Pix.	%	Pol.	Pix.	%	Pol.	Pix.	%
1	427	6486	7	142	2266	8	143	1660	6
2	196	3369	3	65	803	3	67	993	3
3	49	21298	24	16	5308	20	17	5489	21
4	103	30631	35	34	8194	31	35	10350	40
5	133	16933	19	44	6362	24	46	5372	20
6	19	1428	1	6	309	1	7	338	1
7	23	3399	3	7	948	3	9	858	3
8	33	2595	3	11	1455	5	12	762	2
Tot	983	86139	100	325	25645	100	336	25822	100

Table 6.1: Distribution of classes, polygons and pixels for each dataset.

6.2 Random forest

We need to adjust the dataset because the Random Forest model does not accept time series as input.

	b_0	b_1	...	b_{15}
t_0	0.2	1.6	...	1.7
t_1	1.3	1.8	...	1.8
⋮	⋮	⋮		⋮
t_{53}	0.6	0.4	...	1.3

(a) One observation in a 2-dim array

$t_0 :: b_0$	0.2
$t_0 :: b_1$	1.6
⋮	⋮
$t_{53} :: b_{15}$	1.3

(b) One observation in a 1-dim array

Figure 6.1: Transformation of one observation from a 2-dim array to a 1-dim array

As shown in Figure 6.1a, each sample is initially represented by a 2-dimensional array of size (54, 16), where 16 bands capture the characteristics of the pixel and 54 time steps reflect its evolution. However, the transformation process transforms the representation into a 1-dimensional array of size (864, 1), as shown in Figure 6.1b.

The training of the model uses the new representation of the data, composed of 864 inferred features, to generate 300 trees. Table 6.2 shows the results of the Random Forest model trained with and without pre-imputed missing values. The table includes the number of nodes, the number of features used, and the overall accuracy for each case.

Case	Nodes	Used features	Overall Accuracy
Pre imputation	523,636	753	91.03 ± 0.42
No imputation	519,932	718	89.72 ± 0.84

Table 6.2: Random forest results

6.3 TempCNN

In this section, we present the results of our experiments investigating the effectiveness of the TempCNN model on time series data from satellite imagery. We investigated the effect of the network depth and the width of its convolutional layers on the classification accuracy. Furthermore, we investigated the effect of the inclusion of pooling layers and the filter size on the classification accuracy. Finally, we investigated the effect of batch size on classification accuracy.

By thoroughly investigating these factors, we gained insight into how to optimize the TempCNN model for this type of data.

6.3.1 Data preparation

To normalize the data, we used min-max normalization, the same used in [44]. The traditional min-max normalization performs a subtraction of the minimum, then a division by the range, i.e., the maximum minus the minimum [24]. This normalization is very sensitive to extreme values, so we used the 2% (or 98%) percentile instead of the minimum (or maximum) value. For each feature, both percentile values are extracted from all timestamp values.

6.3.2 Experimental results

Depth In a convolutional neural network (CNN), depth refers to the number of layers in the network. A deeper network can learn more complex features by using a hierarchy of layers, each layer building on the features learned by the previous layer. However, increasing the depth can also lead to the vanishing gradient problem, where the gradients become too small to effectively update the weights during training. Therefore, finding the optimal depth is a critical consideration when designing a CNN.

To investigate the effect of network depth on model performance while keeping complexity constant, we vary the number of layers in the network while reducing the number of units in deeper layers. We experiment with six architectures consisting of between one and six convolutional layers, each with a varying number of units ranging from 256 to 16. Additionally, each architecture includes a dense layer with a number of units ranging from 64 to 2048. In each experiment, we train the model twice: first using the dataset with pre-imputed missing values, and then using a modified dataset where all missing values are replaced with zeros. As shown in the Table 6.3, the highest accuracy is achieved with two or three convolutional layers.

Model			No imputation	Pre imputation
1CONV256	+	1FC64	91.39 ± 0.74	91.12 ± 0.57
2CONV128	+	1FC128	91.32 ± 1.11	91.32 ± 1.28
3CONV64	+	1FC256	$\mathbf{92.57 \pm 0.84}$	$\mathbf{92.93 \pm 1.50}$
4CONV32	+	1FC512	92.33 ± 1.32	90.45 ± 1.98
5CONV16	+	1FC1024	91.20 ± 0.94	88.20 ± 2.95
6CONV8	+	1FC2048	87.83 ± 2.89	88.58 ± 2.90

Table 6.3: Influence of depth on classification accuracy.

Width The width of a convolutional neural network (CNN) refers to the number of neurons in each layer. Increasing the width can enhance the network's ability to learn complex features because more neurons are available for training. However, a network that is too wide may be prone to overfitting, where it memorizes the training data rather than generalizing to new data. Therefore,

finding the optimal width is critical for achieving the best performance on the test data.

To evaluate the impact of width on model performance, we experimented with seven CNN architectures. Each architecture includes three convolutional layers, one dense layer with 256 neurons, and a Softmax layer, as illustrated in Figure 4.1. The architectures differ in the number of parameters, and we vary the width of the convolutional layers from 16 to 1024 neurons.

Model			No imputation	Pre imputation
3CONV16	+	1FC256	92.44 ± 0.83	91.62 ± 0.41
3CONV32	+	1FC256	92.52 ± 0.68	92.16 ± 0.68
3CONV64	+	1FC256	92.45 ± 0.89	$\mathbf{93.47 \pm 0.63}$
3CONV128	+	1FC256	92.26 ± 1.84	92.13 ± 1.35
3CONV256	+	1FC256	91.66 ± 1.96	92.04 ± 1.67
3CONV512	+	1FC256	91.97 ± 1.23	92.75 ± 1.73
3CONV1024	+	1FC256	$\mathbf{92.76 \pm 1.46}$	93.39 ± 1.54

Table 6.4: Influence of width on classification accuracy.

The results reported in Table 6.4 show that the architecture is remarkably robust, even when there is considerable variation in the number of neurons. This is illustrated by the fact that the overall accuracy shows a difference of only 1.7% between the model with lower parameters and the one with higher parameters when using pre-imputed data, while the difference is only 0.32% when there is no imputation for missing values. Although the standard deviation increases with the number of parameters, the architecture consistently achieves high accuracy rates, indicating that it is capable of maintaining good performance even with a larger number of neurons.

We chose to use the model with three convolutional layers of 64 neurons each and a dense layer of 256 neurons for the upcoming experiments because it offers a favorable balance between bias and variance. This choice was made considering the size of our training dataset, which consists of 80,000 samples.

Batch size In machine learning, batch size refers to the number of training examples used in a gradient descent iteration. Batch size plays a critical role in determining the efficiency and accuracy of the training process. A larger batch size can result in faster training because the algorithm can process more examples in each iteration. However, using a larger batch size can also result in a less accurate model because it can cause the optimization algorithm to converge to a suboptimal solution. On the other hand, using a smaller batch size may result in slower training, but may help the algorithm converge to a better solution. Thus, choosing an appropriate batch size is an important consideration in machine learning.

We conducted experiments to determine the optimal batch size for training the TempCNN model. Specifically, we tested four different batch sizes: 16, 32, 64, and 128. Our analysis, as presented in Table 6.5, shows that batch size has a noticeable impact on training time; larger batch sizes result in faster training times. However, we observed that the batch size does not have a significant effect on the overall accuracy of the TempCNN model. In fact, we found that the accuracy values for all four batch sizes are comparable, indicating that selecting an appropriate batch size is not a critical factor in achieving high accuracy for this model.

Batch size	Train time	No imputation	Pre imputation
16	52min	91.92 ± 1.50	92.16 ± 1.75
32	42min	$\mathbf{91.61 \pm 1.19}$	$\mathbf{93.74 \pm 0.03}$
64	26min	90.88 ± 1.63	92.30 ± 0.89
128	20min	91.54 ± 1.69	91.52 ± 1.67

Table 6.5: Influence of batch size on training time and classification accuracy.

Pooling layers Pooling is a common technique used in deep learning for dimensionality reduction and feature extraction. In the context of computer vision, the two most popular types of pooling are local max-pooling [48] and global average pooling [25]. Local max-pooling takes the maximum value of a small subregion of a feature map, while global average pooling takes the average of all feature map values. However, for time series data, global average pooling has been shown to be more effective in previous studies [52, 13]. In this work, we investigate whether these findings can be generalized to time series classification tasks. Specifically, we will compare the performance of the TempCNN model using both local max-pooling and global average pooling to determine which pooling method is most effective for our specific application.

Filter size Filter Size is an important Hyperparameter in Convolutional Neural Networks (CNNs) The filter size, also known as the kernel size, determines the receptive field of the convolutional layer and influences the network's ability to capture spatial features in the input. A larger filter size allows the network to capture more complex patterns and details, but at the cost of increased computation and potential overfitting. Conversely, a smaller filter size reduces computation and may improve the network's ability to generalize to new data, but at the risk of losing important features. Therefore, the filter size should be chosen based on the complexity of the task and the size of the input data.

Figure 6.2 displays the OA values as a function of filter size. Each bar represents a different configuration: local max-pooling (MP), local max-pooling and global average pooling (MP + GAP), local average pooling (AP), local and global average pooling (AP + GAP), and global average pooling (GAP). The horizontal

magenta dashed line corresponds to the OA values obtained without pooling layers in the previous experiment.

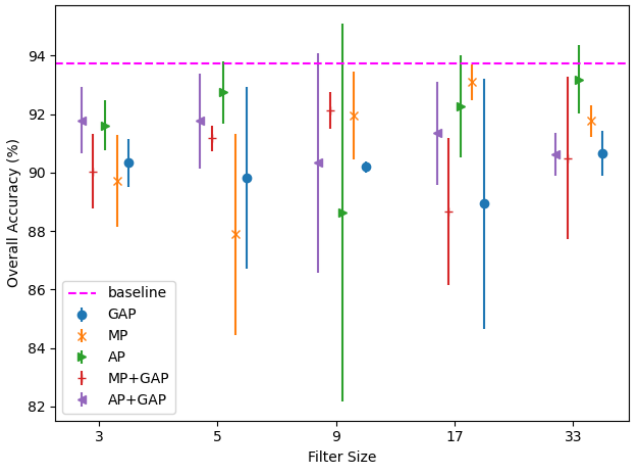


Figure 6.2: Overall Accuracy as a Function of Filter Size for Different Pooling Methods: Local max-pooling (MP) in orange, local max-pooling and global average pooling (MP + GAP) in red, local average pooling (AP) in green, local and global average pooling (AP + GAP) in purple, and global average pooling (GAP) in blue.

Figure 6.2 shows that the use of pooling layers performs poorly: the OA results are almost always below the one obtained without pooling layers (magenta dashed line).

6.3.3 Findings

Our experiments on the TempCNN model for satellite image time series data have provided some valuable insights. First, we found that a network depth of 3 layers is sufficient to achieve high classification accuracy. In addition, we discovered that a width of 64 neurons per layer is sufficient. Furthermore, we found that a batch size of 32 is sufficient for training the model. Finally, we observed that the use of pooling layers did not contribute significantly to the classification accuracy.

The model has been implemented in Python using the Keras library [9] and the TensorFlow backend [1]. The source code for this model is available at `https://github.com/dnldsht/temporalCNN`, which is a forked version of the orginal TempCNN implementation [44].

6.4 AJ-RNN

The AJ-RNN model proposed in the paper was implemented in Python 2.7 and Tensorflow 1.0. However, given the changes in technology and advances in the field since the publication of the paper, we found it necessary to re-implement the model in a more recent version of TensorFlow. As such, we spent a considerable amount of time and effort to re-implement the model in Tensorflow 2.0 in a modularized fashion, with the aim of enhancing code readability and facilitating experimentation with different configurations.

In addition to the re-implementation of the model, we also performed a careful validation process to ensure that the accuracy achieved by our model was consistent with the results reported in the original paper, using the same datasets. This involved testing the model on a range of datasets and comparing the results with the original paper. By doing so, we were able to ensure that our implementation was both accurate and reliable.

Furthermore, we extended the implementation of the model to support multivariate time series data. This undertaking was quite significant as it required extensive additional coding and validation efforts. However, it was essential to ensure that the AJ-RNN model could be effectively utilized in various applications, including our own.

6.4.1 Experimental results

We tried different combinations of hyper-parameters to find the best configuration for our dataset. The hyper-parameters we tuned are the RNN cell type, the units of the cell, the learning rate, the batch size, and the number of epochs for the generator and discriminator.

In our experiments, we investigated the performance of the AJ-RNN model with both LSTM and GRU cells. We tested different numbers of units (64, 128) and dropout rates (0.2, 0.4, 0.6, 0.8) for each cell type. The experimental results are summarized in Table 6.6. A learning rate of 0.001 and a batch size of 256 were used in all experiments, and the generator and discriminator were trained for the same number of epochs. Our findings show that the GRU cell outperforms the LSTM cell in all cases.

Cell type	Units	Dropout	Overall Accuracy
GRU	64	0.2	82.10 ± 1.72
GRU	64	0.4	81.75 ± 1.18
GRU	64	0.6	77.70 ± 2.92
GRU	64	0.8	78.78 ± 2.26
GRU	128	0.2	80.58 ± 1.17
GRU	128	0.4	85.84 ± 1.68
GRU	128	0.6	$\mathbf{86.07 \pm 1.58}$
GRU	128	0.8	85.03 ± 2.66
GRU	256	0.2	80.49 ± 1.47
GRU	256	0.4	75.25 ± 1.08
GRU	256	0.6	75.97 ± 1.23
GRU	256	0.8	77.34 ± 1.74
LSTM	64	0.2	79.41 ± 1.76
LSTM	64	0.4	84.92 ± 3.69
LSTM	64	0.6	81.16 ± 2.99
LSTM	64	0.8	79.56 ± 2.33
LSTM	128	0.2	84.89 ± 1.56
LSTM	128	0.4	85.08 ± 0.35
LSTM	128	0.6	83.24 ± 1.29
LSTM	128	0.8	82.39 ± 1.99
LSTM	256	0.2	74.18 ± 1.44
LSTM	256	0.4	67.15 ± 1.32
LSTM	256	0.6	71.36 ± 1.56
LSTM	256	0.8	$\mathbf{85.37 \pm 2.37}$

Table 6.6: Overall Accuracy for GRU and LSTM.

After conducting experiments with both LSTM and GRU cells with varying numbers of units and dropout rates, we determined that the GRU cell outperformed the LSTM cell. As a result, we chose to further experiment with the GRU cell. We found that using a GRU cell with 128 units was a good compromise between model complexity and accuracy. This decision was made after careful consideration of the trade-off between the number of parameters in the model and its ability to accurately classify incomplete time series data.

We also investigated the effect of batch size on the overall accuracy of the model. We tried different batch sizes and found that the batch size of 256 yielded the best results. The results are summarized in Table 6.7.

Batch size	Learning rate	Dropout	Overall Accuracy
256	1e-03	0.0	80.13 ± 2.20
256	1e-03	0.2	81.58 ± 1.17
256	1e-03	0.4	85.85 ± 1.39
256	1e-03	0.6	$\mathbf{86.03 \pm 1.51}$
256	1e-03	0.8	84.03 ± 2.66
256	1e-04	0.4	80.00 ± 1.91
256	1e-04	0.6	85.46 ± 1.01
256	1e-04	0.8	83.88 ± 1.33
32	1e-03	0.0	35.57 ± 0.00
32	1e-03	0.5	29.32 ± 0.00
32	1e-04	0.0	35.58 ± 4.65
32	1e-04	0.5	36.27 ± 8.39
32	1e-04	0.8	35.58 ± 2.33
32	1e-05	0.0	56.12 ± 0.88
32	1e-05	0.5	69.33 ± 0.34
32	1e-05	0.8	67.12 ± 1.23
32	1e-06	0.0	64.06 ± 2.74
32	1e-06	0.5	66.58 ± 3.78
32	1e-07	0.0	72.97 ± 1.56
32	1e-07	0.5	71.70 ± 1.45
32	1e-08	0.0	73.85 ± 1.31
32	1e-08	0.5	72.92 ± 1.40
32	1e-09	0.0	$\mathbf{77.24 \pm 1.12}$
32	1e-09	0.5	76.37 ± 1.36

Table 6.7: Influence of batch size, learning rate and dropout on Overall Accuracy for the GRU network

During our experiments, we observed that the choice of batch size has a significant impact on the accuracy of the model, when the learning rate is was constant. Specifically, we found that using a smaller batch size can increase the likelihood of over fitting the training data, while using a larger batch size can lead to under fitting.

To improve the overall accuracy of the model with a lower batch size, we attempted to decrease the learning rate. However, while we did observe some improvement, we were not able to achieve the same level of accuracy as when using a higher batch size. These findings highlight the importance of carefully selecting hyperparameters, such as batch size and learning rate, to achieve op-

timal performance in machine learning models.

We also explored the impact of increasing the number of the generator G epochs for each training step.

Batch size	G epochs	Overall Accuracy
256	1	$\mathbf{86.03 \pm 1.51}$
256	2	72.58 ± 1.17
256	5	68.05 ± 1.39
32	1	$\mathbf{77.24 \pm 1.12}$
32	2	57.21 ± 1.46
32	5	27.07 ± 1.05

Table 6.8: Influence of G epochs for the GRU network

Our analysis revealed that augmenting the number of G epochs can result in overfitting the training data, as presented in Table 6.8.

6.4.2 Light AJ-RNN

Due to the computational expense of the AJ-RNN model, we decided to conduct experiments with a lighter version of the model, referred to as the Light AJ-RNN. The goal of this approach was to create a baseline model that could be used as a reference point for comparison with the more complex and computationally demanding AJ-RNN model.

In order to achieve this, we used the Keras library to implement the Light AJ-RNN model. Our approach involved removing the imputation of missing values from the original AJ-RNN model and utilizing the pre-imputed data as input to the model. We removed the Discriminator and thus eliminated adversarial joint training. The Light AJ-RNN model only retained the GRU network and the Classifier from the original AJ-RNN model. This approach was aimed at achieving a reduction in computational costs while still maintaining a level of performance comparable to the original AJ-RNN model.

We conducted experiments with the Light AJ-RNN model using the same hyperparameters as the original AJ-RNN model.

Batch size	Dropout	Learning rate	Overall Accuracy
256	0.0	1e-03	75.14 ± 1.85
256	0.5	1e-03	78.48 ± 1.33
256	0.0	1e-04	87.25 ± 1.20
256	0.5	1e-04	82.60 ± 2.17
256	0.8	1e-04	$\mathbf{87.29 \pm 1.08}$
256	0.0	1e-05	83.29 ± 2.85
256	0.0	1e-06	85.59 ± 1.30
256	0.0	1e-07	81.72 ± 1.32
256	0.5	1e-07	81.76 ± 2.21
32	0.5	1e-03	30.85 ± 1.35
32	0.0	1e-04	33.55 ± 1.42
32	0.5	1e-04	32.33 ± 1.33
32	0.0	1e-05	44.23 ± 2.02
32	0.0	1e-06	54.32 ± 3.73
32	0.0	1e-07	52.98 ± 2.22
32	0.0	1e-08	$\mathbf{55.80 \pm 1.27}$
32	0.5	1e-08	52.32 ± 1.36

Table 6.9: Overall accuracy of Light AJ-RNN model for different hyperparameters

The results of the experiments are shown in Table 6.9.

Despite our initial hopes that the Light AJ-RNN model would prove to be a viable alternative to the computationally expensive original AJ-RNN model, the results of our experiments showed that the overall accuracy of the Light AJ-RNN model was comparable to that of the original AJ-RNN model for most of the hyperparameter combinations tested.

Interestingly, despite using pre-imputed data as input to the model, we did not observe the expected improvement in overall accuracy that we had hoped for. While disappointing, this result was nonetheless informative and suggested that the original AJ-RNN model was indeed necessary for achieving the highest levels of accuracy in our application.

Therefore, we made the decision to halt further experimentation with the Light AJ-RNN model and instead focused our efforts on continuing to refine and improve the original AJ-RNN model.

6.4.3 Findings

In the study conducted on the AJ-RNN model, a deep learning model for classifying time series while imputing missing data. Our experiments aimed to explore the performance of the model and to find the best hyperparameters for our real-world dataset.

Overall, our findings were mixed. While the model showed promise on some of our experiments, we found that the overall accuracy on our real-world dataset was not as high as we expected. It is worth noting that training RNNs is very computationally expensive, and we had to run many experiments to find the best hyperparameters.

We also found that the adversarial joint training is really sensitive to the hyperparameters. We experimented with different learning rates, batch sizes, and dropout rates and found that these parameters can significantly impact the performance of the model. Therefore, it is important to carefully tune these hyperparameters to achieve the best possible results.

In conclusion, we found that the AJ-RNN model can be an effective tool for imputing missing data. However, it is important to carefully consider the hyperparameters and to perform extensive experimentation to achieve the best possible results.

The model has been implemented in Python using the Keras library [9] and the TensorFlow backend [1]. The source code for this model is available at `https://github.com/dnldsht/AJ-RNN`, which is a reimplemented and extended version of the initial AJ-RNN implementation [38].

6.5 L-TAE

In order to embed the input images into the L-TAE model, we needed to replace the Pixel Set Encoder (PSE) used in the original model with an encoder that was more appropriate for our dataset.

This led to the adoption of the Dense Encoder (DE), which is a linear neural network that takes a 16-channel image as input and uses a hyperbolic tangent (tanh) activation function to generate a 64-dimensional vector. Our goal in designing the DE was to ensure computational efficiency, allowing faster training and inference times, while maintaining strong performance on our dataset.

6.5.1 Experimental results

After adapting the L-TAE model to our dataset, we trained it by adjusting the hyperparameters to find the optimal configuration, guided by the results of their research.

In the following, we have described the findings of the authors on the impact of the different hyperparameters on the performance of the model.

Param	Description
E	size of the embeddings (E), if input vectors are of a different size, a linear layer is used to project them d_model-dimensional space
H	Number of attention heads
K	Dimension of the key and query vectors
MLP	Number of neurons in the layers of MLP

Table 6.10: Hyper-parameters of L-TAE model

Number of heads It appears that the performance is only marginally impacted by the number of heads. Their hypothesis is that an increase in the number of heads (H) can be advantageous, but a reduction in group size (E') can be detrimental.

Dimension of keys The experiments indicate that smaller key dimensions, as opposed to the typical values used in NLP or for the TAE ($K = 32$), lead to better performance on the problem. The L-TAE can achieve comparable performance to the TAE with only 2-dimensional keys.

Dimension of Input The expected outcome of having larger input embeddings is an increase in performance, as it corresponds to a more comprehensive representation. Nevertheless, on the dataset being considered, the benefits of increasing the number of parameters are diminishing.

Query-as-Parameter To evaluate the impact of the design choices, a variation of the network is trained using the same master-query scheme as the TAE. The larger linear layer that results increases the model's size to a total of 170k parameters. The observation that the resulting mIoU is only 49.7 suggests that the query-as-parameter scheme is advantageous not only in terms of compactness but also for achieving better performance.

Table 6.11 shows the performance results of the L-TAE architecture with different configurations of the following hyperparameters: number of heads H, dimension of keys K, and number of channels E in the input sequence. All results were obtained using a 5-fold cross-validation scheme. The performance metrics were measured for two different scenarios: one where missing values were imputed and one where missing values were not imputed.

Params	E	H	MLP	No imputation	Pre imputation
43k	128	8	128	93.37 ± 1.65	93.08 ± 1.16
68k	128	16	128-128	93.43 ± 1.37	93.39 ± 1.16
123k	256	16	256-128	93.15 ± 1.72	93.44 ± 1.22
299k	512	32	512-128	$\mathbf{93.65 \pm 1.30}$	93.40 ± 1.16
749k	1024	32	1024-256-128	92.91 ± 1.90	$\mathbf{93.58 \pm 1.29}$

Table 6.11: Results of the L-TAE model with different parameters

Overall, the results reported in Table 6.11 demonstrate the effectiveness of the L-TAE architecture in accurately classifying the input data even with a lower number of parameters. The inclusion of missing data through imputation did not significantly impact the performance of the model, suggesting that the L-TAE architecture is robust to the presence of missing values.

The model has been implemented in Python using the PyTorch library [43]. The source code for this model is available at `https://github.com/dnldsht/lightweight-temporal-attention-pytorch`, which is a forked version of the orginal L-TAE implementation [18].

6.6 Comparing results

The performance of the different models on both the imputed and non-imputed datasets is compared in this section.

Model	Overall Accuracy
RF	91.03 ± 0.42
Light AJ-RNN	87.29 ± 1.08
TempCNN	93.74 ± 0.03
L-TAE	93.58 ± 1.29

Table 6.12: Overall accuracy of the best models with pre imputed missing values

Model	Overall Accuracy
RF	89.72 ± 0.84
AJ-RNN	86.07 ± 1.58
TempCNN	92.57 ± 0.84
L-TAE	93.65 ± 1.30

Table 6.13: Overall accuracy of the best models without imputation of missing values

Overall, our experiments show that TempCNN and L-TAE outperformed the other models on both datasets. Specifically, L-TAE achieved an accuracy of 93.58% and 93.65% on the imputed and non-imputed datasets, respectively, while TempCNN achieved an accuracy of 93.74% and 92.57%. RF and AJ-RNN both achieved lower accuracies. These results suggest that deep learning models such as TempCNN and L-TAE may be more suitable for modeling satellite image time series data.

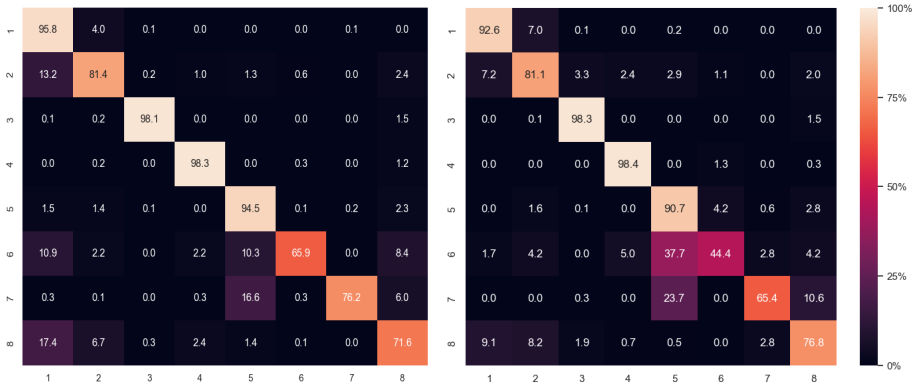


Figure 6.3: Confusion matrices for the TempCNN model (left) and L-TAE model (right) on the test dataset. The y-axis represents the true labels, and the x-axis represents the predicted labels.

Based on the confusion matrices, both TempCNN and L-TAE models seem to have difficulty distinguishing between the orchard and crop classes, as indicated by the high number of misclassifications between these two classes. This could be due to the fact that these two classes may share similar temporal patterns in the time series data, making it more difficult for the models to distinguish between them. It may be worth exploring additional features or data sources to improve the performance of the models in distinguishing between these two classes.

6.6.1 Imputation of missing values

The inclusion of missing data is often a challenge in machine learning tasks, as missing values can have a significant impact on the performance of the model. In the first scenario, we addressed this issue by imputing the missing values prior to training the models. This allowed us to utilize all available data in the training process, potentially leading to improved performance.

In the second scenario, we did not impute the missing values, which meant that the training process was performed using only the available non-missing data. This scenario may be particularly useful in cases where imputation may introduce bias or distort the underlying patterns in the data.

Overall, the results reported in Tables 6.12 and 6.13 show that the inclusion of missing data through imputation did not significantly affect the performance of the models.

Conclusions

Experimenting with all of the models presented in this thesis has been a challenging but rewarding experience. It took a considerable amount of time and effort to understand the underlying concepts of each model and to adapt them to the specific problem at hand. However, this process has greatly enhanced my machine learning and data analysis skills, as well as my ability to critically evaluate and compare different models. Overall, this work has provided me with valuable insights into the application of various deep learning models to time series classification tasks and has broadened my knowledge in the field.

7.1 Future work

There are several opportunities for future research to expand on the findings of this study. Firstly, additional evaluation metrics could be included for a more comprehensive analysis of model performance. For example, F1-Score and Mean IoU are widely used metrics in image classification and segmentation tasks and could provide more insights into the performance of the models.

Another area of research could be graph neural networks (GNNs), which have recently gained popularity in various domains, including computer vision and natural language processing. GNNs are particularly useful for data with graph structures, such as social networks, chemical compounds, and 3D objects. It would be interesting to explore the potential of GNNs for time series classification tasks and how they compare to traditional deep learning models.

Another potential direction for future work is the use of Im-BiLSTM, a novel

architecture proposed by Chen et al. [8] for time series classification tasks. Im-BiLSTM use bidirectional LSTM layers to enhance the learning of long-term dependencies and improve the interpretability of the model. It would be interesting to compare the performance of Im-BiLSTM with the models evaluated in this study.

Finally, transfer learning could be another area of future research for time series classification tasks. Transfer learning has been successful in various computer vision tasks and involves leveraging pre-trained models to improve the performance of a new task with limited training data. It would be interesting to investigate the potential of transfer learning for time series classification and how it can be applied to the models evaluated in this study.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015. Software available from tensorflow.org.

[2] C. Aggarwal, *Neural networks and deep learning: a text with integrated labs*, Springer, 2018.

[3] M. Arjovsky, S. Chintala, and L. Bottou, *Wasserstein gan*, 2017.

[4] E. M. Barnes, T. R. Clarke, S. E. Richards, P. D. Colaizzi, J. Haberland, M. Kostrzewski, ..., and R. J. Lascano, *Coincident detection of crop water stress, nitrogen status and canopy density using ground based multispectral data*, in Proceedings of the Fifth International Conference on Precision Agriculture, vol. 1619, Bloomington, MN, USA, July 2000, ASA, CSSA, and SSSA.

[5] B. Boehmke, *Computer vision & CNNs: MNIST revisited*. `https://rstudio-conf-2020.github.io/dl-keras-tf/04-computer-vision-cnns.html`, 2020.

[6] L. Breiman and A. Cutler, *Random forests*, Chapman and Hall/CRC, 2001.

[7] K. Charles, *Random forests: A simple introduction*, Towards Data Science, (2020).

[8] B. Chen, H. Zheng, L. Wang, O. Hellwich, C. Chen, L. Yang, T. Liu, G. Luo, A. Bao, and X. Chen, *A joint learning im-bilstm model*

for incomplete time-series sentinel-2a data imputation and crop classification, International Journal of Applied Earth Observation and Geoinformation, 108 (2022), p. 102762.

[9] F. Chollet, *Keras*. `https://keras.io`, 2015. Accessed on 1 February 2018.

[10] G. Developers, *Overview of GAN structure*. `https://developers.google.com/machine-learning/gan/gan_structure`.

[11] V. Dumoulin and F. Visin, *A guide to convolution arithmetic for deep learning*, 2016.

[12] Exxact, *A friendly introduction to graph neural networks*, 2020.

[13] H. I. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller, *Deep learning for time series classification: A review*, arXiv preprint arXiv:1809.04356, (2018).

[14] G. finance. `https://www.google.com/finance`.

[15] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky, *Domain-adversarial training of neural networks*, Journal of Machine Learning Research, 17 (2017), pp. 2096–2030.

[16] B.-c. Gao, *NDWI—A normalized difference water index for remote sensing of vegetation liquid water from space*, Remote Sensing of Environment, 58 (1996), pp. 257–266.

[17] Y. Gao, J. Cheng, H. Meng, and Y. Liu, *Measuring spatio-temporal autocorrelation in time series data of collective human mobility*, Geo-spatial Information Science, 22 (2019), pp. 166–173.

[18] V. S. F. Garnot and L. Landrieu, *Lightweight temporal self-attention for classifying satellite image time series*, 2020.

[19] V. S. F. Garnot, L. Landrieu, S. Giordano, and N. Chehata, *Satellite image time series classification with pixel-set encoders and temporal self-attention*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020.

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, Adaptive Computation and Machine Learning, MIT Press, 2016.

[21] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, *Generative adversarial networks*, Advances in neural information processing systems, 27 (2014), pp. 2672–2680.

[22] A. Graves, *Generating sequences with recurrent neural networks*, arXiv preprint arXiv:1308.0850, 2013.

[23] M. Guillame-Bert, S. Bruch, R. Stotz, and J. Pfeifer, *Yggdrasil decision forests: A fast and extensible decision forests library*, 12 2022.

[24] J. Han, J. Pei, and M. Kamber, *Data Mining: Concepts and Techniques*, Elsevier, Amsterdam, The Netherlands, 2011.

[25] K. He, X. Zhang, S. Ren, and J. Sun, *Deep Residual Learning for Image Recognition*, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, Nevada, USA, June 2016, IEEE, pp. 770–778.

[26] T. K. Ho, *Ensemble Methods in Data Mining: Improving Accuracy Through Combining Predictions*, Morgan Kaufmann, 2009.

[27] D. Ienco, R. Interdonato, R. Gaetano, and D. Ho Tong Minh, *Combining sentinel-1 and sentinel-2 satellite image time series for land cover mapping via a multi-source deep learning architecture*, ISPRS Journal of Photogrammetry and Remote Sensing, 158 (2019), pp. 11–22.

[28] influxdata. `https://influxdata.com`.

[29] J. Inglada, A. Vincent, M. Arias, B. Tardy, D. Morin, and I. Rodes, *Operational high resolution land cover map production at the country scale using satellite image time series*, Remote Sensing, 9 (2017).

[30] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, CoRR, abs/1502.03167 (2015).

[31] B. Jayita, *Hands-on implementation of perceptron algorithm in python.*

[32] D. Kingma and J. Ba, *Adam: A method for stochastic optimization*, in Proceedings of the International Conference on Learning Representations (ICLR), Banff, AB, Canada, April 2014.

[33] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel, *Handwritten digit recognition with a back-propagation network*, in Advances in Neural Information Processing Systems, D. Touretzky, ed., vol. 2, Morgan-Kaufmann, 1989.

[34] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. P. Aitken, A. Tejani, J. Totz, Z. Wang, et al., *Photo-realistic single image super-resolution using a generative adversarial network*, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, vol. 2, 2017.

[35] S. C.-X. Li, B. Jiang, and B. Marlin, *Learning from incomplete data with generative adversarial networks*, in International Conference on Learning Representations, 2019.

[36] B. Lukas and V. P. Chris, *Weight and biases: A tool for visualizing and tracking deep learning experiments.*

[37] Y. Luo, X. Cai, Y. Zhang, J. Xu, and Y. Xiaojie, *Multivariate time series imputation with generative adversarial networks*, in Advances in Neural Information Processing Systems 31, 2018, pp. 1601–1612.

[38] Q. Ma, S. Li, and G. Cottrell, *Adversarial joint-learning recurrent neural network for incomplete time series classification*, IEEE Transactions on Pattern Analysis and Machine Intelligence, PP (2020).

[39] S. K. McFEETERS, *The use of the normalized difference water index (ndwi) in the delineation of open water features*, International Journal of Remote Sensing, 17 (1996), pp. 1425–1432.

[40] M. Mirza and S. Osindero, *Conditional generative adversarial nets*, arXiv preprint arXiv:1411.1784, (2014).

[41] ——, *Conditional generative adversarial nets*, 2014.

[42] mltech, *Blog*. `https://mltech.ai/en/w/multivariate_time_series_forecasting_for_bitcoin_pricing`.

[43] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *Pytorch: An imperative style, high-performance deep learning library*, in Advances in Neural Information Processing Systems 32, Curran Associates, Inc., 2019, pp. 8024–8035.

[44] C. Pelletier, G. I. Webb, and F. Petitjean, *Temporal convolutional neural network for the classification of satellite image time series*, 2018.

[45] M. Phi, *Illustrated guide to LSTMs and GRUs: A step by step explanation*. `https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru`, 2018.

[46] D. Podareanu, V. Codreanu, S. Aigner, C. Leeuwen, and V. Weinberg, *Best practice guide - deep learning*, 02 2019.

[47] U. R. Programming, *UC business analytics R programming guide*. `http://uc-r.github.io`.

[48] S. Ren, K. He, R. Girshick, and J. Sun, *Faster r-CNN: Towards Real-Time Object Detection with Region Proposal Networks*, in Advances in Neural Information Processing Systems, Montreal, QC, Canada, Dec. 2015, Curran Associates, pp. 91–99.

[49] F. Rosenblatt, *The perceptron: A probabilistic model for information storage and organization in the brain*, Psychological review, 65 (1958), p. 386.

[50] J. W. Rouse, R. H. Haas, J. A. Schell, and D. W. Deering, *Monitoring vegetation systems in the great plains with erts*, in Third ERTS-1

Symposium, vol. 1 of NASA SP, Washington DC, 1974, NASA, pp. 309–317.

[51] M. Russwurm and M. Körner, *Self-attention for raw optical satellite time series classification*, arXiv preprint arXiv:1910.10536, (2019).

[52] G. Scarpa, M. Gargiulo, A. Mazza, and R. Gaetano, *A cnn-based fusion method for feature extraction from sentinel data*, Remote Sensing, 10 (2018).

[53] scikit learn, *Documentation*. `https://scikit-learn.org/stable/modules/tree.html`.

[54] Sentinel-2, *Sentinel-2 — Wikipedia, the free encyclopedia*.

[55] S. Shah, *Convolutional neural network: An overview*. `https://www.analyticsvidhya.com/blog/2022/01/convolutional-neural-network-an-overview`, 2022.

[56] J. Sklansky, *Ensemble Machine Learning*, Springer, 2013.

[57] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Dropout: A simple way to prevent neural networks from overfitting*, Journal of Machine Learning Research, 15 (2014), pp. 1929–1958.

[58] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, *Attention is all you need*, CoRR, abs/1706.03762 (2017).

[59] Y. Wu and K. He, *Group normalization*, in European Conference on Computer Vision, Springer, 2018.

[60] H. Xu, *Modification of normalised difference water index (ndwi) to enhance open water features in remotely sensed imagery*, International Journal of Remote Sensing, 27 (2006), pp. 3025–3033.

[61] Y. Yang, J. Zhou, J. Ai, B. Yi, A. Hanjalic, H. T. Shen, and Y. Ji, *Video captioning by adversarial lstm*, IEEE Transactions on Image Processing, PP (2018), pp. 1–1.

[62] J. Yoon, J. Jordon, and M. van der Schaar, *Gain: Missing data imputation using generative adversarial nets*, in International Conference on Machine Learning, PMLR, 2018, pp. 5689–5698.

[63] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, *Unpaired image-to-image translation using cycle-consistent adversarial networks*, in Computer Vision (ICCV), 2017 IEEE International Conference on, 2017.