UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Physics

Unsupervised segmentation of submarine recordings

Tor Kjøtrød

FYS-3941 Master’s thesis in applied physics and mathematics - June 2023

Abstract

The thesis focuses on the unsupervised segmentation of submarine recordings collected by the Norwegian Polar Institute (NPI) using hydrophones. These recordings consists of various mammal species, along with other phenomena like vessel engines, seismic activity, and moving sea ice. With sparse labeling of the data, a supervised learning approach is not feasible, necessitating the application of unsupervised learning techniques to uncover underlying patterns and structures.
The thesis begins by providing essential background theory, including the Fourier transform, spectrograms, and an introduction to clustering algorithms. It then covers the forward pass, parameter update and backpropagation of neural networks. Key components of neural networks and machine learning, such as different layers, activation functions, and loss functions, are also explained. Furthermore, well-known deep learning architectures, namely convolutional neural networks, autoencoders, and recurrent neural networks, are introduced. The Temporal Neighborhood coding method, which encodes underlying states of multivariate, non-stationary time-series [1], and the clustering module that integrates a Gaussian mixture model into a loss function for deep autoencoders [2], are introduced.
The proposed data and methods are presented, followed by experimental results evaluating the performance of the two architectures for segmenting both a simulated dataset and the spectrogram of the submarine recordings. The thesis concludes with a discussion of the results and future directions, highlighting the promising outcomes of the segmentation methods while emphasizing the need for additional data information to further enhance model performance and for further evaluation of the methods performance.

Acknowledgements

First i want to thank Heidi Ahonen and the Norwegian Polar Institute, for allowing me to do my thesis with you, and for providing me with the data i have used in this thesis.
I also want to extend my gratitude towards my supervisor during this thesis, Ahcène Boubekki, for guiding and helping me throughout the past year, for being patient with me, but also for always keeping me in a good mood and motivating me when needed, this has been very valuable to me and i cant thank you enough for that. Thank you to all the other master students i have shared an office with this last year, Fredrik, Ole, Adrian, Astrid and Marit. You have made this busy year fun, but also much easier, thank you for showing up every day.
Lastly i want to thank my wonderful family Per Arne, Evelyn, Emilie, Odin and my girlfriend Gina, the support you have showed me the last five years is incredible, i could not have done this without you. For that im forever greatful.

Contents

List of Figures

List of Tables

1 Introduction

The Norwegian Polar Institute (NPI) has collected a large amount of sub marine audio recordings using hydrophones, the data contains recordings of several mammal species, such as bearded whales and beluga whales, there are also recordings of several other phenomenons, such as vessel engines, seismic activity and moving sea ice. The objective of this thesis is segmentation of these submarine recordings. The data we have received is sparsely labeled, which means that applying a supervised learning approach is not feasible. In such a scenario, unsupervised learning techniques become particularly valuable. These methods can help us uncover underlying patterns and structures within the data [3].

The thesis starts with some background theory to provide the necessary context.This includes covering two fundamental signal processing topics, the Fourier transform and spectrograms in Chapter 2. Additionally, we provide a brief introduction to clustering, a central technique in unsupervised learning [4]. Along with some well known clustering algorithms in Chapter 3.
How neural networks process data through the forward pass, and how they can update their parameters through gradient descent and the backpropagation algorithm [5] is covered in Chapter 4. In Chapter 5, we cover essential components of neural networks and machine learning, including different types of layers within a neural network, activation functions that introduce non-linearity to the networks [6], and the loss function.
Moving forward, we introduce some of the well known deep learning architectures, the convolutional neural network, autoencoders and recurrent neural networks in Chapter 6 Subsequently, two deep learning methods that are ap-

plicable in our case, the Temporal Neighborhood coding method introduced by Sana Tonekaboni, Danny Eytan and Anna Goldengerg, a framework that encodes underlying states of multivariate, non-stationary time-series [1]. Together with the clustering module introduced by Ahcène Boubekki, Michael Kampffmeyer, Ulf Brefeld and Robert Jensen, Which, by rewriting a Gaussian mixture model into a loss function, allows a deep autoencoder to learn valueable information from the clustering of its encoding [2]. Is introduced in Chapter 7

The Proposed data and method is introduced in Chapter 8 followed by the experiments and results we achieved in Chapter 9. In this chapter we show the performance of the two architectures in regards of segmenting a simulated dataset aswell as the segmentation of the submarine recordings. In the Final Chapter, we discuss our results and future direction, where we conclude with that the methods applied for segmentation shows promising results, but further steps can be made to improve the performance of the models.

Part I

Background

2 Signal Processing

Signal processing is a discipline for manipulating and understanding information carrying signals [7]. In this chapter we will cover two techniques within signal processing, namely the Fourier transform and the spectrogram. The Fourier transform is a mathematical tool that decomposes a signal into its frequency components allowing for analysis of the signal in the frequency domain [7]. The spectrogram is a visualization tool that combines the time and frequency domain at once.

All Sections from this chapter is from the project paper [8].

2.1 Discrete Fourier Transform

The Fourier transform can be applied to a time dependant signal to transform the signals domain from time to frequency. By applying the Fourier transform to a time signal, it is possible to analyse the frequency components present in the signal and their magnitude. On the other side, it is not possible to analyse how the frequency spectrum changes with time, by only applying a Fourier transformation to a time signal [7]. The discrete Fourier transforms input, will be a discrete time signal $x[n]$, with real or complex values, consisting of N samples. The signal is then transformed, often into a complex valued, signal in frequency domain $X[k]$ with N samples [7].
Definition. The discrete Fourier transform is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-i(2\pi/N)kn}. \tag{2.1}$$

Where $X[k]$ is the kth frequency component of the discrete Fourier transform output, k is the frequency index ranging from 0 to $N-1$,$x[n]$ is the input discrete time signal with length N, where n is the time index, also ranging from 0 to $N-1$ [7].

From equation 2.1, one can see that to evaluate one $X[k]$ term, for any $k \neq 0$, $(N-1)$ multiplications and summations are needed. To evaluate all terms, the total number of multiplications becomes $(N-1)^2$ aswell as $(N^2 - N)$ summations, which can be costly, computationally, for larger signals [7]. The Fast Fourier transform (FFT) overcomes the limitation of costly computations by exploiting the periodic properties of the discrete Fourier transform [7].

2.2 Spectrogram

To analyse where in time each frequency occurs, and the power of the frequencies, the approach is slightly different. While still applying the Fourier transform to the signal, the signal is first divided into small batches of equal size, that often overlap eachother. Then a windowing function is applied to every batch followed by the Fourier transformation of each batch. This process is called the short time Fourier transform. The spectrogram is the plot of all the Fourier transformed batches of the entire signal [9].
Definition. The Short time Fourier Transform is defined as:

$$X[k, n_s] = \sum_{m=0}^{L-1} w[m]x[n_s + m]e^{-j(2\pi k/N)m}. \tag{2.2}$$

Where $X[k, n_s]$ is the Short time Fourier transformed signal, n_s is called the analysis time index, which specifies where the short time discrete Fourier transform is taken,$w[m]$ is a windowing function, that is non-zero only on the interval $m = 0$, 1,..., $L - 1$, where L is much smaller than the length of $x[n]$ and $x[n]$ is the input time series [7].
It is common to scale the output of the spectrogram. One way of scaling is to calculate the dB amplitude[10], which is calculated as: $20 \log_{10}(X[k, n_s])$.

3 Clustering

Clustering is a technique that reveals hidden structures within data by grouping similar data points into clusters [11]. We will begin this chapter by introducing the concept of partitional clustering, and introduce some of the well known partitional clustering algorithms, the k-means algorithm and Gaussian mixture models (GMM). K-means partition the dataset into distinct clusters [12]. GMM, on the other hand, employs probabilistic modeling with Gaussian distributions to capture data patterns [13].

3.1 Partitional Clustering

Partitional clustering seeks to assign each datapoint to one and only one cluster [13].
Given a dataset $\mathbf{X} = [\mathbf{x}_1, .., \mathbf{x}_j, ..., \mathbf{x}_N]$, where $\mathbf{x}_j = [x_{j1}, .., x_{jd}]$ that is to be partitioned into K clusters $C = [C_1, .., C_K]$, where $K \leq N$, partitional clustering seeks to partition $\mathbf{X}$ such that:

$$C_i \neq \emptyset, i = 1, .., K, \tag{3.1}$$

$$\bigcup_{i=1}^{K} C_i = \mathrm{X}, \tag{3.2}$$

$$C_i \cap C_j = \emptyset, i, j = 1, ..., K, i \neq j. \tag{3.3}$$

Equation 3.1 tells us that no cluster C_i for $i = 1, .., K$ is to be an empty set. Equation 3.2 Tells us that the union of all clusters should be the dataset, that means that all the datapoints in $\mathbf{X}$ should be assigned to a cluster. Finally equation 3.3 tells us that the intersection between any two clusters should be the empty set, meaning that a datapoint $\mathbf{x}_i$ for $i = 1, .., N$ should be assigned to a single cluster. [13].

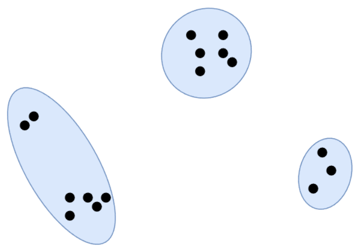


Figure 3.1: Illustration of a clustering, where every black point represents a single datapoint. The blue sections represents the clusters.

Figure 3.1 illustrates partitional. Where we can see that the datapoints are grouped into three clusters. One can also notice that none of the points are assigned to more than one cluster (equation 3.3), no clusters are empty (equation 3.1) and no points are unassigned to any cluster (equation 3.2).

Equations 3.1 - 3.3 does not set any constraints to which points are to be clustered together, clustering with these constraints alone, may therefore yield trivial solutions. Proximity measures can be used to cluster datapoints together based on their distance to one another. A common proximity measure is the

l_2-norm [13]. Objective functions, also known as clustering criteria, are used in clustering algorithms, where they evaluate the quality of a clustering solution. Clustering algorithms aim to find the optimal partition of the datapoints, through optimizing the objective function. A common objective function is the sum of squared errors:
Definition. The sum of squared errors is defined as:

$$\begin{aligned} \mathrm{J}_s(\Gamma, \mathrm{M}) &= \sum_i \sum_j \gamma_{ij} ||\mathrm{x}_j - \mathrm{m}_i||^2 && (3.4) \\ &= \sum_i \sum_j \gamma_{ij} (\mathrm{x}_j - \mathrm{m}_i)^T (\mathrm{x}_j - \mathrm{m}_i). && (3.5) \end{aligned}$$

Where $\Gamma = \{\gamma_{ij}\}$ is is a partition matrix, where γ_{ij} takes on binary values and is equal to 1 if the point x_j is assigned to the ith cluster, and 0 else, M is the cluster centroid, and m_i is the sample mean of the ith cluster [13].

3.2 K-Means Algorithm

The K-means algorithm is one of the most commonly used clustering algorithms. The K-means algorithm is optimized by the minimization of the sum of squared errors function, showed in equation 3.5. The algorithms works by first initializing the partition matrix randomly, and calculating the cluster centroids, M in equation 3.5. Then all points in the dataset, are assigned to the cluster which they have the closes euclidean distance to. The cluster centroids are then recalculated. The datapoints are then reassigned to the now shifted cluster centroids they have the closest euclidean distance to. This assigning of datapoints to clusters, and recalculation of the clusters centroids are repeated iteratively, until there is no change in the clusters [13].

Given a dataset $\mathbf{X}$ consisting of N data points: $\mathbf{X} = [\mathbf{x}_1, .., \mathbf{x}_j, ..., \mathbf{x}_N]$, where $\mathbf{x}_j = [x_{j1}, .., x_{jd}]$ that is to be partitioned into K clusters $C = [C_1, .., C_K]$, for $K \leq N$. The K-means algorithm can be expressed as:

1. Initialize:
   - Randomly initialize K cluster centroids $\mu_1, \dots, \mu_K$.
2. Assign data points to clusters:
   - For each data point x_j:
     - Compute the l_2-distance $d(\mathrm{x}_j, \mathrm{c})$ between x_j and the cluster centroids .
     - Assign x_j to the cluster with the closest centroid: $C^* = \underset{k}{\mathrm{argmin}}(d(\mathrm{x}_j, \mathrm{c}_k))$.
3. Update cluster centroids:
   - For each cluster $C = [C_1, \dots, C_K]$:
     - Calculate the new centroids:

$$\mu_k = \frac{1}{N_k} \sum_{\mathrm{x}_i \in C_k} \mathrm{x}_i,$$

where C_k is the set of data points assigned to cluster k, and N_k is the number of data points assigned to C_k.

4. Repeat steps 2 and 3 until convergence [13].

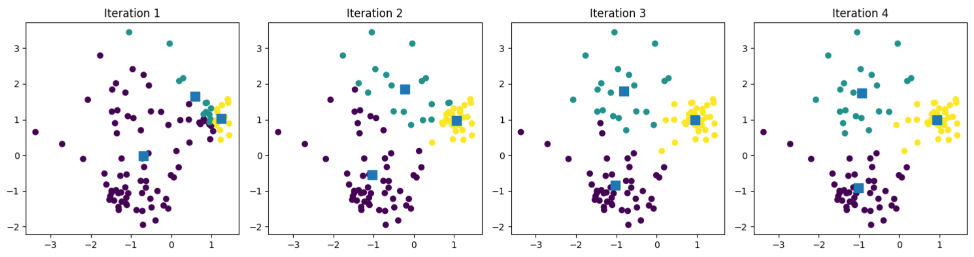


Figure 3.2: Illustration of K-means iteratively process of recalculating the centroids and reassigning the points.

Figure 3.2 illustrates how the k-means algorithm starts with a random initialization of the centroids, and how the datapoints are assigned to the nearest cluster center, after every iteration, the cluster centres are recalculated, and the points are reassigned to the new nearest clusters.

3.3 Gaussian Mixture Model

A Gaussian mixture model (GMM) is a clustering method that predicts the probability density function of the data, assuming that it follows a multivariate normal distribution [14]:

$$p(\mathrm{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}\left(\mathrm{x}|\boldsymbol{\mu}_k, \Sigma_k\right). \tag{3.6}$$

Where $p(\mathrm{x})$ is the probability density function of x, π_k is the prior probability of a datapoint belonging to mixture component k and $\mathcal{N}(\cdot)$ is the multivariate Gaussian distribution with mean vector $\boldsymbol{\mu} \in R^d$ and covariance matrix $\Sigma \in R^{d\times d}$ [14]. The clustering via Gaussian mixture models is done by maximizing the log likelihood function with respect to the variables π_k $\boldsymbol{\mu}_k$ and Σ_k [14].

The maximization of the log likelihood function can be done by the Expectation maximization (EM) algorithm [12]. The log likelihood function for a Gaussian mixture model, that is to be fitted to a dataset (X) of N observations, can be expressed as:

$$\ln\left(p(\mathrm{X}|\pi, \boldsymbol{\mu}, \Sigma)\right) = \sum_{n=1}^{N} \ln\left(\sum_{k=1}^{K} \pi_k \mathcal{N}\left(\mathrm{x}|\boldsymbol{\mu}_k, \Sigma_k\right)\right). \tag{3.7}$$

The EM algorithm starts by initializing the parameters π, $\boldsymbol{\mu}$ and Σ, and evaluating the log likelihood function. Then the cluster responsibilities are evaluated using the current parameter values, the cluster responsibilities are calculated using the equation [12]:

$$\textbf{E}\text{-step:} \quad \gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(\mathrm{x}_n|\boldsymbol{\mu}_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathrm{x}_n|\boldsymbol{\mu}_j, \Sigma_j)}. \tag{3.8}$$

The evaluation of the cluster responsibilities are considered the **E**-step of the EM algorithm. The **M**-step of the EM algorithm is the re-estimation of the parameters. following the equations [12]:

$$\boldsymbol{\mu}_k^{\text{new}} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \mathrm{x}_n, \tag{3.9}$$

$$\textbf{M}\text{-step:} \quad \Sigma_k^{\text{new}} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \left(\mathrm{x}_\mathrm{n} - \boldsymbol{\mu}_k^{\text{new}}\right) \left(\mathrm{x}_\mathrm{n} - \boldsymbol{\mu}_k^{\text{new}}\right)^{\mathrm{T}}, \tag{3.10}$$

$$\pi_k^{\text{new}} = \frac{N_k}{N}. \tag{3.11}$$

Where N_k can be considered as the total responsibility assigned to cluster k, expressed as [12]:

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk}). \tag{3.12}$$

after every **E** and **M** step of the EM algorithm, the log likelihood is evaluated, to check for convergence [12].

When the Gaussian mixture model is fitted to the data, each datapoint is assigned to the mixture component based on the posterior probability of the datapoint belonging to the mixture component. As the assignment to each cluster is based on probabilities, the GMM method performs soft clustering, meaning that one single datapoint can be assigned to multiple clusters, or mixture components, but with different probabilities [12].

4 Neural Network

In this section we will discuss feed-forward networks and forward propagation, providing insights into how information flows through the network. Optimization algorithms, such as stochastic gradient descent and the Adam optimizer, will be introduced as they optimize the network's performance [15]. Lastly, we will go through the backpropagation algorithm.

4.1 Feed Forward Networks and Forward Propagation

This section is based on parts from the project paper [8].
A traditional feed forward neural network consists of at least one layer, the output layer. The input layer receives the input data and applies weights and usually biases to the input, then the processed input is passed to the output layer where another set of weights and biases is applied to create an output. In addition to the mandatory output layer there can be hidden layers, the hidden layers also consist of neurons with weights and biases, and are between the input and output layer [16].

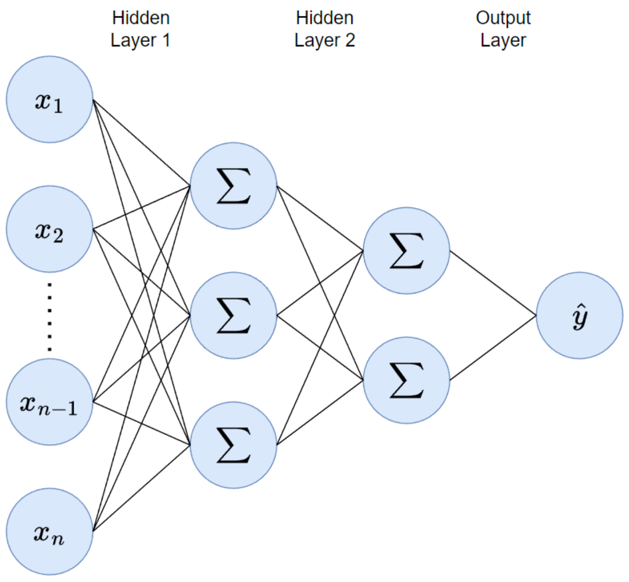


Figure 4.1: Illustration of Feed forward dense network architecture.

Figure 4.1 illustrates the architecture of a generic feed forward Network. Every feed forward network has at least one output layer, where all layers between the input and output layer are called hidden layers. As seen in the illustration, every input or neuron in one of the hidden layers connects to all neurons in the following layer. There are weights, and usually biases, associated with every neuron. The weights and biases is applied to all inputs the neuron receives to create a weighted sum. The weighted sum is passed through an activation function before its passed to the next layer [16]. The values for the weights and biases is learned during the training of the network [17]. When the final layer, the output layer, is reached. A final weighted sum and activation is performed and the model makes a prediction [18].

The activation of a neuron in a neural network is the activation function applied to the weighted sum of the neurons input. The weighted sum of the neurons input follows the equation:

$$v = \mathrm{w}x + b. \tag{4.1}$$

Where $\mathbf{w}$ is the weight matrix associated to the neuron, b is the bias vector associated to the neuron, and x is the input data vector to the neuron. The activation of the neuron follows the formula:

$$a = g(v). \tag{4.2}$$

Where a is the activation of the neuron, $g(\cdot)$ is the activation function and v is the weighted sum from equation 4.1 [19].

The forward propagation is the passing of the input through all the layers of the model, where every neuron calculates the weighed sum following equation 4.2 [20]. After the forward propagation is done, the loss is calculated. In order to optimize the model such that the loss is minimized, the model has to update its parameters.

4.2 Gradient Descent

A small loss value indicates that the model is fitting to the data [21]. Therefore the need to minimize the loss curve with respect to the weights of the model occurs. In order to do this, the gradient of the loss with respect to the weights of the model is needed. The gradient of the loss tells us the rate of change of the loss with respect to each weight. By computing the gradient of the loss, we can adjust the weights in the direction that reduces the loss the most [20]. The most common optimizing scheme is to update the model parameters using gradient descent. Where the gradient of the loss with respect to a parameter is calculated and the parameter is updated in the opposite direction of the gradient. Mathematically the gradient descent parameter optimization can be expressed as:

$$\mathrm{w}_j^{(r)}(\text{new}) = \mathrm{w}_j^{(r)}(\text{old}) + \nabla \mathrm{w}_j^{(r)}. \tag{4.3}$$

Where $\mathbf{w}_j^{(r)}$ is the weight vector for the jth neuron in the rth layer. $\mathbf{w}_j^{(r)}(\text{new})$ indicates the updated weight $\mathbf{w}_j^{(r)}(\text{old})$ indicates the old weight and $\nabla\mathbf{w}_j^{(r)}$ is the gradient of the loss with respect to the weight $\mathbf{w}_j^{(r)}$ [18]. As $\nabla\mathbf{w}_j^{(r)}$ is the gradient of the loss with respect to the loss, aswell as we know we want to adjust the weights in the negative direction of the gradient, we can express

$\nabla \mathbf{w}_j^{(r)}$ as:

$$\Delta \mathbf{w}_j^{(r)} = -\eta \frac{\partial \mathcal{L}}{\partial \mathrm{w}_j^{(r)}}. \tag{4.4}$$

Where η is the learning rate of the model, the learning rate is a parameter that evaluates the magnitude of the step taken in the negative gradient direction [20]. $\frac{\partial \mathcal{L}}{\partial \mathrm{w}_j^{(r)}}$ is the derivative of the loss function $\mathcal{L}$ with respect to the weight vector of the jth neuron in the rth layer[18].

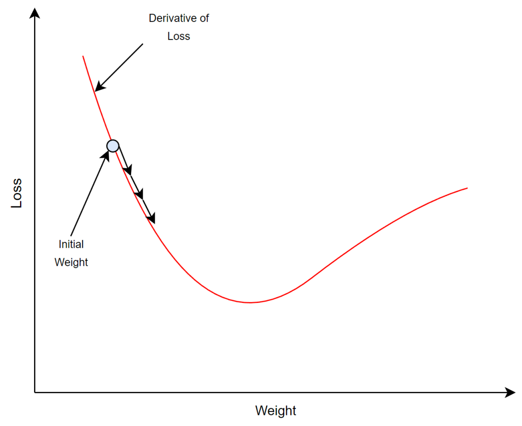


Figure 4.2: Illustration of gradient descent.

Figure 4.2 Illustrates the gradient descent scheme, where the red line is the derivative of the loss function with respect to the weight, the blue point on the line is the initial weight corresponding to $\mathbf{w}_j^{(r)}$(old) in equation 4.3, the arrows pointing from the initial weight is the updated weights position or $\mathbf{w}_j^{(r)}$(new) in equation 4.3. The length of the arrow corresponds to the learning rate parameter, η in equation 4.4. The Figure illustrates how moving in the opposite direction of the gradient, loss is minimized.

The simplest parameter update scheme is by gradient descent[20]. There are other more complex optimizing schemes that further builds on stochastic gradient descent schemes, where stochastic gradient descent is the same as gradient descent, but the gradient is estimated from a smaller portion, or batch of the data, instead of calculating the exact gradient [14]. One of these more complex variants is the Adam optimizer. The Adam optimizer updates two exponential moving averages estimates of the gradient and the squared gradient, where the first moving average is of the mean of the gradient and the second moving average is of the uncentered variance of the gradient [22]. The moving averages are:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \tag{4.5}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2. \tag{4.6}$$

Where m_t is the moving average estimate of the mean of the gradient, v_t is the moving average estimate of the uncentered variance, both at time step t. $\beta_1, \beta_2 \in \{0, 1)$ are hyperparameters that controls the exponential decay rate of the moving averages and g_t is the gradient at time step t [22]. As the estimates of the mean and variance moment are biased, the Adam optimizer also have a bias correction for the two momentum estimates:

$$\hat{m_t} = \frac{m_t}{(1 - \beta_1^t)}, \tag{4.7}$$

$$\hat{v_t} = \frac{v_t}{(1 - \beta_2^t)}. \tag{4.8}$$

Where $\hat{m_t}$ and $\hat{v_t}$ are the bias corrected momentum estimates of the mean and variance from equations 4.5 and 4.6 respectively [22].

The weight updating scheme of the Adam optimizer follows the equation:

$$w_t = w_{t-1} - \eta \frac{\hat{m_t}}{\sqrt{\hat{v_t}} + \epsilon}. \tag{4.9}$$

Where w_t is the new weight, w_{t_1} is the old weight, η is the learning rate, $\hat{m_t}$ is the bias corrected mean momentum estimate for the weight, $\hat{v_t}$ is the bias corrected variance momentum estimate for the weight and ϵ is a small value to avoid division by zero [22]. From equation 4.9, one can see that the learning rate of the optimizer is scaled by the momentum estimates of the gradient, thus yielding an adaptive learning rate for each weight, in contrary to one constant learning rate for all weights, like in the basic gradient descent scheme from equation 4.3 [14].

4.3 Backpropagation

To update the weights, an optimizing scheme is applied like described in Section 4.2. But to calculate the gradient itself, the backpropagation algorithm is applied. The backpropagation algorithm is propagating through the model, similarly to the forward propagation described in Section 4.1, However the backpropagation algorithm starts at the output of the model [19]. In the output layer, the derivative of the loss function with respect to $v_j^{(L)}(i)$ are explicit [18], where $v_j^{(L)}(i)$ is the weighted sum performed by the jth neuron in the output layer L. The derivative of the gradient is thus:

$$\delta_j^{(L)}(i) = \frac{\partial \mathcal{L}(i)}{\partial v_j^{(L)}(i)}. \tag{4.10}$$

Where $\mathcal{L}(i)$ is the loss calculated given model output $g(v_j^{(L)}(i))$, where $g(\cdot)$ is the activation function in the output layer, and label $y(i)$, where $i = 1, 2, ..., N$, $\mathbf{v}_j^{(L)}(i)$ is the weight of the jth neuron in the output layer (L) and $\partial_j^{(L)}(i)$ is the loss gradient with respect to the $v_j^{(L)}(i)$ [18].

For all the other layers, or hidden layer $r < L$ the gradient is calculated differently. This is because of the successive dependence among the layers [18].

$$\frac{\partial \mathcal{L}(i)}{\partial v_j^{(r-1)}} = \sum_{k=1}^{k_r} \frac{\partial \mathcal{L}(i)}{\partial v_k^{(r)}(i)} \frac{\partial v_k^{(r)}(i)}{\partial v_j^{(r-1)}(i)}, \tag{4.11}$$

where $\frac{\partial \mathcal{L}(i)}{\partial v_j^{(r-1)}(i)}$ is the derivative of the loss function with respect to the output of the j-th neuron in the $(r-1)$th layer, $v_j^{(r-1)}(i)$ and k_r is the number of neurons in the rth layer. If we apply the notation from equation 4.10 in equation 4.11, we get:

$$\delta_j^{(r-1)}(i) = \sum_{k=1}^{k_r} \delta_j^{(r)}(i) \frac{\partial v_k^{(r)}(i)}{\partial v_j^{(r-1)}(i)}. \tag{4.12}$$

The second term in the sum is the derivative of the output of the kth neuron in the rth layer with respect to the output of the jth neuron in the $(r-1)$th layer, $v_j^{(r-1)}(i)$. This term is given by the derivative of the activation function $g(\cdot)$ applied to the weighted sum $v_j^{(r-1)}(i)$ [18]:

$$\frac{\partial v_k^{(r)}(i)}{\partial v_j^{(r-1)}(i)} = \frac{\partial}{\partial v_j^{(r)}(i)} g(v_j^{(r-1)}(i)) = g'(v_j^{(r-1)}(i)) w_{kj}^{(r)}. \tag{4.13}$$

If we insert the equation 4.13 into equation 4.12, we get:

$$\delta_j^{(r-1)}(i) = \sum_{k=1}^{k_r} \delta_j^{r}(i) g'(v_j^{(r-1)}(i)) w_{kj}^{(r)}. \tag{4.14}$$

Where $w_{kj}^{(r)}$ is the weight connecting the kth neuron in the $(r-1)$th layer to the jth neuron in the rth layer, and $g'(\cdot)$ is the derivative of the activation function $g(\cdot)$ [18].

Finally, to get the derivative of the loss with respect to the weights of the model, the chain rule is applied once again:

$$\frac{\partial \mathcal{L}(i)}{\partial \mathrm{w}_j^{(r)}} = \frac{\partial \mathcal{L}(i)}{\partial v_j^{(r)}(i)} \frac{\partial v_j^{(r)}(i)}{\partial \mathrm{w}_j^{(r)}}, \tag{4.15}$$

where $\frac{\partial \mathcal{L}(i)}{\partial \mathrm{w}_j^{(r)}}$ is the derivative of the loss with respect to the weight vector of the jth neuron in the rth layer, $\frac{\partial \mathcal{L}(i)}{\partial v_j^{(r)}(i)}$ is the derivative of the loss with respect to the argument of the activation function for the jth neuron in the kth layer and $\frac{\partial v_j^{(r)}(i)}{\partial \mathrm{w}_j^{(r)}} = \mathbf{y}^{(r-1)}(i)$ [18] is the derivative of the argument of the activation function for the jth neuron in the kth layer with respect to the weights in the jth neuron in the rth layer. If we apply the notation from equation 4.10 we get:

$$\frac{\partial \mathcal{L}(i)}{\partial \mathrm{w}_j^{(r)}} = \delta_j^{(r)}(i)\mathrm{y}^{(r-1)}(i), \tag{4.16}$$

Inserted to equation 4.4 we get[18]:

$$\Delta \mathbf{w}_j^{(r)} = -\eta \sum_{i=1}^{N} \delta_j^{(r)}(i)\mathrm{y}^{(r-1)}(i). \tag{4.17}$$

5 Neural Networks

This chapter introduces some of the key concepts in neural networks. We start this chapter with fully connected layers, followed by convolutions, which are central to convolutional neural networks (CNNs). These operations act as filters, extracting features from the input data [23]. Next, we look at other architectural components like down and -upsampling layers followed by dropout and batch normalization, and their purpose. Lastly we show some popular activation functions that introduce non-linearity into neural networks [24] and the role of loss functions in quantifying the error of the model.

5.1 Fully Connected Layers

Fully connected layers are feed forward networks like shown in figure 4.1, with only an input and output layer. Where all neurons in the input layer connects to every neuron in the output layer, that performs a weighted sum as in equation 4.1 followed by an activation function.

5.2 Convolution

This section is based on parts from the project paper [8].
In convolutional neural networks, feature maps are created by performing a sliding dot product between the input and learnable filters, this sliding dot product is the convolution operation [23].

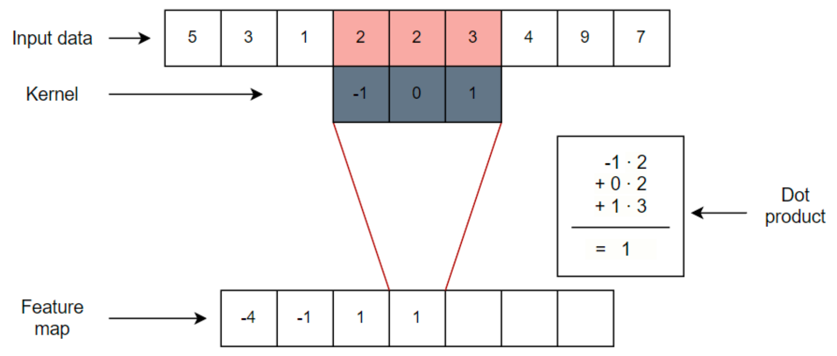


Figure 5.1: Illustration of one dimensional convolution.

Figure 5.1 is a visual illustration on how a convolution between a one dimensional input and a filter of equal dimensionality generates a feature map. **Definition.** The convolution operation is defined as:

$$y[n] = \sum_{k=0}^{M} h[k]x[n-k]. \tag{5.1}$$

where $y[n]$ is the output of the convolution operation at the position n in the feature map, x is the input of length N and h is the kernel with length $M+1$. A common way to express the convolution between an input sequence and a kernel is: $y[n] = h[n] * x[n]$. Where $*$ denotes the convolution operation [7].

A convolution can also be applied as a form of down sampling the input, as the size of the resulting feature map may differ from the input. The parameters that decide the size of the feature map is input size, padding, kernel size and stride. Padding is the adding of extra pixels to the edges of the input data [25], kernel size is the size of the filter applied to the input data [26] and stride is how many pixel values the filter moves between each convolution operation [27]. The size of the resulting feature map can be calculated as follows:

$$S_o = \frac{S_i + 2p - k}{s} + 1. \tag{5.2}$$

Where, S_o is the output size, S_i is the input size, p is the padding, k is the kernel size and s is the stride[28].

In the illustration visualized in figure 5.1 the input size is 9, padding is 0, kernel size is 3 and stride is 1. These values inserted in equation 5.2 results in a output size of 7. In other words, the convolutional operation uses multiple input values to generate one output value, as a result, the size of the output is smaller than the input value.

Convolutional layers are layers where the running dot product between the input data and a set of kernels is performed. The convolutional layers are often used in the beginning of the network, with the intention that the network should learn to extract relevant features in the form of feature maps. The kernels in the model can be random initialized, and the model will update the weights of the kernels during the training of the model. Convolutional layers are especially good for images, or other forms of data were there are spatial patterns present relevant for the task at hand, as the convolutional layers can capture these spatial patterns [29]. As the kernel slides over the input it is processing, the kernel is able to detect small, but still meaningful features from the input, such as edges [30]. Also due to the sliding dot product, the same weights are applied to almost all the datpoints of the input, as a result, the kernel only need to learn one set of parameters, instead of a separate set for each location in the input [30]. This parameter sharing leads to something called translation equivariance, which means that if something is changed in the input, the resulting feature map will also change in the same way [30].

5.3 Transposed Convolution

The transposed convolution is an operation that takes in a low dimension input and returns a higher dimensional output, thus up sampling the input. Transposed convolutions can thus be applied as a response to the down sampling of a convolutional operation [31].

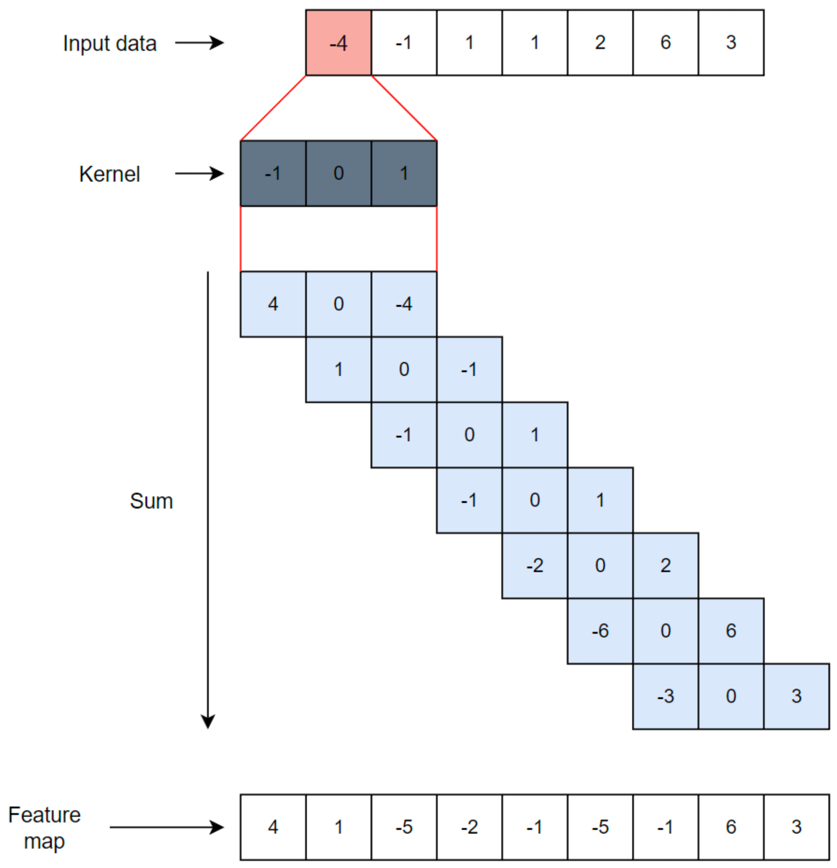


Figure 5.2: Illustration of 1D transposed convolution.

Figure 5.2 illustrates the how the transposed convolution between a one dimensional input and a kernel will result in a larger, one dimensional output. One can see from the illustration, that one value in the input space corresponds to multiple values in the output space, which is opposite to the convolution operation, where multiple values in the input space corresponds to one value in the output space

5.4 Downsampling Layers

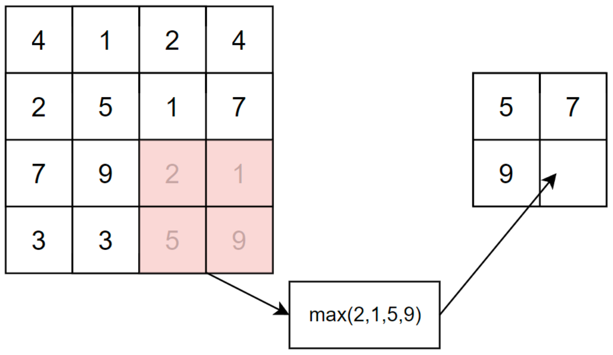


Figure 5.3: Illustration of the Maxpool operation.

Downsampling layers are performed to reduce the dimensionality of the the intermediate representations [32], As described in Section 5.2, Convolutional layers can be applied to reduce the dimensionality, but there are also pooling operations that can be applied as a downsampling layer. The pooling operation is also a form of convolutional operation. The Maxpooling operation is illustrated in figure 5.3, which, similar to the convolution, slides over the input, but instead of calculating a dot product, the maxpool operation simply returns the maximum value. There are other operations similar to the maxpooling operation, such as the average pool, which instead of returning the maximum value, it returns the average value [30]. In addition to reducing the dimensionality of the feature maps, the pooling operation also contributes to making the feature maps invariant to small translations of the input, this means that if the input is altered slightly, the resulting feature map would not change [30].

5.5 Upsampling Layers

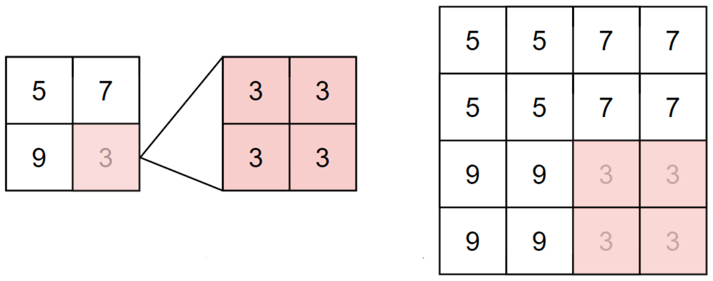


Figure 5.4: Illustration of Upsampling operation.

Upsampling layers, are the opposite of downsampling layers, where the goal is to increase the dimensionality of the representation or feature map. From Section 5.3, we know that transposed convolutions can be applied for upsampling, but there are also other operations that are used. One of the simplest approaches to upsampling is by nearest-neighbor interpolation [33] and is illustrated in figure 5.4. From the figure one can see that one value in the data that is to be upsampled, is repeated in order to double the dimensionality.

5.6 Dropout

Overfitting is a phenomenon in machine learning that occurs when for example a very large network is trained on a relatively small training set [34]. One consequence of an overfitted model is that the model can achieve very high accuracy on the training data, but performs poorly in the testing phase, where a new dataset with a similar distribution is presented to the model [34]. Overfitting occurs when the model starts to not only learn patterns in the data, but also the noise present in the data [35]. This leads to so called co-adaptation of feature detectors, where the activation of a neuron, for example, is only relevant if the activation of other specific neurons occur [34]. To avoid overfitting, dropout layers are introduced to the model, where a percentage of the feature detectors, such as neurons in a dense layer, are omitted from the network [34]. By omitting a percentage of the neurons the remaining neurons are forced to be more robust and not rely on the activity of other specific neurons [36].

5.7 Batch Normalization

When training a Neural network, we can feed one datapoint (x) at a time through the network, or as mentioned in Section 4.2, it is also common to feed the Neural network batches of data. As shown in Section 4.2 The parameters of the model are updated during training by backpropagating the gradients. One problem that arises by doing so, is called internal covariate shift, which means that the distribution of the input of every layer in the model changes during training, thus slowing down the training of the model, one approach that addresses this problem is Batch normalization [37].
Definition. Batch normalization is defined as:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}. \tag{5.3}$$

Where x is a d-dimensional input $x = (x^{(1)}, ..., x^{(d)})$ and $\mathbb{E}[\cdot]$ is the expected value and $\mathrm{Var}[\cdot]$ is the Variance [37].

Batch normalization is a technique that helps to standardize the input of the layers in the model. as a result of adding batchnormalizing after each layer, the optimization curve smooths out, and the gradients are more stable[38].

5.8 Activation Functions

In Deep learning, activation functions or transfer functions are used to introduce non-linearity to the model, as we discussed in Section 4.1, the neurons in a neural network performs a weighted sum to its inputs, thus performing a linear mapping of its input [24]. This weighted sum is passed through the chosen activation function to introduce non-linearity to the neurons output [6]. There are multiple different activation functions to choose from, and the choice of activation function may impact the performance of the model, as the activation function has an impact on how the input of the model is mapped [6]. A common activation functions is the Sigmoid function [39].
Definition. The Sigmoid function and its derivative is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-ax}}, \tag{5.4}$$

$$\sigma'(x) = \frac{1}{1 + e^{-ax}}\left(1 - \frac{1}{1 + e^{-ax}}\right) = \frac{ae^{-ax}}{(1 + e^{-ax})^2}. \tag{5.5}$$

Where $\sigma \in (0, 1)$ is the Sigmoid, $x \in (-\infty, \infty)$ is the input, a is a slope parameter [39] and $\sigma' \in (0, 0.25)$ is the derivative of the Sigmoid function.

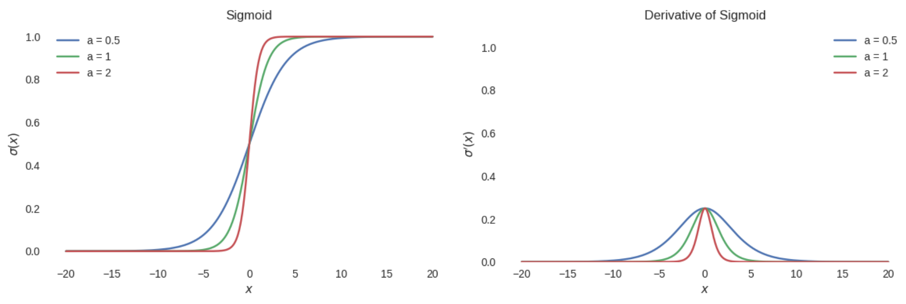


Figure 5.5: Sigmoid function and its derivative for different slope parameters.

Figure 5.5 illustrates how the Sigmoid function from equation 5.4 and its derivative from equation 5.5 changes for different slope parameters (a). Another common activation function is the Rectified Linear Unit (ReLU) function.
Definition. The Rectified Linear Unit and its derivative is defined as:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}, \tag{5.6}$$

$$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}. \tag{5.7}$$

Where $f(x) \in (0, \infty)$ is the ReLU function, $f'(x) \in (0, 1)$ is the derivative of ReLU, $x \in (-\infty, \infty)$ is the input [6].

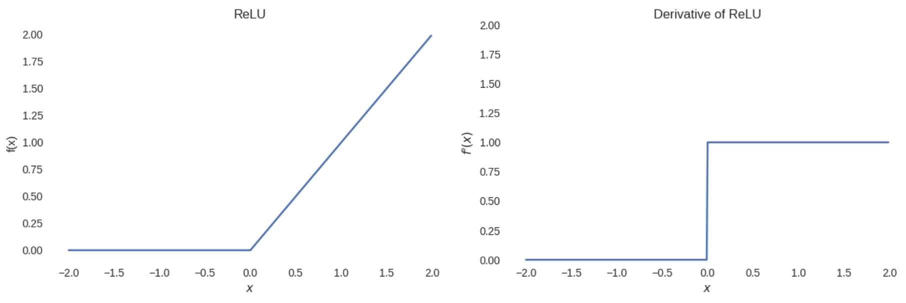


Figure 5.6: ReLU function and its derivative.

Figure 5.6 shows how the ReLU function from equation 5.6 and its derivative from equation 5.7 changes for different input values.

An activation function similar to the ReLU function is the Exponential Linear Unit (ELU) function, which similarly to ReLU, returns x for $x > 0$, However instead of returning 0 for other values, it returns a weighted exponential of the

value.
Definition. The Exponential Linear Unit and its derivative is defined as:

$$g(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha\left(e^{x} - 1\right), & \text{otherwise} \end{cases}, \tag{5.8}$$

$$g'(x) = \begin{cases} 1, & \text{if } x > 0 \\ g(x) + \alpha, & \text{otherwise} \end{cases}. \tag{5.9}$$

Where $0 < \alpha$ is a hyperparameter to define the value of the function for $x < 0$ [40], $g(x) \in (-\alpha, \infty)$ is the Exponential Linear Unit, $x \in (-\infty, \infty)$ is the input and $g'(x) \in (0, 1)$ is the derivative of the ELU function.

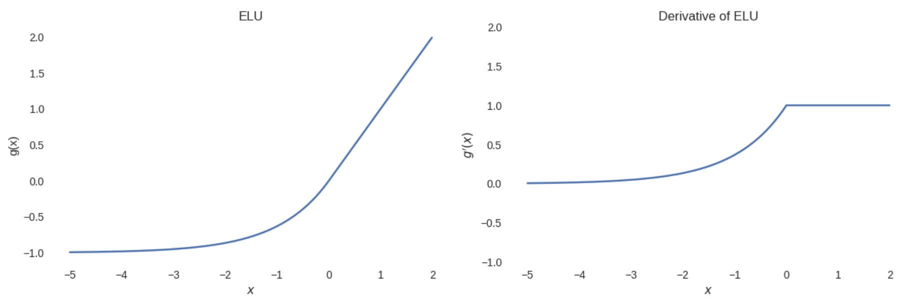


Figure 5.7: ELU function and its derivative.

Figure 5.7 shows how the ELU function from equation 5.8 and its derivative from equation 5.9 changes for different input values.

Finally the hyperbolic tangent function.
Definition. The hyperbolic tangent and its derivative is defined as:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{5.10}$$

$$\tanh'(x) = 1 - \tanh^{2}(x) \tag{5.11}$$

Where $\tanh \in (-1, 1)$ is the hyperbolic tangent, $\tanh' \in (0, 1)$ is the derivative of the hyperbolic tangent and $x \in (-\infty, \infty)$ is the input [41].

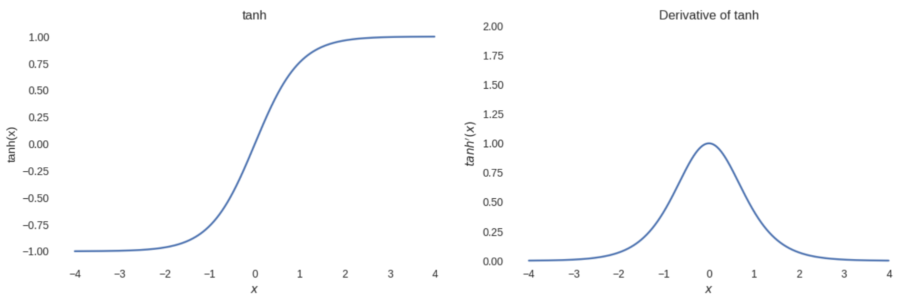


Figure 5.8: Hyperbolic tangent function and its derivative.

Figure 5.8 shows how the hyperbolic tangent $\tanh(\cdot)$ and its derivative $\tanh'(\cdot)$ changes for different input values x.

Softmax is also an activation function, but in contrast to the other activation functions mentioned above, the Softmax function is mostly used in the output layer of the model [42].
Definition. The Softmax function is defined as:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \text{ for } i = 1, ..., K \text{ and } \text{x} = (x_1, ..., x_K) \in R^K. \quad (5.12)$$

For each value (x_i) of a vector $(\mathbf{x})$, the Softmax function $(\text{Softmax}(\cdot))$ applies the exponential to said value and divides it by the sum of the exponential applied to all values of the vector. This ensures the sum of the values of the output vector to be equal to 1 [42].

5.9 Loss

This section is based on parts from the project paper [8].
The loss function of a neural network is an integral part of how machine learning algorithms are able to learn and improve their performance [43]. The loss function is used to inform the model on its performance of the task at hand. In classification tasks, the loss is a measure of whether the predicted class is equal to the true class. For an autoencoder the loss measures how similar the reconstructed output is to the input. During the training of a feed forward machine learning algorithm, the model passes the input through all its layers, to yield an output, the loss function calculates the performance of the model and the model can then adjust its parameters such that the loss is minimized [17]. The loss function to choose depends on the task at hand, for a classification problem, one possible loss function can be the cross-entropy loss function, which in the binary class case is called the binary cross entropy.

Definition. The Binary cross entropy loss function is defined as:

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)). \tag{5.13}$$

Where N is the number of samples, $y_i \in \{0, 1\}$ is the true class of sample i and $\hat{y}_i \in \{0, 1\}$ is the predicted class for sample i [44].

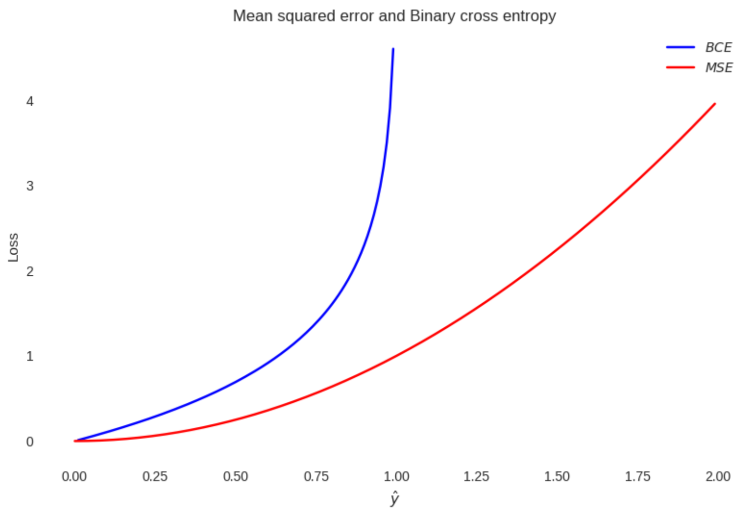


Figure 5.9: Comparison of the Mean squared error loss and the binary cross entropy loss

The blue line in figure 5.9 shows how the binary cross-entropy function from equation 5.13 changes for different predicted class values ($\hat{y}$), when $y = 0$. First notice that the loss has its minimum when the predicted class is equal to the true class ($y = \hat{y}$). One can also see that the cross-entropy loss increases exponentially as the difference between the true class and predicted class increases.

In the case of where the number of classes is not binary, the categorical cross entropy loss function can be applied.
Definition. The categorical cross entropy loss function is defined as:

$$\mathcal{L}_{\text{CCE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} p_{ic}\log(y_{ic}). \tag{5.14}$$

Where N is the number of samples, C is the number of classes, p_{ic} is a one-hot encoded label for sample i and y_{ic} is the predicted probability distribution, or the output from the model for sample i [45].The categorical cross entropy has the same features as the binary cross entropy function, where the loss has its minimum when $y = \hat{y}$ and the loss increases exponentially with the difference of the predicted class and true class.

Another common loss function is the mean squared error loss.
Definition. The mean squared error loss function is defined as:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2. \tag{5.15}$$

Where N is the number of samples, y is the label and $\hat{y}$ is the model prediction. In the case of an autoencoder, y would be the input of the model and $\hat{y}$ would be the autoencoders reconstruction of the input.

The red line in figure 5.9 illustrates how the mean squared error loss changes as the difference between the label and model prediction increases, where the label value is set to be zero. From the figure one can see that the loss has its minimum when the model prediction is equal to the label $y = \hat{y}$. Because of the squared term in equation 5.15, the loss increases rapidly when the model prediction is diverges from the label value[46]. One can also notice that the MSE loss increases slower than the cross-entropy loss, as the predicted value diverges from the true value.

6 Deep Learning Architectures

In this chapter we will go through some general deep learning architectures, we start with convolutional neural networks (CNN) and autoencoders, lastly we introduce Recurrent neural networks (RNN) Long Short-Term memory (LSTM) architecture.

6.1 Convolutional Neural Networks

This section is based on parts from the project paper [8].
Convolutional neural networks (CNNs) are neural networks that applies convolutional and downsampling layers like described in Section 5.2 and 5.4 respectively, to learn spatial or temporal patterns in the data [29] and reduce dimensionality of the data [32]. After the convolutional and downsampling, fully connected layers are applied, described in Section 5.1 [47]. Because of the convolutional layers ability to learn and extract spatial or temporal patterns, convolutional neural networks are well suited for data that contains such spatial or temporal patterns [48].

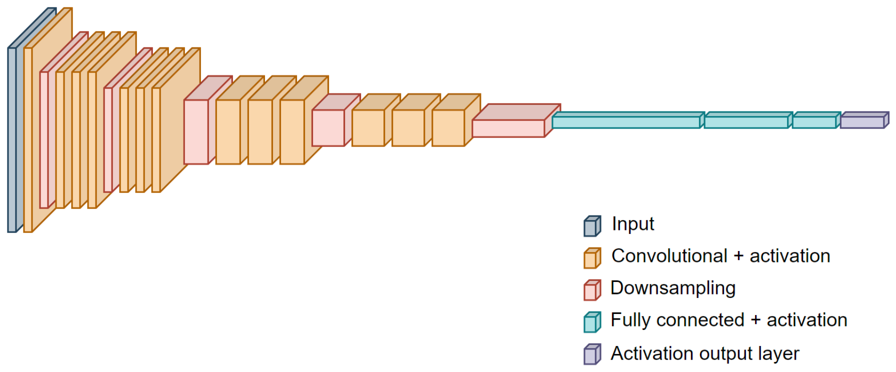


Figure 6.1: Illustration of a generic Convolutional neural network. (Heavily inspired by [49]).

Figure 6.1 shows the architecture of a generic convolutional neural network, Where the input data (Gray box) is passed through convolutional and downsampling layers (Orange and red boxes) The feature maps from the last convolutional layer is flattened and passed through fully connected layers. where the final output layer makes a prediction or classification. Also worth noticing is that after every convolutional layer and after every fully connected layer an activation function is applied to the output of said layer. It is also possible to implement a Batch Normalizing layer after every activation function, to smooth out the optimizing curve [38] and have dropout layers to avoid overfitting [34].

6.2 Autoencoder

An autoencoder is a unsupervised deep learning method consisting of an encoder and decoder, the encoder part of the autoencoder is given the input and creates a compressed representation of the given input. the compressed representation, or the encoding of the input, is then given to the decoder, which have to recreate the input of the autoencoder [50]. As the autoencoder aims to minimize the loss between the input space and the output space, and because the encoding space often has a lower dimensionality than the input space. The autoencoder has to create an efficient compressed representation of the input space. Such that the decoder can recreate the input as good as possible, while only having access to said representation [51].

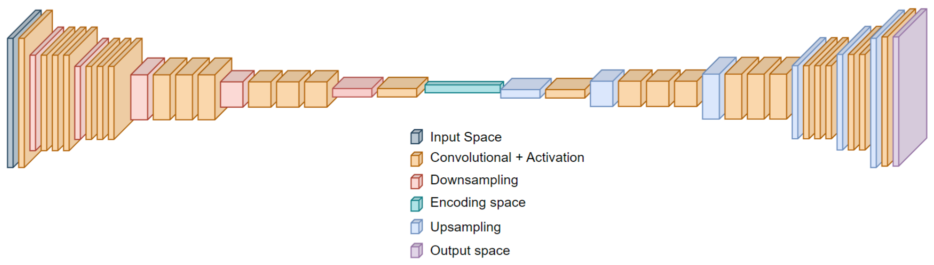


Figure 6.2: Illustration of a generic Autoencoder architecture.

Figure 6.2 shows the architectural structure of a generic convolutional neural network autoencoder. The input of the autoencoder illustrated as a gray box, is passed through a number of alternating convolution and downsampling operations, illustrated with orange and red boxes respectively, to create the encoding of the input space. As mentioned in Section 5.2, the convolution operation itself can be applied as a downsampling operation, but there are also operations like the maxpool operation from Section 5.4 that are frequently used for downsampling[52]. The encoding space is illustrated as the teal square in the middle of the figure. After the encoding is done, the decoding of the encoding space is performed, again, in an alternating fashion, convolutions, but now in combination with upsampling operations, are applied to the encoding, the upsampling is illustrated with blue boxes. The Upsampling operation can be in the form of transposed convolutional layers, shown in Section 5.3 or by using the nearest neighbor interpolation method from Section 5.5. The output of the autoencoder, illustrated as the purple box, has the same dimensionality as the input.

As mentioned earlier, an autoencoders goal is to recreate its input, but there are also other applications of the autoencoder. As the output of the autoencoder is the same size as the input, autoencoders can be used for image segmentation [53] or as generative models [50]. Autoencoders can also be applied in classification [50] or clustering tasks by utilizing the efficient, compressed representation found in the encoding space of the autoencoder [51].

6.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a type of neural network that is commonly used in deep learning applications that deal with sequential data [54]. RNNs are designed to handle variable-length sequences of data [30], where each element in the sequence is processed by the network, and the output at each time step is used as input for the next time step. This ability to process sequences of data makes RNNs particularly useful for natural language processing, speech recognition, and time-series prediction [54].

The main difference between RNNs and other neural network architectures is the ability to 'memorize' information from previous inputs, in order to better understand and predict the current input [55]. This is achieved by using recurrent connections. This allows the network to build a memory of the previous inputs and use this information to make an informed prediction at the current time step [56].

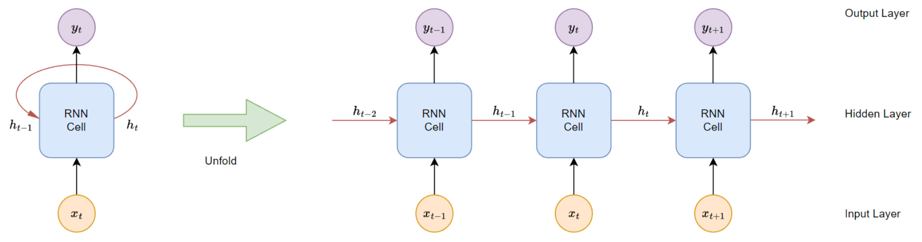
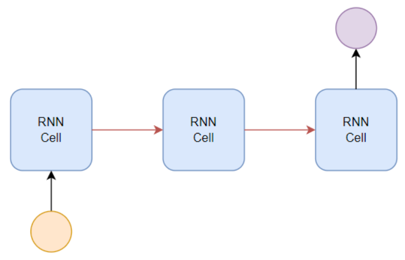
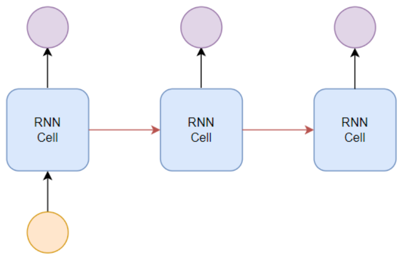


Figure 6.3: Illustration of standard Recurrent Neural Network Architecture(Heavily inspired by [57]).

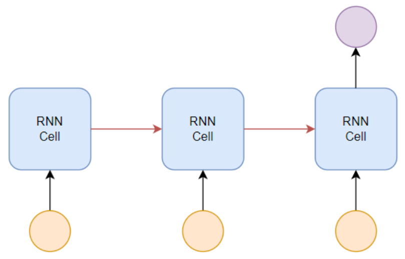
Figure 6.3 Illustrates how a standard RNN architecture would look like, and how temporal data x_t together with the information from the previous state h_{t-1} is passed to the RNN cell together create an output y_t and the input to the next hidden layer h_t.



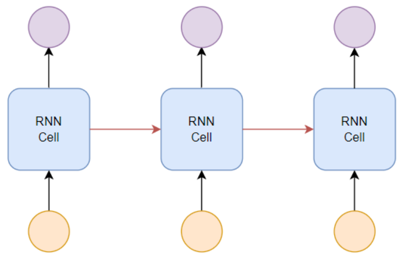
(a) One-to-one architecture.



(b) One-to-Many architecture.



(c) Many-to-One architecture.



(d) Many-to-Many architecture.

Figure 6.4: Different input/output schemes of a recurrent neural network.

RNNs can have a variable input size, aswell as a variable output size, and the different combinations of input and output sizes lets us classify RNNs into four categories, there is the one-to-one RNN, that has one input layer and one output layer (figure 6.4a), one-to-many, which has one input layer and multiple output layers (figure 6.4b). Many-to-one, which has many input layers and one output layer (figure 6.4c). Finally many-to-many has many input layers and many output layers(figure 6.4d) [58].

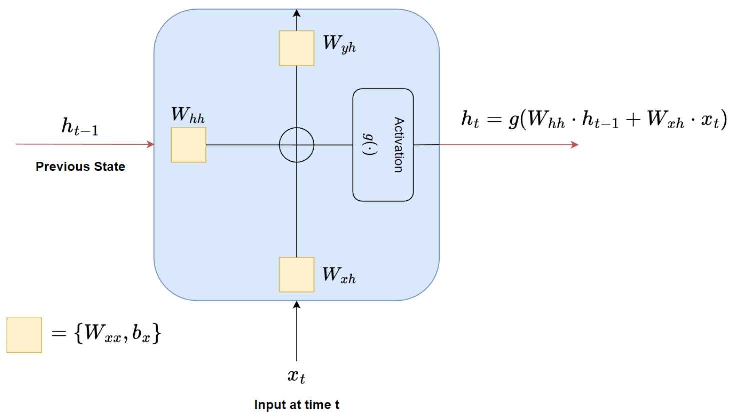


Figure 6.5: Illustration of the RNN cell.

Figure 6.5 illustrates how the RNN cell, has weights and biases (W_{xh}) for the input data (x_t), (W_{hh}) for the previous state h_{t-1} and (W_{yh}) for the output layer. The figure also shows how the sum of weighted input and previous state are sent to the activation function $g(\cdot)$ to form the next hidden state h_t [56].

Another feature of a Recurrent neural network is the weights of the network, illustrated as W_{hh}, W_{xh} and (W_{yh}) in figure 6.5. The same weights are applied for every timestep of the model, where a timestep is a transition between the RNN cells in figure 6.3 [56].

One of the biggest drawbacks of a standard Recurrent neural network, like the one illustrated in figure 6.3, is that its ability to store, or memorize information for long periods of time is not sufficiently good enough [56]. This is due to a problem known as vanishing gradient, which is a phenomenon that occurs when updating the same weights recursively with gradients, leading to exponentially small weight updates for longer sequences [59]. The Long Short-Term Memory (LSTM) architecture adresses the vanishing gradient problem by introducing a memory cell, that is controlled by a gating mechanism [30].

6.4 Long Short-Term Memory

Long Short-Term Memory (LSTM) network. LSTMs use a special gating mechanism that allows the network to selectively remember or forget information from previous inputs, enabling them to retain information over long periods of time [59]. LSTM networks have a memory cell, that enables the network to retain the information, what information to store within the memory cell, is controlled by three gates, the forget gate, input gate and output gate [56].

The forget gate controls how much of the information from the previous timesteps that should retain in the memory cell. The input gate controls how much of the current input x_t should be added to the memory cell [56], and the output gate controls how much of the information stored in the memory cell that should be used to form the output of the LSTM at the current timestep [60].

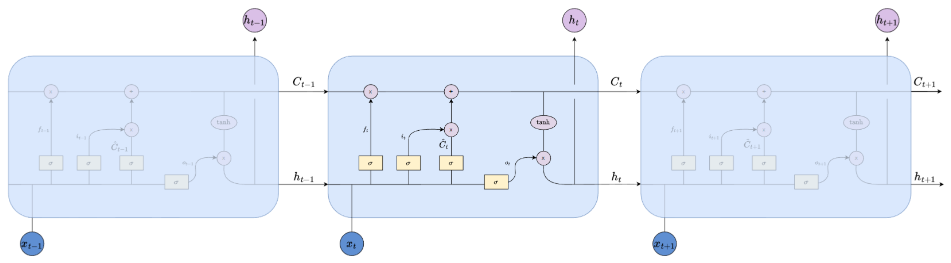


Figure 6.6: Illustration of the Long Short-Term Memory architecture (Heavily inspired by [57])

Figure 6.6 illustrates the architecture of the LSTM architecture. With the cell state C_t at timestep t and the three gating mechanisms f_t, i_t and o_t that represents the forget, input and output gates respectively. and the output h_t.
From the figure one can see that the input is merged together with the input from the previous layer. The forget gate and its weights determines what to information that is to be forgotten from the cell state [61]. The input gate decides what information that should be added to the cell state, preventing irrelevant information to be stored [62].

Part II

Method

7 Models

7.1 Deep Clustering Model

As mentioned in Section 6.2, autoencoders can be useful in clustering tasks, by utilizing the compressed representation from the embedding of the autoencoder [51]. However, by using clustering methods such as the k-means algorithm from Section 3.2, the autoencoder is prevented from learning valuable information from the clustering. The clustering module introduced in [2] allows for a learnable clustering together with the embedding. This is done by rephrasing a Gaussian Mixture Model to act as a loss function for a one hidden layer autoencoder, and therefore passing the clustering abilities of a GMM onto the simple autoencoder, creating a clustering module. The clustering module can then be trained simultaneously with a deep autoencoder, where the deep autoencoder can create non-linear representations of the input data, for the clustering module to partition.

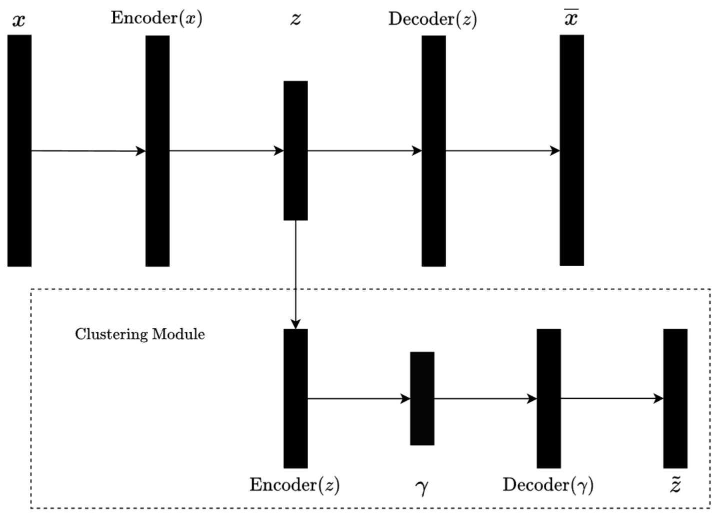


Figure 7.1: Illustration of a deep clustering architecture, the upper architecture is a deep autoencoder, the clustering module is connected to the deep autoencoders encoding space.

Figure 7.1 illustrates how a deep clustering model needs to be setup, where the upper architecture is where the input data (x) is fed to the deep autoencoder, the embedding of the deep autoencoder (z) is fed to the decoder part of the deep autoencoder to create a reconstructed output ($\overline{x}$), but is also passed to the lower architecture, where the clustering module, with the clustering abilities of a GMM, is clustering the embedding of the deep autoencoder (γ) and recreates the deep autoencoders embedding ($\tilde{z}$).

The Loss function of the deep clustering model is defined as:

$$\begin{aligned}\mathcal{L}_{\text{AE-CM}}(X|\Theta) =& \beta\sum_{i=1}^{N}||\mathrm{x}_i - \bar{\mathrm{x}}_i||^2 \\ &+ \sum_{i=1}^{N}||\mathrm{z}_i - \tilde{\mathrm{z}}_i||^2 \\ &+ \sum_{i=1}^{N}\sum_{k=1}^{K}\gamma_{ik}(1-\gamma_{ik}) \\ &+ \sum_{k=1}^{K}(1-\alpha_k)\log(\tilde{\boldsymbol{\gamma}}_k) \\ &+ \lambda||\boldsymbol{\mu}^{\mathrm{T}}\boldsymbol{\mu} - \mathrm{I}_k||\end{aligned} \tag{7.1}$$

Where x and $\bar{\mathrm{x}}$ is the input and output of the autoencoder respectively, z and $\tilde{\mathrm{z}}$ is the input and output of the clustering module, $\boldsymbol{\mu}$ is the center of the centroid, and γ is the cluster responsibility. The first two terms of the loss function:

$$L_1 = \beta\sum_{i=1}^{N}||\mathrm{x}_i - \bar{\mathrm{x}}_i||^2, \tag{7.2}$$

and

$$L_2 = \sum_{i=1}^{N}||\mathrm{z}_i - \tilde{\mathrm{z}}_i||^2, \tag{7.3}$$

are the reconstruction loss functions for the two different autoencoders. Equation 7.2 is the loss of the deep autoencoder, weighted by a constant β. The loss of the simple autoencoder in the clustering module is shown in equation 7.3. These terms penalizes the model if the reconstructed output isn't equal to the models input, this will push the model towards making efficient encodings, such that the decoders are able to reconstruct the input. The third term:

$$L_3 = \sum_{i=1}^{N}\sum_{k=1}^{K}\gamma_{ik}(1-\gamma_{ik}), \tag{7.4}$$

encourages the model to make clearer cluster assignments, this is because L_3 is only zero when γ is a one-hot vector. Minimizing L_3 leads to a sparse γ, and the resulting cluster assignments are clearer. The fourth term:

$$L_4 = \sum_{k=1}^{K}(1-\alpha_k)\log(\tilde{\boldsymbol{\gamma}}_k), \tag{7.5}$$

balances the Dirichlet prior, if all elements in α_k are non-zero, the model is incentivized to utilize all the clusters. The final term:

$$L_5 = \lambda||\boldsymbol{\mu}^{\mathrm{T}}\boldsymbol{\mu} - \mathrm{I}_k||, \tag{7.6}$$

encourages the cluster centroids to be orthogonal.Where λ is a hyperparameter that controls the importance of this term in the overall loss function and I_k is the identity matrix with dimensionality $k \times k$. When the cluster centroids are orthogonal to eachother, better separation of the clusters is achieved [63].

7.2 Temporal Neighborhood Coding

Temporal Neighborhood coding (TNC) is an unsupervised representation learning for time series method introduced by Sana Tonekaboni, Danny Eytan and Anna Goldengerg. The method defines neighborhoods in time with stationary properties and ensures that the distribution of signals from a certain neighborhood is distinguishable from distributions originating from other neighborhoods in the encoding space of the model [1].

The neighborhoods are defined as stationary regions of the signal, with size η, where η is evaluated by using the Augmented Dickey-Fuller test (ADF-test). The ADF-test is used to measure the stationarity of a given time series. The starting value for η is 1, and the p-value is calculated. If the p-value is greater than a threshold value, the test fails, and the signal is not considered stationary. If the test is passed, the value of η is increased iteratively, performing the ADF-test and calculating the p-value for every step, until the test failes. This ensures that the widest neighborhood, where the signal still remains stationary, is used [1].

The Augmented Dickey-Fuller test assumes that the time series can be expressed as a AR(p)-process [64], the time series (x_t) and its first difference (Δx_t) can then be expressed as:

$$\begin{aligned} x_t &= \sum_{j=1}^{p} \phi_j x_{t-j} + w_t, \\ \Delta x_t &= \gamma x_{t-1} + \sum_{j=1}^{p-1} \psi_j x_{t-j} + w_t. \end{aligned}$$

Where $\gamma = \sum_{j=1}^{p} \phi_j - 1$ and $\psi_j = -\sum_{i=j}^{p} \phi_i$ for $j = 2, .., p$ [65].

The ADF-test is used to test the hypothesis that the autoregressive polynomial $\phi(z)$, defined as:

$$\phi(z) = 1 - \phi_1 z - \ldots - \phi_p z^p, \tag{7.7}$$

has a unit root at 1, meaning that $\phi(1) = 0$. The hypothesis is thus $H_0 : \gamma = 1$, meaning that if the test is passed, the time series is not stationary. The test is performed through a Wald test based on $\hat{\gamma}$ [65]:

$$t_{\gamma} = \frac{\hat{\gamma}}{\text{se}(\hat{\gamma})}. \tag{7.8}$$

where $\hat{\gamma}$ is the estimated value of γ through the regression of Δx_t on $x_{t-1}, \Delta x_{t-1}, ..., \Delta x_{t-p+1}$ and se$(\cdot)$ denotes the standard error [65].

Further, the assumption that windows within a particular neighborhood share similar characteristics. Windows outside of the neighborhood, represented as $\bar{N}_t$ are likely to contain unsimilar features, is made. If assumed that windows sampled from $\bar{N}_t$ are negative samples, a sampling bias can occur, as the sampled window from $\bar{N}_t$ might not necessarily be a negative sample, but in contrary, the sampled window might contain similar characteristics to the reference window. Thus, by wrongfully labeling the $\bar{N}_t$-sampled window as a negative sample, the bias occurs, and it can impact the learning of the model [66]. To avoid the sampling bias, windows sampled from $\bar{N}_t$ are considered unlabeled. The classifier learns by sampling labeled, positive, data from a reference neighborhood and unlabeled data from another neighborhood$(\bar{N}_t)$. The unlabeled data is then considered as a mixture of both positive and negative samples with a positive class prior π [1]. Every sample drawn from the reference neighborhood is considered a positive sample, and is unit weighted. The samples drawn from $\bar{N}_t$ is considered as combination of both positive and negative samples, with weights w and $1 - w$, where w is the probability of a window sampled from $\bar{N}_t$ to be a positive sample [1].

When the neighborhoods are defined. An encoder is trained such that the representations of samples from the same neighborhood are distinguishable from samples from other neighborhoods. This is done by a encoder that maps the input to a lower dimensional space followed by a discriminator that receives two samples from the encoding and predicts the probability of the two samples to originate from the same neighborhood. [1].

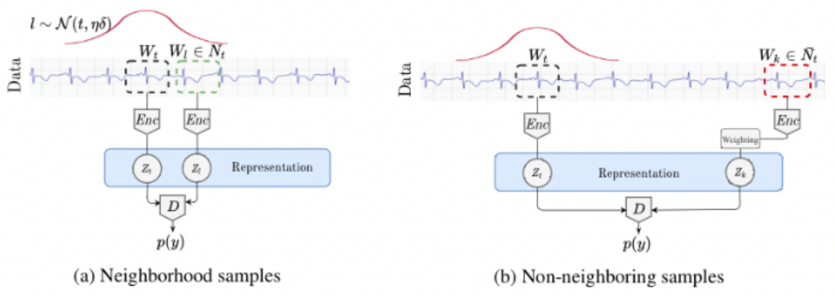


Figure 7.2: Illustration of the TNC algorithm [1].

Figure 7.2 shows how the TNC algorithm first defines the neighborhood distribution of the sample window, the sample window is the area enclosed by the black dotted square. Then in the left plot, the encoder is fed two samples originating from the same neighborhood distribution, the encodings of the two samples are fed to the discriminator, where the discriminator is predicting the probability of the two samples being from the same neighborhood. In the right plot, the encoder is given one sample from the reference neighborhood, again, the area enclosed by the black dotted square, and a sample from another distribution, originating from the area enclosed by the red dotted line. Here the encoding of the other sample is weighed to adjust for the sampling bias.

As the discriminator yields the probability of whether the two samples originate from the same neighborhood or not, the loss function is thus opted to make the discriminator accurate, where it will be close to 1 if the two representations originates from neighboring samples and 0 if the two representations originate from non-neighboring samples. While also the non-neighboring samples are weight adjusted to adjust for positive, non-neighboring samples.

$$\begin{aligned}\mathcal{L} = &-\mathbb{E}_{W_t \sim X}\Big[\mathbb{E}_{W_l \sim N_t}\big[\log \mathcal{D}(Z_t, Z_l)\big] \\ &+ \mathbb{E}_{W_k \sim \bar{N}_t}\big[(1 - w_t) \times \log(1 - \mathcal{D}(Z_t, Z_k)) \\ &+ w_t \times \log \mathcal{D}(Z_t, Z_k)\big]\Big]. \end{aligned} \tag{7.9}$$

Where $\mathbb{E}_{x \sim y}$ is the expected value of x sampled from y, $\mathcal{D}(Z_t, Z_k)$ is the output of the discriminator, where the input is the encoding of W_t and W_k and w is the weights that consider the positive samples in the non neighboring distribution [1].

8 Proposed Data and Model Architectures

8.1 Data

In the thesis we have used two different datasets. one of the datasets we have used is the same as in [1], the simulated dataset. The simulated dataset was made to replicate a very long, non-stationary and high frequency time series, with three underlying dynamics and four different states. As this dataset is simulated, the four different states are balanced, when it comes to appearance of the four different states in the files [1].The other dataset is the same as we used in the project paper [8]. A dataset consisting of audiofiles recorded by hydrophones located in different locations in the seabed around Svalbard. This dataset has multiple recordings of different mammal species, aswell as other recordings, such as moving ice and recordings of boat engines. The individual audiofiles vary in length, where the shortest audiofile is 6.04 minutes and the longest audiofile is 17.1 minutes. The labeling of the audiofiles varies, as the labeling for the appearance of boats in the audiofiles are good, where every timestep is binary labeled according to whether there is a boat present in the audiofile at that timestep or not, on the other hand, the appearance of other phenomenons in the dataset, such as a recording of a mammal, is only labeled once for the entire dataset, meaning that we know which audiofile that contains a recording of a bearded seal, but we do not now where in the audiofile the recording is, nor how long the recording is. Based on this, we decided to instead of cluster each mammal class, we decreased the number of classes to the nature of the sounds, some of the mammals, like the walrus and belugas, can make sounds that resembles clicking noises, while the bowhead, for example, makes more of a continuous song. These two sound characteristics makes up two classes, we also needed a class for no sound, in addition to the presence of boats, this adds up to 6 different classes. Namely: clicking, song, no sound, all three with and without the presence of boat engines.

The dataset we used from the TNC was preprocessed in the same way as in the paper. The simulated data is generated using a hidden Markov model, with 4 different states. The resulting signals are also normalized to have zero mean and standard deviation equal to one [1].
The audiofiles are preprocessed by first converting the audiofiles into spectrograms. The spectrograms are further averaged such that every timestep in the spectrograms is equal to one second of audio, further, the highest frequencies are clipped such that only the first 80 frequency bins are used. Finally the spectrograms were normalized to have zero mean and standard deviation equal to one.

8.2 Proposed Models

During training and inference of the models the simulated data was split into windows of length 50. The labels of the windows were defined as the label occurring most within each window. The spectrograms were similarly split into windows of length 14, which is equal to 14 seconds of audio per spectrogram. The labels denoting occurrences of boats in the spectrograms were decided if boat audio occurred in more than 50% of the window, all other was set to no boat. The labels denoting if other phenomena occurred in the audiofiles were assigned to all the windows of each spectrogram. So if the audiofile was labeled to contain audio of walrus, then all windows of that audiofile was labeled to contain walrus audio. All datasets were split into training and testing sets with a ratio of 80/20.

In addition to the models used in [1], we also did a further build on the TNC models, such that they follow the structure of an autoencoder, with an encoder and a decoder. This gives us a total of four models, one clustering model utilizing convolution, one clustering model utilizing the LSTM architecture, and one encoder utilizing convolutional and one encoder utilizing the LSTM architecture. In our experiment, all of the models are applied to both the datasets.

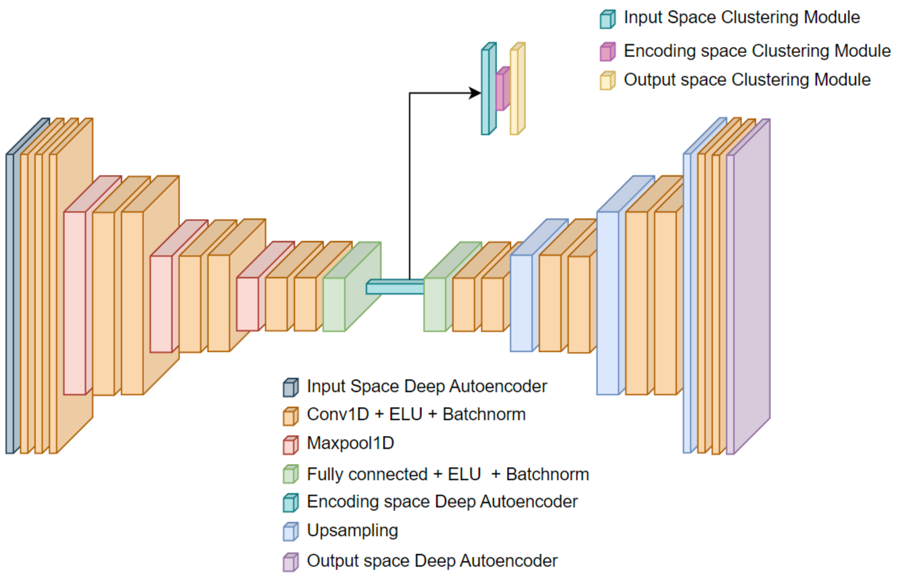

Input Space Clustering Module
Encoding space Clustering Module
Output space Clustering Module
Input Space Deep Autoencoder
Conv1D + ELU + Batchnorm
Maxpool1D
Fully connected + ELU + Batchnorm
Encoding space Deep Autoencoder
Upsampling
Output space Deep Autoencoder

Figure 8.1: Architecture of the autoencoder utilizing one dimensional convolutions, pooling and upsampling layers for the encoder/decoder together with the implemented Clustering module applied on the deep autoencoders encoding space.

Figure 8.1 illustrates the deep clustering model utilizing convolutional layers to encode and decode the data. This deep clustering model was applied on the simulated data and the spectrogram data, the only difference between the autoencoder used for the two different datasets is the shape of the input/output space, due to the nature of the two datasets. The orange boxes represent one-dimensional convolutional layers with the ELU activation function and batch normalization. The red boxes represent maxpool operations. The green boxes represent fully connected layers, including a hidden layer and dropout layer. The encoding space is passed to the clustering module that partitions the data. The decoder starts with a fully connected layer, followed by convolutional layers and upsampling layers using nearest neighbors interpolation like described in Section 5.5, illustrated as the blue boxes. The loss function for the model is shown in equation 7.1, and the model was optimized using the Adam optimizer from Section 4.2 with a learning rate equal to 10^{-3} and a weight decay factor of 10^{-5}.

The deep clustering model, that utilizes the LSTM structure described in Section 6.3 to encode and decode the input data has similar structure to the model illustrated in figure 8.1, however, the convolutional layers, aswell as the pooling and batch normalization layers in the encoder and decoder has been replaced by a the LSTM architecture. The input data is passed to a LSTM encoder, which encodes the data into a smaller dimensionality, the LSTM encoding is similarly to the CNN deep clustering module passed to a one hidden layer fully connected layer, which also has a dropout layer with probability of 0.5. The encoding space is also passed to a clustering module for clustering, and to another one hidden layer fully connected layer with dropout, followed by a LSTM decoder, which reconstructs the models input. The loss function is the same as for the CNN, equation 7.1 together with a Adam optimizer with learning rate 10^{-3}, and a weight decay factor of 10^{-5}.

Part III

Results and Discussion

9 Experiments and Results

9.1 Experiments

All the models were trained for 200 epochs, but the weights that achieved the lowest loss for the testing data was saved and used for inference.

9.2 Results

9.2.1 TNC Architecture

In this section we will present the results from applying the TNC architecture to the simulated dataset aswell as the spectrograms. The encoder utilizing convolutions was tested for different kernel sizes and the encoder utilizing the LSTM architecture was tested for different number of hidden layers. The plots for all tests follow the same structure, consisting of 3 parts, the upper plot shows the test sample applied to the model, the middle part shows the encoding of the model, and the bottom part of the plots will show spectral clustering applied to the encoding of the model, this will act as an unsupervised segmentation of the input data.

Simulated Data

As mentioned in Section 8.2 both the encoder utilizing convolutional layers and the LSTM architecture was applied to the simulated data. First we will present the performance of the encoder that used convolutional layers. The encoder utilizing convolutional layers was tested with kernel size 3 and 5. In this section we will see the following sample of the simulated data:

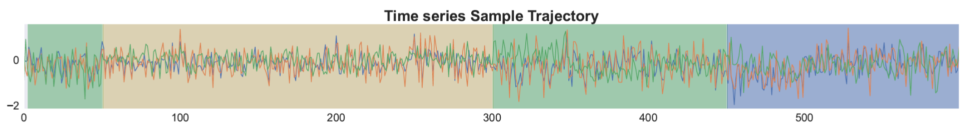


Figure 9.1: Test sample from simulated data where the background is colored dependent on the label of the timeseries at that specific point in time

The plot in figure 9.1 shows a window of one of the testing data time series, for the simulated dataset. The window has a length of 600 timesteps. The background of the plot illustrates the label of the time series at every timestep, the change in color illustrates a transition from one class to another, signifying different underlying patterns or characteristics within the data. This test sample is used for inference of the models trained on the simulated data.

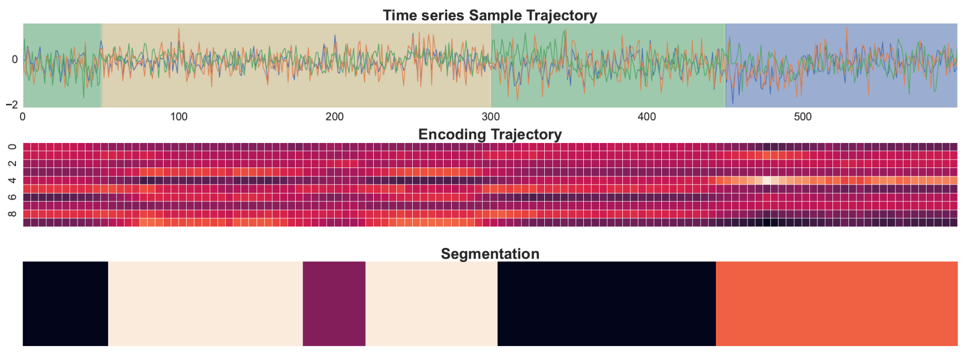


Figure 9.2: Labeled sample together with encoding, from inference of TNC architecture utilizing convolutional layers with kernel size 3.

The upper plot of figure 9.2 shows the time series window with the different labels as mentioned in Section 9.2.1, the middle plot shows a heatmap of the encoding space of the trained encoder with kernel size 3, as it processes the input, where black is the lowest value and white is the highest value. The choice of encoding size is 10, as in the paper [1]. The bottom plot shows the spectral clustering of the encoding.

The upper plot reveals four distinct transitions in the labels of the time series: starting with green, followed by yellow, returning to green, and finally transitioning to blue. From analyzing the heatmap corresponding to the area where the timeseries is labeled green, approximately the first 50 timesteps and the period between timestep 300 and 450. We can see that the indices 5 and 8 of the heatmap are brightest. This indicates that the model has learned the underlying patterns of the data when the label is green, as we expect the same indexes to light up when the model receives input consisting similar patterns. For the yellow area in the timeseries plot, we can observe that indices 3,5,8 and 9, are bright, the same indices are bright for the entire yellow section of the timeseries where the label is yellow, except in the are around time index 200. Where all indicies are almost equally bright. This can indicate uncertainties in the model regarding the yellow label of the dataset. For the last 150 timesteps, when the timeseries is labeled blue, we observe that index 4 is brightening up.

Something worth noticing is that when the time series label is yellow and green, we observed that indicies 5 and 8 was prominent for both labels, this might indicate that the model struggles with distinguishing the two labels from one another, as the model uses both these indicies to represent different underlying patterns. However when the blue label is occuring in the timeseries, we observed that index 4 is prominent, which in this example only occured for the blue label, indicating that the model is able to separate the blue label from the green and the yellow.

When analyzing the segmentation of the timeseries, the results are as expected, where we can observe that the segmentation is black for for the green part of the time series, and orange for the blue part of the time series, which is correctly segmented. We also observe that for the most part, the segmentation of the yellow part of the time series is correctly, however the area around the 200 mark of the time series is wrongly segmented. This is expected because of the encoding in that area.

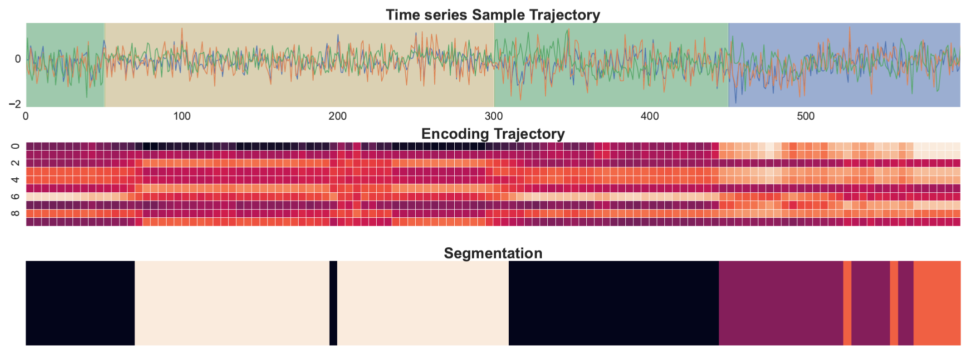


Figure 9.3: Labeled sample together with encoding, from inference of TNC architecture utilizing convolutional layers with kernel size 5.

Figure 9.3 shows how the results of the convolutional encoder, but with the kernel size increased from 3 to 5. From the segmentation in this plot we can observe that although the wrongfully segmentation of the timeseries around the 200 mark of the timeseries has decreased in size, the correctly segmented part of the blue part in figure 9.2 of the timeseries has now been introduced to uncertainty in the model, resulting in wrongly segmented areas in this part. So although the model has decreased its error for the yellow label, the uncertainty for the blue label has increased.

Now onto the encoder utilizing the LSTM architecture, in the paper, 100 hidden layers was used. We start of with the encoder utilizing the LSTM architecture with 50 hidden layers.

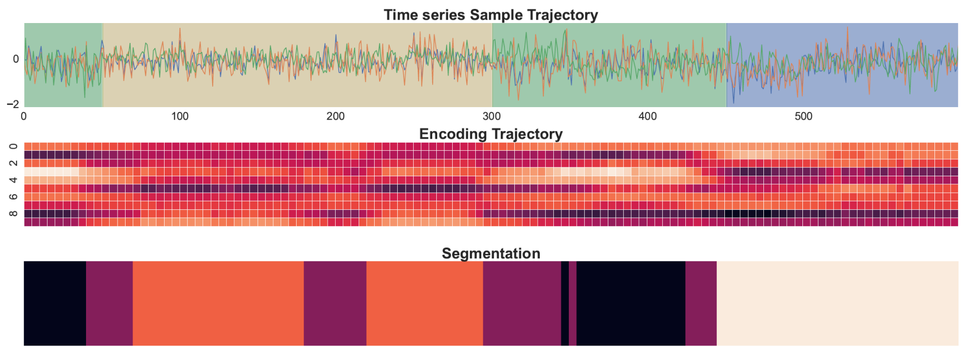


Figure 9.4: Labeled sample together with encoding, from inference of TNC architecture utilizing the LSTM architecture with 50 hidden layers.

Figure 9.4 Shows the reference time series, the 50 hidden layer encoding of the time series and the segmentation of the encoding. From the encoding plot, we observe similarities from the plots in figure 9.2, where specific indicies are brightening up dependent on the label of the time series. Where indices 3 and 4 are prominent for the time series where the green label occurs, 7, 8 and 9 are bright for the yellow labeled part of the time series and indexes 0 and 1 are bright for the blue part of the time series.
The segmentation of the timeseries shows some similarities to the CNN experiments, but there are some differences, first we observe that the consistency of the segmentation for the green labeled part of the time series, has declined, as for this network, the segmentation of the green labeled parts of the time series, is almost equally much purple as black, for the yellow part of the time series, we observe that for the most part, the segmentation is orange, with exceptions of the 200 mark and in the areas where the model transitions from green to yellow and the transition from yellow to green. For the area where the label is blue, the segmentation is correct.

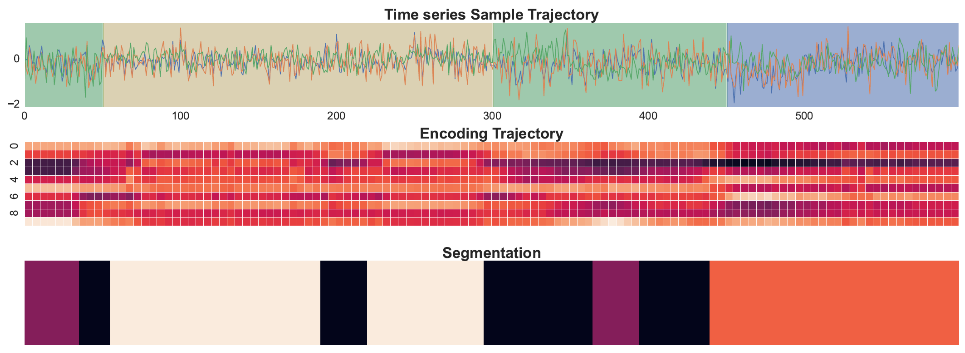


Figure 9.5: Labeled sample together with encoding, from inference of TNC architecture utilizing the LSTM architecture with 100 hidden layers.

Figure 9.5 Shows the timeseries, encoding and segmentation, when the data is applied to the 100 hidden layer LSTM encoder. From this plot one can see that the segmentation errors has decreased for the yellow part of the time series, but it seems like the segmentation of the green labeled parts of the time series has decreased in quality, as the segmentation for the green labeled part in the start seems to be segmented to the purple color, while the segmentation of the green labeled part after time step 300 is for the most part being black. We can also see similar patterns from the 50 hidden layer LSTM encoder, where the segmentation is wrong in the transitions between labels.

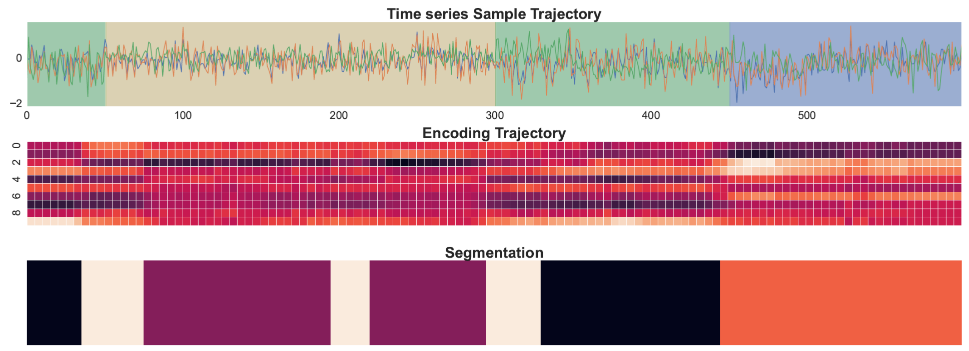


Figure 9.6: Labeled sample together with encoding, from inference of TNC architecture utilizing the LSTM architecture with 256 hidden layers.

Figure 9.6 shows the timeseries, encoding and segmentation where a LSTM encoder with 256 hidden layers have been used. Again the same pattern of the segmentation in the transitions between labels is prominent, on the other hand, the segmentation quality of the green labeled parts of the time series has improved.

From the plots above, it seems that the encoders utilizing convolutions, outperforms the LSTM architecture when it comes to segmenting the test sample of the timeseries, where the encoder using kernel size 3 has the best performance, we can also notice that all segmentation's were incorrect for the 200 mark of the test time series, indicating that perhaps that part of the timeseries has different underlying patterns then expected for that specific label and can perhaps be considered an outlier. We can also notice that for all the encoders utilizing the LSTM architecture, the segmentation's are incorrect in the area where the timeseries transitions between different labels.

Spectrogram Data

In this section the TNC encoders are applied to the spectrograms of the audiofiles we received from the Norwegian polar institute, similarly to the simulated data, we have used one sample from the testing data to analyse the testing of the performance of the models:

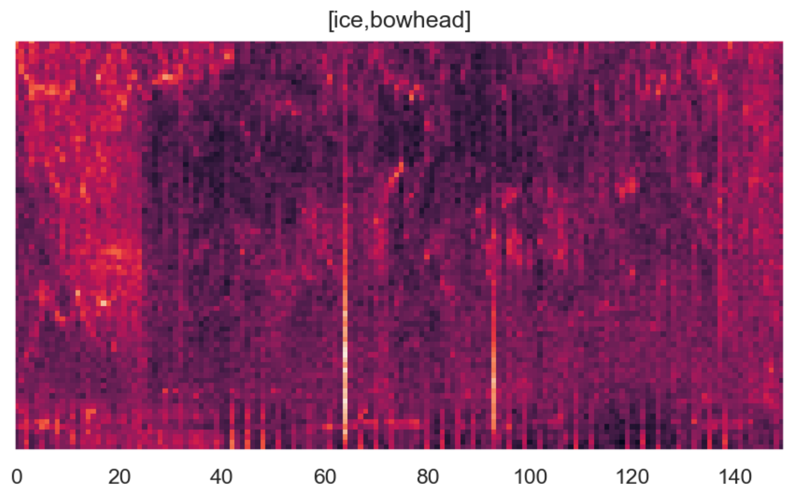


Figure 9.7: Spectrogram test sample. The label of the spectrogram is ice and bowhead

Figure 9.7 shows the test sample used below, the label of the audiofile is ice and bowhead, meaning the audiofile has recordings of moving ice, aswell as recordings of a bowhead We also know that there is no recordings of boats in this audiofile. As the labeling of the spectrogram data is not for every timestep, but for the file as a whole, it is hard to say for sure what everything that occurs in the spectrogram is, but we can still see some occurrences in the spectrogram that have similar features to eachother, like the two spikes that occur, in the spectrogram between timestep 60 and 100.

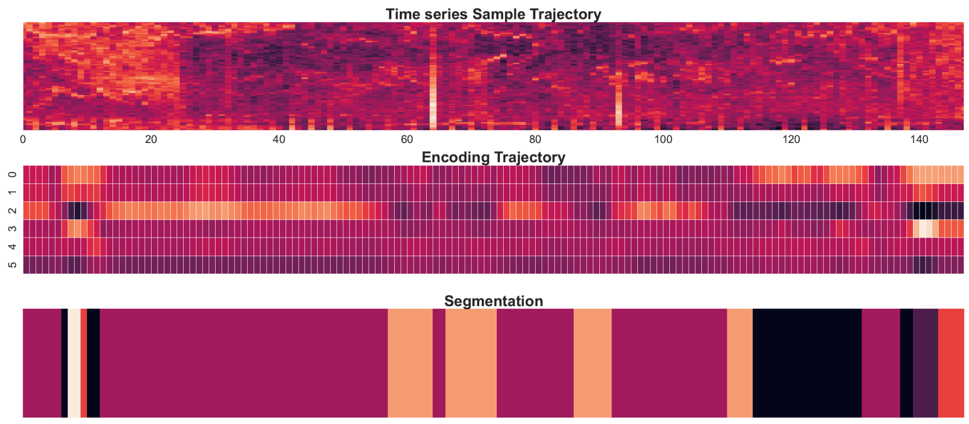


Figure 9.8: Spectrogram test sample together with encoding, from inference of TNC architecture utilizing convolutional layers with kernel size 3.

Figure 9.8 shows the same spectrogram as in figure 9.7. Together with the encoding of the TNC encoder with kernel size 3 in the middle plot, and the segmentation of the image using spectral clustering in the bottom plot. We can see that for almost the entirety of the input, the model yields one prominent index, index 2. This is not unexpected as for the most part of the spectrogram, there seems to be only noise, however, we can see for the two spikes in the spectrogram, between time index 60 and 100, that there is a pause in the prominence of index two, this might indicate that the model has learned to separate the spikes from other features in the spectrograms, we can see from the segmentation image in the bottom, that almost for the entire first part of the image, the segmentation is consistent to the purple color, this is until the spikes occur, and segmentation changes color. We also see the same pattern for the other spike in the spectrogram.

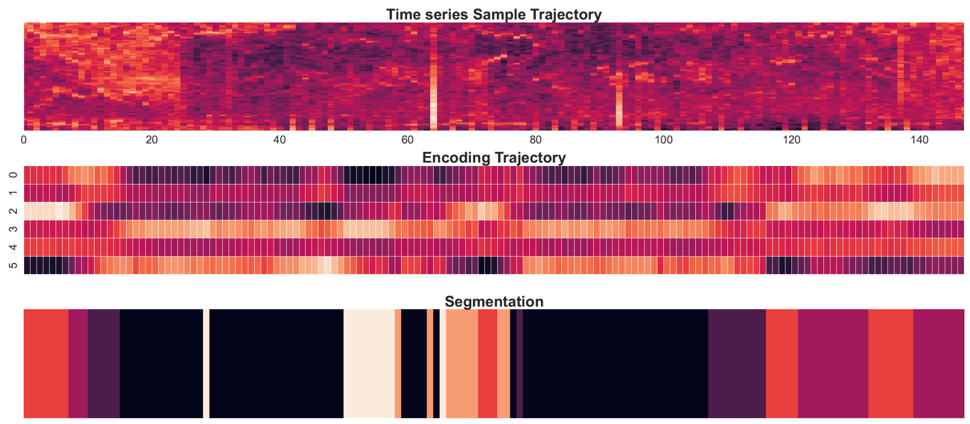


Figure 9.9: Spectrogram test sample together with encoding, from inference of TNC architecture utilizing the LSTM architecture with 256 hidden layers.

Figure 9.9 shows how the TNC architecture utilizing the LSTM architecture with 256 hidden layers performed on the spectrogram test sample. We can observe that for the first spike in the spectrogram there is a change in the segmentation, but the same change does not occur for the second spike in the spectrogram, indicating uncertainties, in the models prediction of that specific phenomenon.

9.2.2 Deep Clustering Architecture

In this section we will present the results when applying the deep clustering architecture to the same datasets as in Section 9.2.2. We have created two different deep clustering models, one model utilizing convolutions with kernel size 3, and one using the LSTM architecture with 256 hidden layers, as these performed best with the TNC architecture.

	Simulated Data		Spectrogram Data	
	DCM_{CNN}	DCM_{LSTM}	DCM_{CNN}	DCM_{LSTM}
α	1.1	1.1	1.3	1.2
β	4	4	15	50
λ	1	4.5	4.5	4.5
Orthonormal	False	False	True	True

Table 9.1: Table of hyperparameters applied in the Loss function for the different deep clustering models.

Table 9.1 shows the final choice of the hyperparameters of the clustering modules loss function, that created the following outputs. Where DCM_{CNN} denotes a deep clustering model that utilizies convolutions, and DCM_{LSTM} denotes a deep clustering model that utilizes the LSTM architecture.

Simulated Data

For the deep autoencoder utilizing convolutions we did the same approach as in [2], where the deep clustering model that processed the simulated data, was pretrained for 5 epochs, where only the reconstruction loss of the deep autoencoder, equation 7.2, is backpropagated. After the first 5 epochs, the clustering module was initialized by using the Kmeans algorithm on the embedded dataset. Followed by the training of the full model using the loss function $\mathcal{L}_{\text{AE-CM}}$ from equation **??** for the remaining 195 epochs. This was only done for this model.

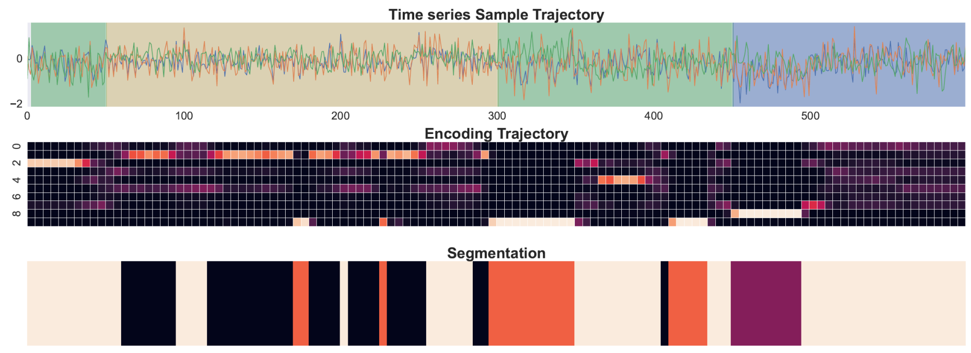


Figure 9.10: Labeled sample together with encoding, from inference of deep clustering architecture utilizing convolutional layers with kernel size 3.

Figure 9.10 shows how the performance of the deep clustering model utilizing convolutions with kernel size 3. The testing sample is the same as for the TNC models. From the segmentation, we can observe that the model performs well at the start of the timeseries, where the first 50 timesteps are uniformly segmented to the white class. The transition in labels from green to yellow in the timeseries approximately at timestep 50, results in a similar shift in the segmentation, with a transition from white to black. However, during the span of the yellow label, we can observe that the encoding starts to fluctuate between different indexes, resulting in a similar fluctuation in the segmentation, which is not ideal, as we want the heatmap of the embedding to remain stable as long as the labels remain unchanged. On the other hand, for the most part of the yellow timeseries, the embedding shows strong activity at index 1, which corresponds to the black segmentation. After the yellow period of the timeseries, the label is changed back to green again. As mentioned earlier, ideally, the embedding would then transition back to index 2, which is the index with strong activations in the previous part of the timeseries when the label was green. This is not the case, as the embedding shows a strong activation in index 9, which results in the incorrect segmentation of the first part green labeled signal. Eventually the segmentation transitions back to white, which is the correct label. For the last part of the timeseries, we observe the total opposite, where the embedding first has distinguished the blue part of the signal from the yellow and green signal, as the embedding is purple, but then the segmentation transitions back to white, which corresponds to the green label.

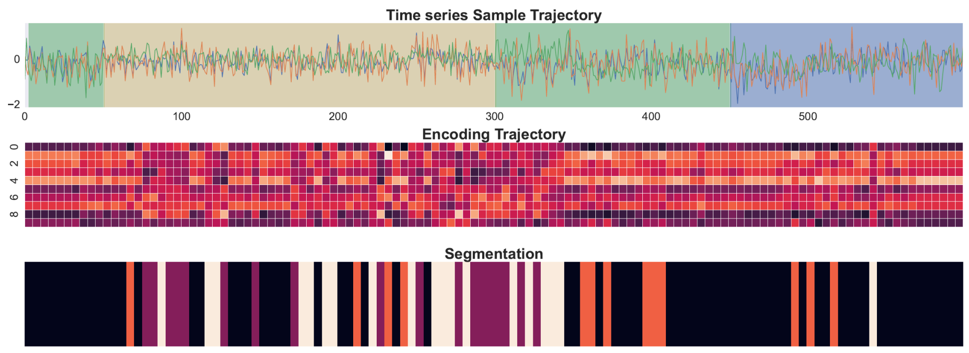


Figure 9.11: Labeled sample together with encoding, from inference of deep clustering architecture utilizing the LSTM architecture with 256 hidden layers.

Figure 9.11 shows the deep clustering architecture utilizing the LSTM architecture with 256 hidden layers. We can see that the model has learned some of the underlying patterns as the segmentation is consistent with the label in the start of the time series, in the transition to the yellow label and during the yellow phase of the time series, we can see that the heatmap of the encodings are fluctuating alot, resulting in equal fluctuations in the segmentation, this indicates uncertainties in the model when it comes to the yellow label, further, we observe that when the timeseries transitions back to the green phase, the heatmap of the encodings stabelizes, and the segmentation is for the most part back to black again, however, we can see that in the transition to the blue phase, the encodings do not change, resulting in an flawed segmentation for the blue part of the timeseries.

Spectrogram Data

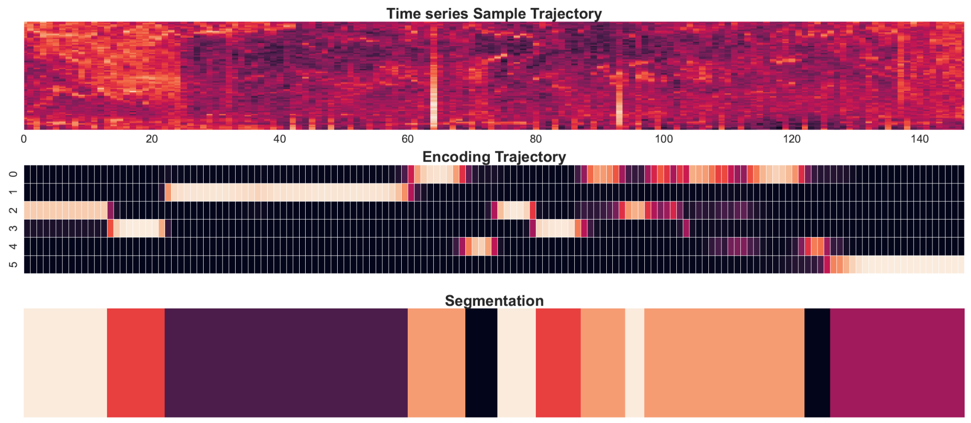


Figure 9.12: Spectrogram test sample together with encoding, from inference of deep clustering architecture utilizing convolutions with kernel size 3.

Figure 9.12 shows the spectrogram test sample, together with the deep clustering embedding of the input image, together with the segmentation of the timeseries. We can observe from the embedding that the embedding indexes are more distinct compared to the TNC architectures embedding, we can also observe that for both the spikes occuring in the spectrogram, index 0 is active, indicating that the model maybe have learned to distinguish this pattern from others, we can also observe that the noise in the start of the spectrogram also is segmented differently than the other noises in the spectrogram.

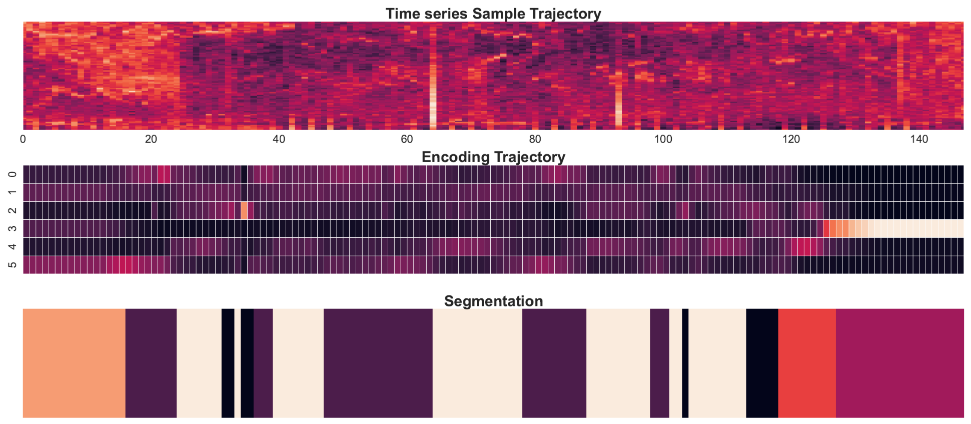


Figure 9.13: Spectrogram test sample together with encoding, from inference of deep clustering architecture utilizing the LSTM architecture with 256 hidden layers.

Figure 9.13 shows the performance of the deep clustering model, using the LSTM architecture with 256 hidden layers. We can see that the noise in the beginning of the spectrogram has a consistent segmentation, this can mean that the model has learned the underlying pattern of that phenomenon, we can also observe for both of the spikes in the spectrogram, the segmentation is the same, indicating that the model might have learned the pattern correlated with the spikes.

9.2.3 Clustering

As the plots in the two previous sections only consider one test sample, we have also performed a clustering of the entire simulated dataset, this is to give us some insight in the overall performance of the different models. In addition to clustering the data, we also calculated the homogenity score of the clusters. The homogenity score, is a measure of the homogenity of the classes assigned to each cluster [67]. The clustering of the dataset was performed with the kmeans algorithm applied to the encodings of the various models. Because of the sparse labels of the spectrograms, the clustering and homogenity scores was only applied to the simulated data.

	Homogenity Score
TNC_{LSTM}	0.916
TNC_{CNN}	0.808
DCM_{LSTM}	0.508
DCM_{CNN}	0.472

Table 9.2: Table of Homogenity scores for the different architectures.

Table 9.2 Shows the homogenity score for 4 different models, The TNC algorithm that utilized convolutions with kernel size 3. The TNC algorithm that utilized the LSTM architecture with 256 hidden layers, aswell as the deep clustering model, that utilized convolutions with kernel size 3 and the LSTM architecture with 256 hidden layers. As we can see from the table, the highest homogenity score was achieved with the TNC architecture that used the LSTM architecture, and the lowest scoring model is the deep clustering model, utilizing convolutions. We can also see that both the TNC models are the highest scoring architectures.

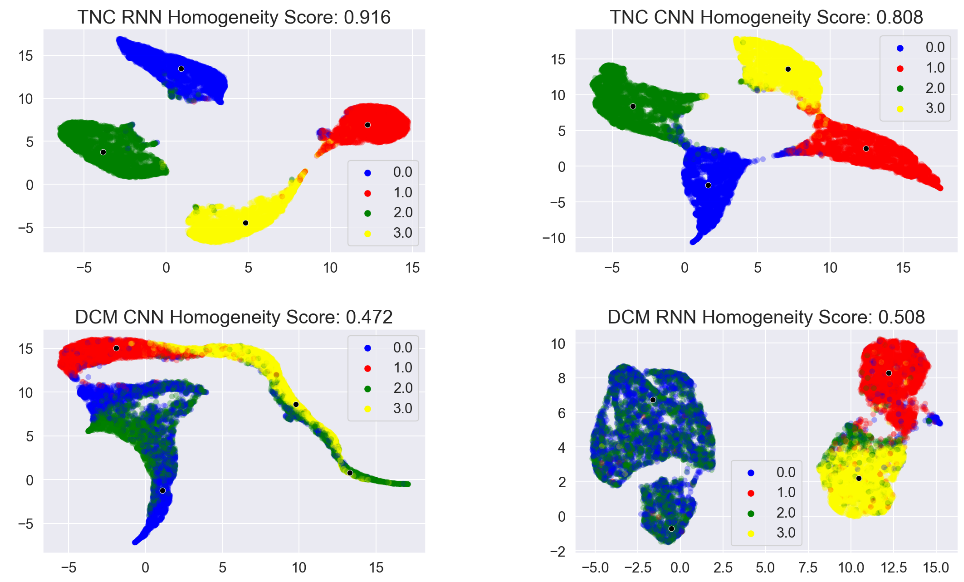


Figure 9.14: Clustering of all simulated data, for all models, upper left is the embedding of the TNC model utilizing the LSTM architecture with 256 hidden units, upper right is the clustered embedding of the TNC architecture utilizing convolutions with kernel size 3, bottom left is the embedding of the Deep clustering model utilizing convolutions and kernel size 3, finally the bottom right plot is the clustered embedding of the deep clustering model utilizing the LSTM architecture with 256 hidden units.

Figure 9.14 shows the clustering performance of the 4 different models. The colors of the points are consistent with the background of the timeseries used as a test sample in the previous section. The upper left plot is the clustering of the embedding from the TNC architecture with LSTM, the upper right plot is the embedding of the TNC architecture using convolutions. the bottom left plot is the clustered embedding of the deep clustering module using convolutions and the bottom right is the clustered embedding of the deep clustering module using the LSTM architecture. As we can see from the plot above, the TNC model that utilized the LSTM architecture has performed best in separating the different classes. Which corresponds well with the homogenity scores showed in table 9.2. We can also see that neither of the deep clustering models have done a very good separation of the classes, but some classes are better separated than others.

10 Discussion and Conclusion

10.1 Discussion

In Chapter 9, we present our results regarding the segmentation of both the simulated dataset and the spectrograms of the submarine recordings. We explore the performance of different architectures, focusing on the Temporal neighborhood coding and deep clustering model.

In Section 9.2.1 we presented the TNC architectures performance, showing that it performs well in the unsupervised segmentation of the simulated dataset.Most of the models demonstrate satisfactory performance on the test sample, except for a specific area in the timeseries where all models incorrectly segment. This discrepancy could be explained as that part of the time series being an outlier, but we dont know for sure. The TNC architecture's performance on the spectrograms, as presented in Section 9.2.1 is difficult to quantify, this is because of the sparse labeling of the data. Therefore, we rely on visual analysis. This was heavily based on two distict features within the spectrogram used as a test sample. Based on that we found that the segmentation of the encoding of the TNC architecture using convolutions were consistent with the occuring of the spikes in the spectrogram. This leads us to believe that the model has learned to distinguish that feature of the spectrograms from other features.

The Deep clustering models performance on the simulated data, showed in Section 9.2.2, also showed promise for the test sample, although not as consistent as the TNC architectures, these models still managed to somewhat distinguish the different underlying patterns from eachother. Where on the specific test sample, the deep clustering module utilizing convolutions seemed to outperform the one utilizing the LSTM architecture. The deep clustering models performance on the spectrograms was presented in Section 9.2.2, again, because of the limited label information we have, we did a visual inspection of the resulting segmentation. Based on the visual inspections, we can say that the deep clustering algorithms performed well, where both the convolutional network aswell as the recurrent, managed to segment the spikes in the spectrograms, we can also see a more detailed segmentation of the noise in these spectrograms, atleast compared to the TNC architecture that utilized convolutions.

Finally we performed a clustering of the entire simulated time series data, this was done to get a more full scale overview of the different models performance, and not make any conclusion based on one test sample alone. From the clustering we saw, that the TNC algorithms performed much better compared to the deep clustering models. One reason for this might be the nature of the dataset, as the TNC architecture is made for the purpose of time series analysis, its fair to assume that the TNC has an inherent advantage over the Deep clustering model. We also observe that for both the TNC architecture and for the Deep clustering models, the models utilizing the LSTM architecture outperformed the models using convolutions. This is most likely due to the nature of the data, as the simulated dataset is resembling a multivariate time series, the LSTM's outperformance of the convolutional counterpart is expected [68, 69]. From the clustering of the deep clustering algorithms embedding, we notice that the yellow and red classes are best separated, while the blue and green classes are hardly separated at all, we see the same patterns if we consider the segmentation of the test time series sample. Here we can see that the segmentation of the time series during the green and blue phase of the plot, are segmented equally.
Overall, our findings shed light on the performance of different architectures in the segmentation of the simulated dataset and spectrograms. The TNC architectures demonstrate solid results, while the deep clustering models show promise but exhibit slightly less consistency. These insights provide valuable information for further research.

10.2 Future Directions

Further investigation into the effect of tuning the hyperparameters within the loss function introduced in Section 7.1 is crucial for enhancing the performance of our model. The optimization of these parameters significantly influences the model's ability to capture relevant patterns, achieve accurate reconstructions,

and produce meaningful clusters. Therefore, we need to tune the hyperparameters β, α, and λ to achieve optimal performance.
Tuning β allows us to control the trade-off between an accurate reconstruction and clustering performance. Also by tuning α, thus regulating the impact of the priors, finally λ which controls the relevance of the orthogonality term. The optimizing of these parameters is important for achieving the best possible performance.

Another future direction might be to get access to better labeled data or utilizing the limited label information we already have in a better way. Either way, by improving our labels, we can improve our evaluation of the models segmentation abilities.

By gaining more label information and by tuning our hyper parameters, we can gain a more comprehensive understanding of the model's performance and thus perform better segmentation of our recordings.

10.3 Conclusion

In this thesis, we looked into the application of Temporal Neighborhood Coding (TNC) and a deep clustering model for segmenting submarine recordings. The segmentation would need to be in an unsupervised manner due to the limited label information available.

TNC, a method designed to encode the underlying states of multivariate, nonstationary time-series [1], played a crucial role in our research. We aimed to capture the underlying patterns and temporal dependencies present in the submarine recordings by leveraging its encoding capabilities. We also implemented a deep clustering model, made possible by using the clustering module introduced in [2], which incorporates the clustering capabilities of a Gaussian mixture model into a simple autoencoder. This is accomplished by rephrasing the Gaussian mixture model as a loss function.

Because of the lack of label information, we also segmented a simulated dataset in an unsupervised manner, the segmentation of the simulated dataset showed promising results, where the TNC architecture outperforms the deep clustering method. However the performance on the submarine recordings is more challenging to quantify due to the limited label information available. Nevertheless, our visual analysis of the segmentation results provided valuable insights into the capabilities of both TNC and the deep clustering model.

Moving forward, a more comprehensive exploration of hyperparameter tuning could lead to even better segmentation results. Additionally, access to improved labeled data would enable a more robust evaluation of our models' segmentation abilities on the submarine recordings.

In conclusion, our investigation of Temporal Neighborhood Coding and the deep clustering model for the unsupervised segmentation of submarine recordings and simulated data has provided valuable insights. The application of these methods has demonstrated their efficacy in successfully segmenting the data by capturing underlying patterns and structures. However, further research is needed to fully optimize and refine these methods.

Bibliography

[1] Sana Tonekaboni, Danny Eytan, and Anna Goldenberg. *Unsupervised Representation Learning for Time Series with Temporal Neighborhood Coding*. 2021. arXiv: `2106.00750 [cs.LG]`.

[2] Ahcène Boubekki et al. "Joint optimization of an autoencoder for clustering and embedding." eng. In: *Machine learning* 110.7 (2021), pp. 1901–1937. ISSN: 0885-6125.

[3] Hojin Lee et al. "Feature selection practice for unsupervised learning of credit card fraud detection." English. In: *Journal of Theoretical and Applied Information Technology* 96.2 (Jan. 2018). Funding Information: This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2017-2015-0-00403) supervised by the IITP (Institute for Information communications Technology Promotion) and by Institute for Information communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0132-16-1004, Development of Profiling-based Techniques for Detecting and Preventing Mobile Billing Fraud Attacks) Publisher Copyright: © 2005 – ongoing JATIT LLS., pp. 408–417. ISSN: 1992-8645.

[4] Krzysztof J Cios et al. *Data Mining: A Knowledge Discovery Approach*. eng. New York, NY: Springer, 2007. ISBN: 9780387333335.

[5] Roy Nuary Singarimbun, Erna Budhiarti Nababan, and Opim Salim Sitompul. "Adaptive Moment Estimation To Minimize Square Error In Backpropagation Algorithm." In: *2019 International Conference of Computer Science and Information Technology (ICoSNIKOM)*. 2019, pp. 1–7. DOI: `10.1109/ICoSNIKOM48755.2019.9111563`.

[6] Marina Adriana Mercioni and Stefan Holban. "The Most Used Activation Functions: Classic Versus Current." In: *2020 International Conference on Development and Application Systems (DAS)*. 2020, pp. 141–145. DOI: `10.1109/DAS49615.2020.9108942`.

[7] James H. McClellan, Ronald W. Schafer, and Mark A. Yoder. *DSP First, second edition*. Pearson Education Limited, 2015. ISBN: 9781292113869.

[8] Tor Kjøtrød. *Comparison of state-of-the-art methods for vessel detection in submarine recordings*. FYS-3740 Project Paper in applied physics and mathematics. 2022.

[9] Nasser Kehtarnavaz. “CHAPTER 7 - Frequency Domain Processing.” In: *Digital Signal Processing System Design (Second Edition)*. Ed. by Nasser Kehtarnavaz. Second Edition. Burlington: Academic Press, 2008, pp. 175–196. ISBN: 978-0-12-374490-6. DOI: https://doi.org/10.1016/B978-0-12-374490-6.00007-6. URL: https://www.sciencedirect.com/science/article/pii/B9780123744906000076.

[10] Tom Bäckström. *Spectrogram and the STFT - Introduction to Speech Processing - Aalto University Wiki*. [Online; accessed 15. Aug. 2022]. Aug. 2022. URL: https://wiki.aalto.fi/display/ITSP/Spectrogram+and+the+STFT.

[11] Minlei Liao et al. “Cluster analysis and its application to healthcare claims data: a study of end-stage renal disease patients who initiated hemodialysis.” eng. In: *BMC nephrology* 17.24 (2016), pp. 25–25. ISSN: 1471-2369.

[12] Christopher M Bishop. *Pattern recognition and machine learning*. eng. New York, 2006.

[13] Rui Xu. *Clustering*. eng. Piscataway, New Jersey, 2015.

[14] Sigurd Løkse. *Leveraging Kernels for Unsupervised Learning*. eng. 2020.

[15] Dami Choi et al. *On Empirical Comparisons of Optimizers for Deep Learning*. 2020. arXiv: 1910.05446 [cs.LG].

[16] Daniel Svozil, Vladimír Kvasnicka, and Jirí Pospichal. “Introduction to multi-layer feed-forward neural networks.” In: *Chemometrics and Intelligent Laboratory Systems* 39 (1 1997), pp. 43–62. ISSN: 0169-7439. URL: https://www.sciencedirect.com/science/article/pii/S0169743997000610.

[17] Yimin Yang et al. “Recomputation of the Dense Layers for Performance Improvement of DCNN.” In: *IEEE* 42 (11 2019), pp. 2912–2925. ISSN: 0162-8828. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8718406.

[18] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition*. eng. San Diego, CA, USA: Elsevier Science, 2008. ISBN: 1597492728.

[19] Ethem Alpaydin. *Introduction To Machine Learning, third edition*. The MIT Press, 2014. ISBN: 9780262028189.

[20] Abhijit Ghatak. *Deep Learning with R*. eng. Singapore, 2019.

[21] Nadikatla Chandrasekhar and Samineni Peddakrishna. “Enhancing Heart Disease Prediction Accuracy through Machine Learning Techniques and Optimization.” In: *Processes* 11.4 (2023). ISSN: 2227-9717. DOI: 10.3390/pr11041210. URL: https://www.mdpi.com/2227-9717/11/4/1210.

[22] Diederik P Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization.” eng. In: (2014).

[23] Narendra Kumar et al. *Advance Concepts of Image Processing and Pattern Recognition: Effective Solution for Global Challenges*. eng. Transactions on Computer Systems and Networks. Singapore: Springer, 2022. ISBN: 9811693234.

[24] Chigozie Nwankpa et al. *Activation Functions: Comparison of trends in Practice and Research for Deep Learning*. 2018. arXiv: 1811.03378 [cs.LG].

[25] Xiaofei He et al. “Coherence Enhancing Diffusion for Discontinuous Fringe Patterns with Oriented Boundary Padding.” eng. In: *Intelligence Science and Big Data Engineering. Image and Video Data Engineering*. Vol. 9242. Lecture Notes in Computer Science. Switzerland: Springer International Publishing AG, 2015, pp. 362–369. ISBN: 9783319239873.

[26] Ignazio Gallo, Shah Nawaz, and Alessandro Calefati. “Semantic Text Encoding for Text Classification Using Convolutional Neural Networks.” In: *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*. Vol. 05. 2017, pp. 16–21. DOI: 10.1109/ICDAR.2017.323.

[27] Michael W Berry et al. “Arabic Phonemes Recognition Using Convolutional Neural Network.” eng. In: *Soft Computing in Data Science*. Vol. 1100. Communications in Computer and Information Science. Singapore: Springer Singapore Pte. Limited, 2019, pp. 262–271. ISBN: 9789811503986.

[28] Aqeel Anwar. “What is Transposed Convolutional Layer? - Towards Data Science.” In: *Medium* (Mar. 2023). ISSN: 4056-3111. URL: https://towardsdatascience.com/what-is-transposed-convolutional-layer-40e5e6e31c11.

[29] Gavnet Singh Chada, Jan Niclas Reimann, and Andreas Schwung. “Generalized Dilation Neural Networks.” In: (2019). URL: https://arxiv.org/pdf/1905.02961.pdf.

[30] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[31] Naoki. “Up-sampling with Transposed Convolution - Naoki - Medium.” In: *Medium* (Oct. 2022). URL: https://naokishibuya.medium.com/up-sampling-with-transposed-convolution-9ae4f2df52d0.

[32] Gangming Zhao, Jingdong Wang, and Zhaoxiang Zhang. “Random Shifting for CNN: a Solution to Reduce Information Loss in Down-Sampling Layers.” In: *PROCEEDINGS OF THE TWENTY-SIXTH INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE*. Ed. by C Sierra. 26th International Joint Conference on Artificial Intelligence (IJCAI), Melbourne, AUSTRALIA, AUG 19-25, 2017. 2017, pp. 3476–3482. ISBN: 978-0-9992411-0-3.

[33] Augustus Odena, Vincent Dumoulin, and Chris Olah. “Deconvolution and Checkerboard Artifacts.” In: *Distill* 1.10 (Oct. 2016), e3. ISSN: 2476-0757. DOI: 10.23915/distill.00003.

[34] Geoffrey E. Hinton et al. *Improving neural networks by preventing co-adaptation of feature detectors*. 2012. arXiv: 1207.0580 [cs.NE].

[35] Xue Ying. “An Overview of Overfitting and its Solutions.” In: *Journal of Physics: Conference Series* 1168.2 (2019), p. 022022. DOI: 10.1088/1742-

6596/1168/2/022022. URL: https://dx.doi.org/10.1088/1742-6596/1168/2/022022.

[36] Pierre Baldi and Peter J Sadowski. "Understanding Dropout." In: *Advances in Neural Information Processing Systems*. Ed. by C.J. Burges et al. Vol. 26. Curran Associates, Inc., 2013. URL: https://proceedings.neurips.cc/paper_files/paper/2013/file/71f6278d140af599e06ad9bf1ba03cb0-Paper.pdf.

[37] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[38] Shibani Santurkar et al. *How Does Batch Normalization Help Optimization?* 2019. arXiv: 1805.11604 [stat.ML].

[39] Simon Haykin. *Neural networks : a comprehensive foundation, second edition*. eng. Upper Saddle River, New Jersey, USA, 1999.

[40] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. 2016. arXiv: 1511.07289 [cs.LG].

[41] Swalpa Kumar Roy et al. *LiSHT: Non-Parametric Linearly Scaled Hyperbolic Tangent Activation Function for Neural Networks*. 2023. arXiv: 1901.05894 [cs.CV].

[42] Kunal Banerjee et al. *Exploring Alternatives to Softmax Function*. 2020. arXiv: 2011.11538 [cs.LG].

[43] Qi Wang et al. "A Comprehensive Survey of Loss Functions in Machine Learning." In: *Ann. Data. Sci.* 9.2 (Apr. 2022), pp. 187–212. ISSN: 2198-5812. DOI: 10.1007/s40745-020-00253-5.

[44] Cai Guo et al. "Multi-Stage Attentive Network for Motion Deblurring via Binary Cross-Entropy Loss." eng. In: *Entropy (Basel, Switzerland)* 24.10 (2022), p. 1414. ISSN: 1099-4300.

[45] A Rusiecki. "Trimmed categorical cross-entropy for deep learning with label noise." eng. In: *Electronics letters* 55.6 (2019), pp. 319–320. ISSN: 0013-5194.

[46] George Seif. "Understanding the 3 most common loss functions for Machine Learning Regression." In: *Medium* (Feb. 2022). URL: https://towardsdatascience.com/understanding-the-3-most-common-loss-functions-for-machine-learning-regression-23e0ef3e14d3.

[47] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. 2015. arXiv: 1511.08458 [cs.NE].

[48] Huimei Han, Ying Li, and Xingquan Zhu. "Convolutional neural network learning for generic data classification." eng. In: *Information sciences* 477 (2019), pp. 448–465. ISSN: 0020-0255.

[49] Kenneth Leung. "How to Easily Draw Neural Network Architecture Diagrams." In: *Medium* (Sept. 2022). ISSN: 6613-8875. URL: https://towardsdatascience.com/how-to-easily-draw-neural-network-architecture-diagrams-a6b6138ed875.

[50] Dor Bank, Noam Koenigstein, and Raja Giryes. *Autoencoders*. 2021. arXiv: 2003.05991 [cs.LG].

[51] Chunfeng Song et al. “Auto-encoder Based Data Clustering.” In: *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*. Ed. by José Ruiz-Shulcloper and Gabriella Sanniti di Baja. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 117–124. ISBN: 978-3-642-41822-8.

[52] Prasant Kumar. “Max Pooling, Why use it and its advantages. - Geek Culture - Medium.” In: *Medium* (Jan. 2022). URL: https://medium.com/geekculture/max-pooling-why-use-it-and-its-advantages-5807a0190459.

[53] Evan M. Yu et al. *An Auto-Encoder Strategy for Adaptive Image Segmentation*. 2020. arXiv: 2004.13903 [eess.IV].

[54] Robin M. Schmidt. *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview*. 2019. arXiv: 1912.05911 [cs.LG].

[55] Andrej Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks*. [Online; accessed 8. May 2023]. Mar. 2022. URL: http://karpathy.github.io/2015/05/21/rnn-effectiveness.

[56] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. eng. 2012th ed. Vol. 385. Studies in Computational Intelligence. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2012. ISBN: 9783642247965.

[57] Christopher Olah. *Understanding LSTM Networks – colah’s blog*. [Online; accessed 9. May 2023]. Sept. 2022. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs.

[58] Amjad Almahairi et al. “Augmented CycleGAN: Learning Many-to-Many Mappings from Unpaired Data.” In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 195–204. URL: https://proceedings.mlr.press/v80/almahairi18a.html.

[59] Navin Kumar Manaswi. “RNN and LSTM.” In: *Deep Learning with Applications Using Python : Chatbots and Face, Object, and Speech Recognition With TensorFlow and Keras*. Berkeley, CA: Apress, 2018, pp. 115–126. ISBN: 978-1-4842-3516-4. DOI: 10.1007/978-1-4842-3516-4_9. URL: https://doi.org/10.1007/978-1-4842-3516-4_9.

[60] Sepp Hochreiter and Jürgen Schmidhuber. “LSTM can Solve Hard Long Time Lag Problems.” In: *Advances in Neural Information Processing Systems*. Ed. by M.C. Mozer, M. Jordan, and T. Petsche. Vol. 9. MIT Press, 1996. URL: https://proceedings.neurips.cc/paper_files/paper/1996/file/a4d2f0d23dcc84ce983ff9157f8b7f88-Paper.pdf.

[61] Junyoung Chung et al. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. 2014. arXiv: 1412.3555 [cs.NE].

[62] Kamilya Smagulova and Alex Pappachen James. “A survey on LSTM memristive neural network architectures and applications.” eng. In: *The*

European physical journal. ST, Special topics 228.10 (2019), pp. 2313–2324. ISSN: 1951-6355.

[63] Wei Wang et al. “Clustering With Orthogonal AutoEncoder.” In: *IEEE ACCESS* 7 (2019), pp. 62421–62432. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2916030.

[64] Efstathios Paparoditis and Dimitris N. Politis. “The asymptotic size and power of the augmented Dickey-Fuller test for a unit root.” eng. In: *Econometric reviews* 37.9 (2018), pp. 955–973. ISSN: 0747-4938.

[65] Robert H Shumway and David S Stoffer. *Time Series Analysis and Its Applications: With R Examples*. eng. Springer Texts in Statistics. Cham: Springer International Publishing AG, 2017. ISBN: 9783319524511.

[66] Ching-Yao Chuang et al. “Debiased Contrastive Learning.” In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 8765–8775. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/63c3ddcc7b23daa1e42dc41f9a44a87
Paper.pdf.

[67] Daniyal Kazempour, Peer Kröger, and Thomas Seidl. “Towards an Internal Evaluation Measure for Arbitrarily Oriented Subspace Clustering.” In: *2020 International Conference on Data Mining Workshops (ICDMW)*. 2020, pp. 300–307. DOI: 10.1109/ICDMW51313.2020.00049.

[68] Sangdi Lin and George C. Runger. “GCRNN: Group-Constrained Convolutional Recurrent Neural Network.” In: *IEEE Transactions on Neural Networks and Learning Systems* 29.10 (2018), pp. 4709–4718. DOI: 10.1109/TNNLS.2017.2772336.

[69] Narek Abroyan. “Convolutional and recurrent neural networks for real-time data classification.” In: *2017 Seventh International Conference on Innovative Computing Technology (INTECH)*. 2017, pp. 42–45. DOI: 10.1109/INTECH.2017.8102422.