GDAŃSK UNIVERSITY OF TECHNOLOGY

TASK Quarterly 28 (2) 2024

Two architectures of neural networks in distance approximation

W. Wojtyna[1], J. Sławiński, R. Tonga

Faculty of Electronics, Telecommunications and Informatics, Gdańsk University of Technology,
Gabriela Narutowicza 11/12, 80-233, Gdańsk, Poland

https://doi.org/10.34808/tq2024/28.2/c

Abstract

In this research paper, we examine recurrent and linear neural networks to determine the relationship between the amount of data needed to achieve generalization and data dimensionality, as well as the relationship between data dimensionality and the necessary computational complexity. To achieve this, we also explore the optimal topologies for each network, discuss potential problems in their training, and propose solutions. In our experiments, the relationship between the amount of data needed to achieve generalization and data dimensionality was linear for feed-forward neural networks and exponential for recurrent ones. Our findings indicate that computational complexity exhibits an exponential growth pattern as the dimensionality of the data increases. We also compared the networks' accuracy in both distance approximation and classification to the most popular alternative, Siamese networks, which outperformed both linear and recurrent networks in classification despite having lower accuracy in exact distance approximation.

Keywords:

Neural Networks (NN), Distance, Topology

[1]E-mail: w.r.wojtyna@gmail.com

1. Introduction

1.1. Objectives and Motivations

The first goal of this study was to investigate the amount of data required to achieve model generalization. For this, we needed to find optimal topologies for linear (feed-forward) and recurrent architectures. We also aimed to compare the abilities of those two networks against the industry standard in comparative classification - Siamese networks. Additionally, we tried to examine the relation between the dimensionality and the computational complexity needed.

1.2. State of the Art

In recent years, neural networks, as universal function approximators, have seen significant development, becoming one of the most important tools in the field of artificial intelligence. Their applications span a wide range of domains, from image processing (e.g., through convolutional neural networks [1]), to speech recognition (thanks to models like Deep Speech [2]), and even text generation (as seen in the recently popular models such as GPT-3 [3]).

Initially, neural networks were used for simple problems, such as classifying data with a small number of features. However, over the years, their ability to learn complex patterns, along with the dynamic development of optimization techniques and the increase in computational power, has led to their widespread application in more challenging tasks. Examples include convolutional neural networks (CNNs) used in image processing tasks and recurrent neural networks (RNNs), which play a key role in sequential processing, such as speech recognition and text analysis.

Due to their ability to represent data in the form of multidimensional vectors, neural networks enable effective modeling of relationships between different observations, which is crucial in many applications. For instance, Long Short-Term Memory (LSTM) recurrent networks have solved issues related to long-term memory in sequences, problems that were previously unsolvable for standard RNNs.

Modern neural networks are evolving toward more complex models, such as hybrid networks and graph-based models (Graph Neural Networks - GNN) [4]. These models combine the advantages of different network architectures, enabling more advanced approaches to analyzing data with complex structures, such as social or molecular networks. Moreover, the application of Reinforcement Learning (RL) [5], in combination with neural networks, allows for more efficient modeling of decision-making sequences, which is crucial in problems like robotics or autonomous systems.

In particular, Siamese networks, which are used for classification tasks requiring the comparison of two objects, are gaining popularity in the context of problems related to distance approximation and similarity measures. By using shared weights, these networks are more efficient and scalable compared to traditional approaches.

In this paper, in addition to the simplest, feedforward, fully connected neural networks, we will also discuss recurrent network architectures.

One of the main applications of distance approximation using neural networks is image processing, where comparing vector representations of images in feature space is key. Research on the use of neural networks for image analysis and comparison is based, among other things, on metrics such as the Euclidean and cosine distance. These studies have gained significance in the context of tasks such as content-based image retrieval and pattern recognition [6].

1.3. Overview of the Use of Distance Measures in Machine Learning

1.3.1 Euclidean distance

Euclidean distance is one of the most widely used distance metrics, especially for numerical data. For two points $p = (p_1, p_2, \ldots, p_n)$ and $q = (q_1, q_2, \ldots, q_n)$ in an n-dimensional space, the Euclidean distance is defined as:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2} \tag{1}$$

this metric is intuitive as it measures the "straight-line" distance between points. Euclidean distance is commonly used in algorithms like *k-NN* (k-Nearest Neighbors) and *K-means*, where finding points with minimal distance is crucial.

Applications:

- k-NN algorithms
- K-means clustering
- Principal Component Analysis (PCA)

1.3.2 Manhattan (City Block) Distance

The Manhattan distance, also known as the city block distance, is calculated as the sum of the absolute differences of the coordinates. For points p and q, it is computed

using the formula:

$$d(p,q) = |p_1 - q_1| + |p_2 - q_2| + \cdots + |p_n - q_n| \quad (2)$$

this distance is particularly useful in high-dimensional spaces and in situations where movement is restricted along axes, making it less susceptible to noise than the Euclidean distance.

Applications:

- k-NN algorithms in high-dimensional spaces
- Clustering in irregular coordinate systems

1.3.3 Minkowski Distance

The Minkowski distance is a generalization of both the Euclidean distance and Manhattan distance. It is defined as:

$$d(p,q) = \left(\sum_{i=1}^{n} |p_i - q_i|^p \right)^{1/p} \quad (3)$$

where he parameter p allows for customization of the distance measure to specific application requirements. For $p = 2$, the measure becomes the Euclidean distance, while for $p = 1$, it corresponds to the Manhattan distance.

Applications:

- Optimization in matching algorithms, depending on data specifics
- Clustering algorithms

1.3.4 Cosine Similarity

Cosine similarity measures the angular difference between vectors, ignoring their magnitude. It is particularly useful when directions are more important than the actual distances between points. It is defined as:

$$d(p,q) = 1 - \frac{p \cdot q}{||p||\ ||q||} \quad (4)$$

this measure is widely used in text analysis, such as TF-IDF vectors, as we often care about contextual similarity rather than absolute differences.

Applications:

- Text analysis
- Natural Language Processing (NLP)
- Recommendation systems

1.3.5 Hamming Distance

Hamming distance measures the difference between two binary vectors. It is defined as the number of positions at which the two vectors have different values. For vectors p and q, their Hamming distance is:

$$d(p,q) = \sum_{i=1}^{n} I(p_i \neq q_i) \quad (5)$$

where I is the indicator function. Hamming distance is used primarily for discrete data, such as comparing binary strings or genotypes, in computational biology.

Applications:

- Algorithms operating on binary data
- Biometrics, DNA analysis
- Hopfield neural networks

1.4. Recurrent Neural Network (RNN) Operation Algorithm

Recurrent neural networks (RNNs) (Fig. 1) offer a promising method for approximating distances between vectors due to their ability to process sequential data element by element. They feature an architecture built with feedback connections, allowing them to consider information from previous processing stages in the final result.

The RNN operation algorithm can be presented as follows:

1. **Sequence Element Fetching**: At the beginning of the algorithm, one element is fetched from the data sequence. Let us denote it as x_t, where t is the element index.
2. **Element Processing**: The fetched element x_t is then processed by the internal fully connected layers of the RNN. The processing result is denoted as h_t.
   This can be mathematically expressed as:
   $$h_t = W_h x_t + U_h h_{t-1} + b_h \quad (6)$$
   where W_h is the weight matrix of the hidden layer, U_h is the weight matrix of the recurrent connections, b_h is the bias vector of the hidden layer, h_{t-1} is the value of the hidden state from the previous step (for $t = 1$, h_{t-1} is initialized to zero).
3. **Output Forwarding**: The processing result h_t from the hidden layer is fed back to the network input. This is a key feature of RNNs, as it allows the network to consider the context of previous sequence elements when processing the current element.
4. **Step Repetition**: Steps 1-3 are repeated for subsequent sequence elements until all data is processed.
5. **Result Generation**: After processing the entire sequence, the RNN can generate an output, such as classification, prediction, or translation. This output is based on information from all sequence ele-

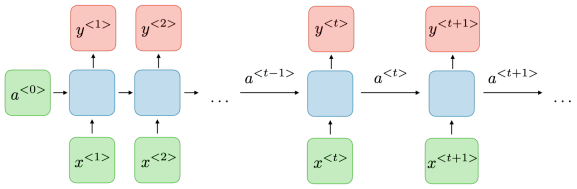

Figure 1: Architecture of a Typical Recurrent Neural Network [7].

ments, considering dependencies and context.

The operation algorithm of recurrent neural networks is relatively simple, but it allows for efficient processing of sequential data. Considering the context of previous elements, RNNs are able to achieve better results in many tasks.

2. Problems and Solutions

2.1. The Problem of Vector Scale

To accurately approximate distance measures, a model must learn two key skills: approximating the shape of the function and calibrating the scale of values.

Approximating the function structure involves the neural network understanding the relationships between input data and function values. The network must learn to recognize patterns and correctly map changes in function values in response to changes in input data.

Calibrating the scale of values, on the other hand, involves adjusting the magnitude of the output values to the correct scale. The neural network must learn to transform input values into values that match the expected range.

Simply put, approximating the structure can be compared to fitting the shape of a function's graph, while calibrating the scale is responsible for scaling this shape appropriately.

It is possible to simplify the process of calibrating the scale of model values by dividing the input values by a constant and multiplying the output values by the same constant.

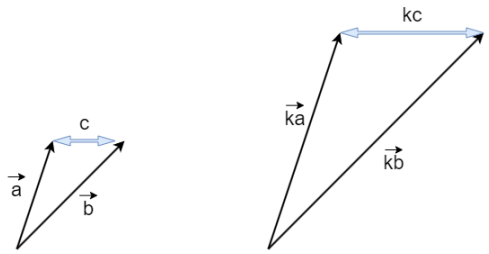

Figure 2: Euclidean Distance.

Minkowski Metrics (e.g., Euclidean (Fig. 2), Manhattan, Chebyshev) scale distances linearly when vectors are multiplied by the same constant. For vectors $\mathbf{a}$ and $\mathbf{b}$, let $c = \|\mathbf{a}-\mathbf{b}\|$. If $\mathbf{a}' = k\mathbf{a}$ and $\mathbf{b}' = k\mathbf{b}$, then

$$c' = \|\mathbf{a}'-\mathbf{b}'\| = \|k(\mathbf{a}-\mathbf{b})\| = |k|\|\mathbf{a}-\mathbf{b}\| = |k|c. \tag{7}$$

Thus, the distance scales by the factor $|k|$.

Cosine Similarity measures the angle between vectors, making it scale-invariant. For $\mathbf{a}$ and $\mathbf{b}$ with

$$\text{similarity}(\mathbf{a},\mathbf{b}) = \frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}, \tag{8}$$

if $\mathbf{a}' = k\mathbf{a}$ and $\mathbf{b}' = k\mathbf{b}$, then

$$\begin{aligned}\text{similarity}(\mathbf{a}',\mathbf{b}') &= \frac{\mathbf{a}'\cdot\mathbf{b}'}{\|\mathbf{a}'\|\,\|\mathbf{b}'\|} = \\ &= \frac{k^2\,(\mathbf{a}\cdot\mathbf{b})}{|k|\,\|\mathbf{a}\|\;|k|\,\|\mathbf{b}\|} = \\ &= \frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{a}\|\,\|\mathbf{b}\|} = \\ &= \text{similarity}(\mathbf{a},\mathbf{b}).\end{aligned} \tag{9}$$

Hence, cosine similarity (and its distance) does not change when both vectors are scaled by the same constant.

Using the above data, we implemented the following mechanism:

1. At the input, we calculate the coefficient k as the maximum value among the input vectors

$$\vec{a} = [a_1, a_2, a_3, \ldots, a_n] \tag{10}$$

$$\vec{b} = [b_1, b_2, b_3, \ldots, b_n] \tag{11}$$

$$k = \max(a_1, b_1, a_2, b_2, a_3, b_3, \ldots, a_n, b_n) \tag{12}$$

2. Then we standardize the values in both vectors by dividing them by the coefficient k.

$$a' = \left[\frac{a_1}{k}, \frac{a_2}{k}, \frac{a_3}{k}, \ldots, \frac{a_n}{k}\right] \tag{13}$$

$$b' = \left[\frac{b_1}{k}, \frac{b_2}{k}, \frac{b_3}{k}, \ldots, \frac{b_n}{k}\right] \tag{14}$$

3. The model calculates the distance for standardized vectors, which is multiplied by the coefficient k before the output:

$$d(a,b) = k \cdot \mathrm{d}(a',b'). \tag{15}$$

In the case of cosine distance, the third step is omitted.

2.2. The Vanishing Gradient Problem

When using RNNs for sequence processing, we must deal with the vanishing gradient problem. This problem occurs when the gradient values become smaller and

smaller during backpropagation. This happens because of repeated multiplication of the gradient by weights of neurons with values less than 1. This causes their value to decrease exponentially. This problem significantly hinders the learning of the model for long-term dependencies.

One solution to the aforementioned problem is a special type of recurrent network - Long Short-Term Memory. LSTM solves the problem by introducing special memory units that can maintain information for a longer time.

2.2.1 LSTM Structure

The main LSTM unit consists of several key elements:

- Memory cell (cell state): Stores long-term information. Information is added or removed through gates.
- Gates: Control the flow of information to and from the memory cell. There are three main gates:
  - **Forget gate**: Decides what information from the memory cell should be forgotten. It is a sigmoid layer that takes the previous hidden state and the current input state and generates a value between 0 and 1, where 0 means "forget completely" and 1 means "keep fully".
  - **Input gate**: Controls which new information will be written to the memory cell. It consists of a sigmoid layer that decides which values to update and a tanh layer that creates new candidate values.
  - **Output gate**: Decides what information from the memory cell will be passed on as output. It is also a sigmoid layer that modifies the cell state through a tanh layer.

2.2.2 LSTM Operation

Each time step in LSTM works as described below.

Forget gate:

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \tag{16}$$

calculates which parts of the memory cell c_{t-1} should be forgotten.

Input gate:

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \tag{17}$$

$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \tag{18}$$

decides which new information g_t will be added to the cell state.

Cell state update:

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t, \tag{19}$$

is updated, combining old information c_{t-1} filtered by the forget gate and new candidate values g_t filtered by the input gate.

Output gate:

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}), \tag{20}$$

$$h_t = o_t \odot \tanh(c_t), \tag{21}$$

decides which parts of the updated cell state c_t will be passed on as output.

3. Our Approach

3.1. Linear Network

When constructing our neural network, we used ready-made modules from the PyTorch library. This network consists of `nn.Linear` layers and ReLU activation functions, which are created based on the given layer sizes - at the input, when creating the model, it therefore receives a list that sequentially describes the sizes of subsequent hidden layers. The input size of the network is defined as $2 \times$ input_dim, where input_dim is the dimension of the input vectors. The last layer of the network consists of a single neuron.

Strictly speaking, only the list of hidden layer sizes is a hyperparameter of this network. However, considering the topology of the network by providing an exact list would be very inconvenient, so in this work we mainly use the number of layers and the coefficient Q

$$Q = 1 - \frac{1}{n-1}\sum_{i=1}^{n-1}\left(\frac{L_{i+1}}{L_i}\right) \tag{22}$$

where n is the number of layers, L_i is the number of neurons in the i-th layer, L_{i+1} is the number of neurons in the $i+1$-th layer.

The Q coefficient is intended to measure how the sizes of the layers converge. In this work, we only consider networks with a constant width (then Q = 0), and those in which each subsequent layer is smaller.

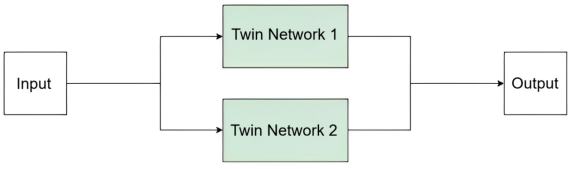

Figure 3: Siamese Network Schematic.

3.2. Recurrent Network

3.2.1 Model Hyperparameters

Similarly to linear networks, to construct our recurrent neural network (RNN), we used ready-made modules from the PyTorch library. We chose the Long-Short-Term Memory variant, available as `nn.LSTM`. The network architecture is defined by four hyperparameters:

- `hidden_dim_r`: Specifies the number of neurons in the recurrent module. It has the same value for the recurrent layer and for the fully connected input layers. This number affects the complexity of the model and its ability to learn the dependencies in the data.
- `hidden_dim_fc`: Specifies the number of neurons in the output layer. This number affects the complexity of the model and its ability to learn the dependencies in the data.
- `recurrent_layers_number`: Specifies the number of recurrent layers in the network. More layers allow for considering the context from further elements of the sequence.
- `fully_connected_layers_number`: Specifies the number of fully connected output layers. These layers transform the output of the recurrent layer to the desired output format.

An important feature of recurrent networks is that they do not require a change in input size. The input size is automatically adjusted to the size of the sequence. This means that the network can process sequences of different lengths without the need to modify its structure. The recurrent layer will perform a number of iterations equal to the length of the sequence.

3.2.2 Input Description

The data passed to the input of the recurrent model differs from the data used in linear and Siamese networks. Instead of a batch of pairs of vectors, we get a three-dimensional array with dimensions `batch_size` × `vector_size` × 2. This tensor contains a set of pairs of corresponding elements from two vectors. A single pair constitutes an element of the sequence that is processed during one pass of the recurrent network. This approach allows the network to better learn the dependencies between corresponding elements of the vectors, which is crucial in the case of Euclidean and cosine distances.

Our Siamese network model was built using ready-made modules from the library. This network consists of `nn.Linear` layers, which are created based on defined layer sizes and the number of hidden layers, resulting in a multi-layer structure.

The model takes three key parameters during initialization: `input_dim`, `hidden_dim`, and `output_dim`, which correspond to the input dimension, the hidden layer dimension, and the output dimension. Furthermore, the number of hidden layers is determined by the parameter `num_layers`. The input, in the form of two vectors, is passed separately to two twin networks, which return vectors of size `input_dim`, between which the Euclidean distance is then calculated.

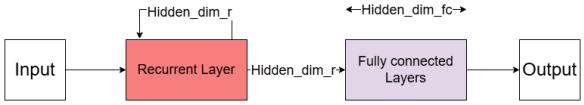

Figure 4: Reccurent network schematic

This is the most classic scheme of a Siamese network (Fig. 3). However, not wanting to impose an impossible dimensionality reduction task on the network, we decided to limit our research to cases where the parameter `output_dim` is equal to the parameter `input_dim`.

In such a situation, one could observe that the network would only have to pass the unchanged input vectors to the output to achieve 100% accuracy. In view of this fact, we did not consider this network in the context of learning the Euclidean distance.

Although recurrent networks allow processing sequences of different lengths without the need to adjust the network topology, we were unable to train a universal model for vectors of arbitrary length. This is due to the mechanism of recurrent networks - adding additional elements to the vector can disrupt the network's operation due to additional passes through the recurrent network. This problem does not occur when reducing the size of the vector, as one can then apply the *padding* technique, which consists of filling in missing elements with the value 0.

4. Experiments and Results

4.1. Hyperparameter Optimization of a Linear Model

This section is dedicated to the optimization of hyperparameters for a linear model. The goal is to find the best possible network topology. We will also examine the impact of the amount of data on the discrepancies between the results for training data and test data.

4.1.1 Topology in the Linear Model (Fig. 5)

Experiment 1 To determine the optimal topology of a linear model for 10-dimensional vectors, we first

generated different possible arrangements, sizes, and numbers of layers for the same computational complexity. For these settings, neural networks approximating the Euclidean distance were trained and tested multiple times for 10,000 epochs. Based on these results, we generated the Fig. 5, from which we can conclude that:

- two layers are the optimal number. Surprisingly, adding more layers does not seem beneficial.
- For the coefficient Q, the optimal value is 0.75. (An example of such a network is a network with the following layer sizes: 248, 186, 139, 104, 78, 59). It can also be observed that, in general, networks that converge more smoothly perform better.

4.1.2 Required Computational Complexity

In order to derive what level of computational complexity is required to achieve a satisfactory accuracy of the network for selected dimensionality, we selected all the networks achieving a loss function of 0.01 from the data generated in the experiment 4.1.1 then grouped them by their dimensionality and calculated the average of each group. From this transformed data, figure 6 was created.

- The shape of the function can be most accurately described as exponential-like.

4.1.3 Required Amount of Data

Experiment 2 We also conducted an experiment where the best-performing networks for a given dimensionality were trained for different sizes of training data in a range between 1000 and 1000000. In these kinds of experiments, we can observe a so-called "point of diminishing returns," i.e., a moment from which adding more vector pairs brings little improvement (see Fig. 7).

Based on the analysis of many graphs, such points were selected for many dimensions, and Fig. 8 was created.

- The relationship between dimensionality and the number of data points needed to reach the point of diminishing returns is linear.

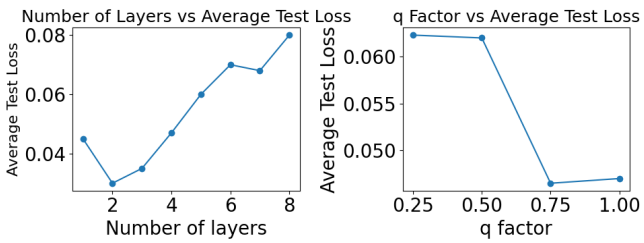

Figure 5: On the left: the dependence of the average loss function value on the number of layers, on the right: the dependence of the average loss function value on the Q factor (i.e on convergence)

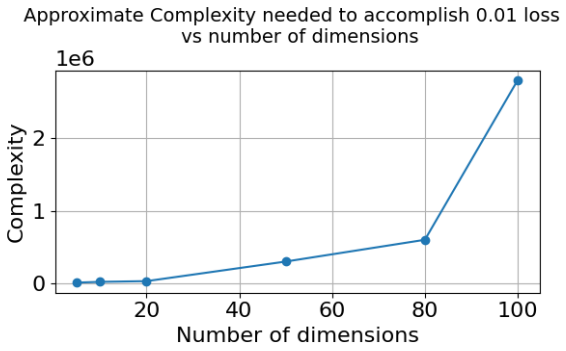

Figure 6: Average complexity of a network with a value of the loss function 0.01 for a given dimensionality

4.2. Hyperparameter Optimization of a Recurrent Model

This section is dedicated to the optimization of hyperparameters for a recurrent model. The goal is to find hyperparameter values that will allow for:

- Minimizing the amount of data needed to train the model: This is important when training a model on real, limited data.
- Minimizing the absolute error: Reducing the absolute error will lead to more accurate results in approximating the distance between vectors.
- Minimizing hyperparameters: Reducing the number of neurons in a layer and the layers themselves will allow faster calculations.

For the purposes of the research, some simplifications were made, thanks to which the research could be carried out more efficiently and with a wider range of data. The simplifications are as follows:

- It was assumed that the training ends for the *loss_tolerance* parameter equal to 0.05. A tolerance of error with such a value may be insufficient for many applications of approximating the distance between vectors, but this study is only intended to illustrate the trends appearing during training.
- It was assumed that if the number of epochs of data needed to train the model exceeds the value of *max_epochs*, then overfitting has occurred.

Both of these assumptions should not significantly affect the conclusions drawn from the research presented below.

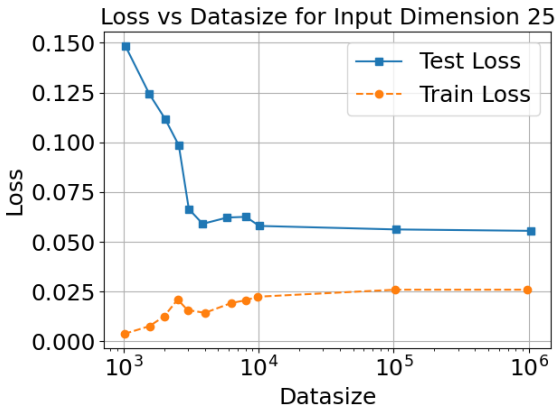

Figure 7: Function loss for train and test data versus the size of the dataset. Adding more data beyond the 5th datapoint does not lower the loss function, thus the 5th datapoint is "the point of diminishing returns".

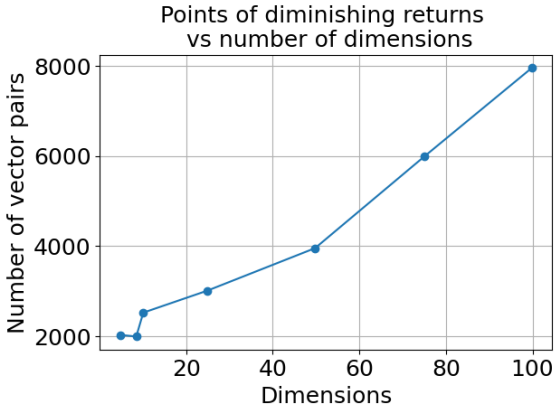

Figure 8: Dataset size necessary to reach the point of diminishing returns versus dimensionality

4.2.1 Number of Layers in a Recurrent Module

The first step is to examine the relationship between the number of layers in a recurrent module and the model's learning speed. These layers aim to find the relationship between two corresponding elements of a pair of vectors. When investigating the number of neurons in a recurrent module, it should be remembered that the computational complexity of the model grows very quickly with increasing this hyperparameter.

Experiment 3 (Subexperiment 3.1) The experiment consists of selecting several *input_dim* parameters and conducting training for them with a continuously increasing *recurrent_layers_number* parameter. This allows us to find the range of *recurrent_layers_number* values where the parameters with the best training results are located. For the experiment, we chose the following hyperparameter values:

- *input_dim* = [15, 30, 45, 75]
- *hidden_dim_r* = [24, 32, 64]
- *hidden_dim_fc* = *hidden_dim_r*
- *learning_rate* = 0.001
- *recurrent_layers_number* = [2, 4, 6, ..., 64]
- *fully_connected_layers_number* = 2
- *loss_tolerance* = 0.05
- *max_epochs_number* = 15000

Due to the occurring overfitting effect, the trainings were stopped at the moments when all the data contributing any value to the research were collected.

On the Figs. (9) we can observe several key phenomena:

- The input size significantly affects the amount of data needed to train the model - based on the collected data, the smaller the input size, the faster the learning and the higher the resistance to overfitting.
- The number of neurons in the recurrent and output layers significantly affects the training results - as can be seen, in most cases, smaller numbers of neurons have a positive impact on the model's results and delay the appearance of overfitting. The exception is the last graph, where *hidden_dim_r* and *hidden_dim_fc* with a value of 32 performed better.
- The range where we should look for the optimal number of layers in the recurrent module is at the beginning of the examined range.

Experiment 3 (Subexperiment 3.2) The experiment consists of training the model for different numbers of layers in the recurrent module and finding the most optimal one. This time, the range [1, 7] is examined, due to the results obtained in Experiment 3.1. First, we assume the following hyperparameter values:

- *input_dim* = [5, 10, 15, 20, 25, 30, 40, 50, 75, 100, 125, 150]
- *hidden_dim_r* = [32, 64]
- *hidden_dim_fc* = *hidden_dim_r*
- *learning_rate* = 0.001
- *recurrent_layers* = [1, 2, 3, ..., 7]
- *loss_tolerance* = 0.05
- *max_epochs_number* = 20000

Models with the above hyperparameters were trained multiple times, the number of epochs was divided by the minimum value in the corresponding set and averaged. The results also include those trials in which the model's error function did not converge to the specified value within 20000 epochs of data. However, they should not significantly affect the conclusions drawn from this study.

The results of the study are presented in Fig. 10. From the graph, we can read that the model learned best

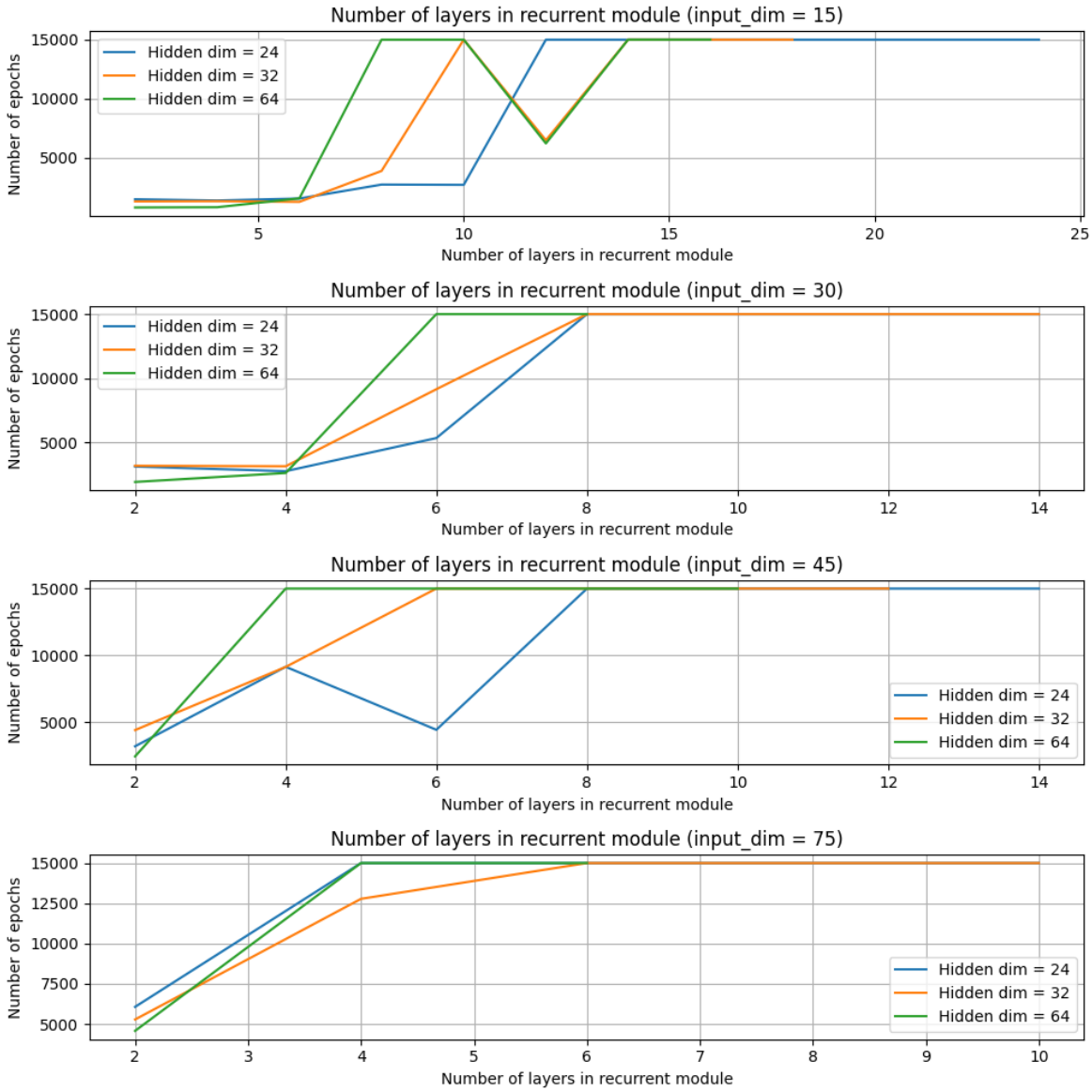

Figure 9: Number of layers in the recurrent module Experiment 3.1

when there were two layers of neurons in the recurrent module. A sudden drop is also visible at the point corresponding to five layers with a dimension of 32 neurons, but it does not occur for layers with a dimension of 64 neurons, so we reject it from potential candidates.

4.2.2 Number of Fully Connected (Output) Layers

Another key hyperparameter in the recurrent network model is the fully connected output layers. Their task is to transform the result obtained from the recurrent module into the approximated distance.

Experiment 4 In this experiment, we train the model multiple times, for continuously increasing values of the parameter *fully_connected_layers_number*. The goal is to observe the relationship between the amount of data needed for learning and the number of layers. The trainings were called for the following hyperparameters:

- *input_dim* = [15, 30, 45, 75, 100]
- *hidden_dim_r* = [32, 64]
- *hidden_dim_fc* = *hidden_dim_r*
- *learning_rate* = 0.001
- *recurrent_layers* = [2, 4, 6, . . . , 24]
- *loss_tolerance* = 0.05
- *max_epochs_number* = 15000

Models for the given hyperparameters were trained multiple times and their results averaged. From the results, we can draw the following conclusions:

- More than 10 layers do not significantly improve training speed.
- A larger number of neurons delays overfitting.

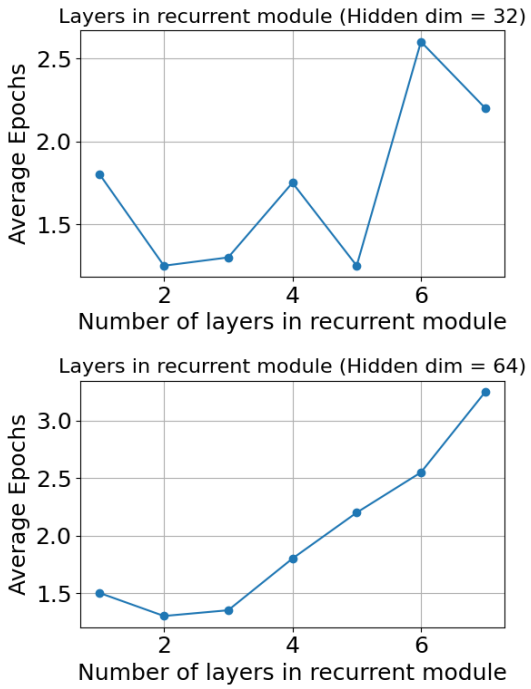

Figure 10: Number of layers in the recurrent module Experiment 3.2

- If we care about high model efficiency after training and have enough data, a number of layers in the range [2, 4] will handle this task well. However, if our dataset is limited and computation time after training does not matter, we can invest in a larger number of layers.

4.2.3 Number of neurons in the recurrent module

The number of neurons in the recurrent module *nn.LSTM* is responsible for the better ability of the model to find dependencies between subsequent data elements. This value is crucial from the perspective of model efficiency; the time needed for calculations grows very quickly with an increase in this hyperparameter.

Experiment 5 The experiment consists of training the model multiple times for changing values of *hidden_dim_r* and checking the dependence of the value of this parameter on the amount of data needed to train the model. The model is trained in this way for different lengths of vectors. The training parameters are as follows:

- *input_dim* = [15, 50, 75]
- *hidden_dim_r* = [16, 32, 48, ..., 256]
- *hidden_dim_fc* = 64
- *learning_rate* = 0.001
- *recurrent_layers* = [2, 4, 6, ..., 24]
- *loss_tolerance* = 0.05
- *max_epochs_number* = 15000

Analyzing the Fig. 12, we can draw the following conclusions:

- The overfitting effect becomes noticeable at a lower number of epochs for larger input sizes. Specifically, for input_dim = 75, the training converges quickly but shows signs of overfitting after approximately 150 epochs, whereas for input_dim = 50 overfitting is present at around 200 epochs. For input_dim = 15, the number of required epochs remains more stable.
- The number of neurons in the recurrent module in the range [32, 64] will handle the task well and be computationally efficient.

4.2.4 Number of neurons in fully connected layers in the output module

The number of neurons in a single layer of the output module is responsible for a more precise transformation of the output of the recurrent module to the expected result. The first layer has an input size of *hidden_dim_r* and an output size of *hidden_dim_fc*, and the last one has an input size of *hidden_dim_fc* and an output size of 1. The rest of the intermediate layers have both input and output equal to *hidden_dim_fc*.

Experiment 6 The experiment consists of training the model multiple times for a continuously increasing parameter, this time *hidden_dim_fc*, and observing changes in the amount of data needed for training. The training parameters are as follows:

- *input_dim* = [15, 50, 75]
- *hidden_dim_r* = 64
- *hidden_dim_fc* = [16, 32, 48, ..., 256, 288, 320, 352, ..., 512, 578, 642, ..., 1024, 1152, 1280, ..., 2096]
- *learning_rate* = 0.001
- *recurrent_layers* = [2, 4, 6, ..., 24]
- *loss_tolerance* = 0.05
- *max_epochs_number* = 15000

As can be seen in Fig. 11, a properly selected number of neurons in the output module can improve the model's results several times. For all three input sizes, the shape of the graph seems to be similar. The optimal values of the parameter *hidden_dim_fc* seem to be in the range [250, 750]. For larger values, a slight performance deterioration can be observed.

4.2.5 Amount of data needed for training

Experiment 7 The experiment consists of training the neural network multiple times for a continuously increasing input size. With each epoch, new data is generated, as in previous experiments, which allows us to capture the worst case of data demand for a specific input size. The

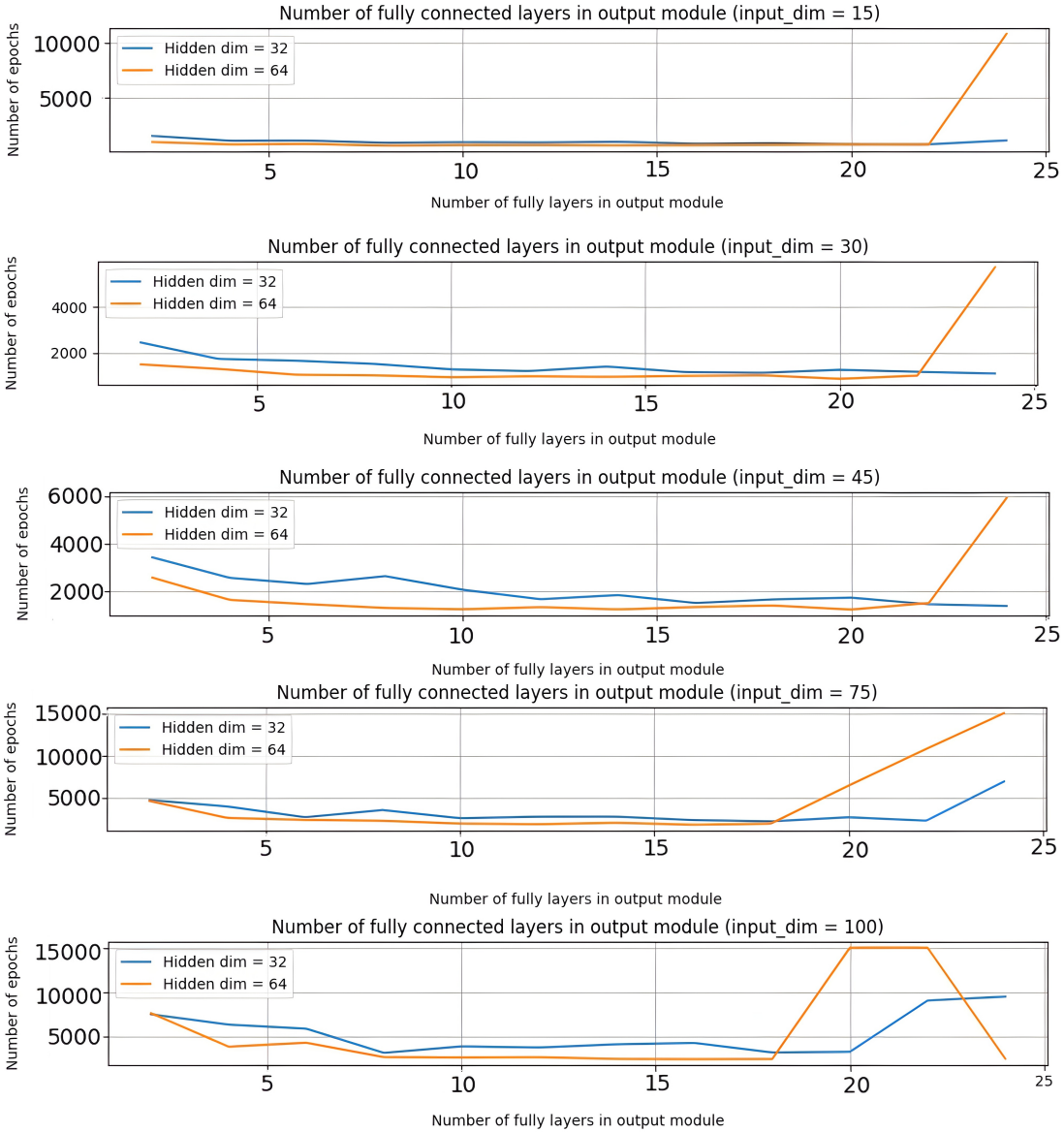

Figure 11: Number of fully connected (output) layers Experiment 4

network will be trained until the loss function value is less than 0.5.

The average growth rate (where the growth rate is defined as $\frac{\Delta V}{\Delta D}$, where ΔV is the change in the number of vector pairs needed and ΔD is the change in dimensionality) is equal to 3031 $\frac{vector\_pairs}{dimension}$. However, toward the right end (higher input dimensions), the curve steepens significantly, indicating an exponential or superlinear growth trend. From Fig. 14, we can see that the demand for data grows quite rapidly with the length of the vectors, even though the tolerance that we have adopted is not large. It is worth noting that we can reduce this demand by first generating our dataset and then training the network on its permutations.

4.2.6 Conclusions

From the conducted experiments, it follows that the best model for approximating the Euclidean distance is a model with parameters close to those given below:

- *recurrent_layers_number* = 2
- *fully_connected_layers_number* = 3
- *hidden_dim_r* = [32, 64]
- *hidden_dim_fc* = [256, 750]

It is possible that for increasing input size, these parameters will have to be adjusted, but due to the rapidly increasing computational complexity of the recurrent module, relative to the input size, the recurrent model may not be the best choice for very large vector sizes.

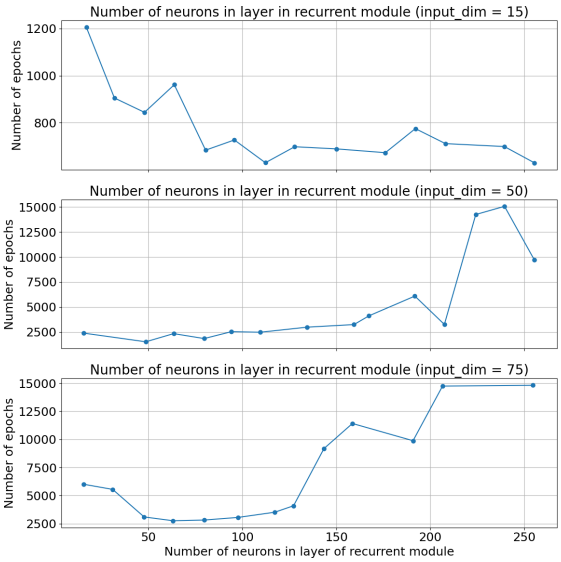

Figure 12: Number of neurons in the recurrent module Experiment 5

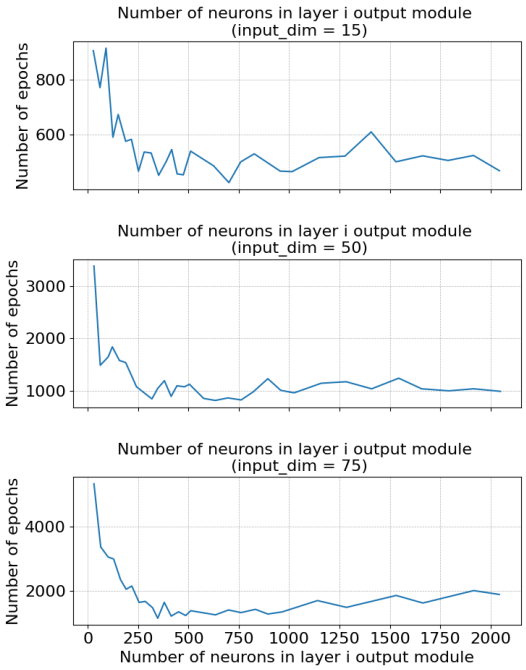

Figure 13: Number of epoches in the Experiment 6.

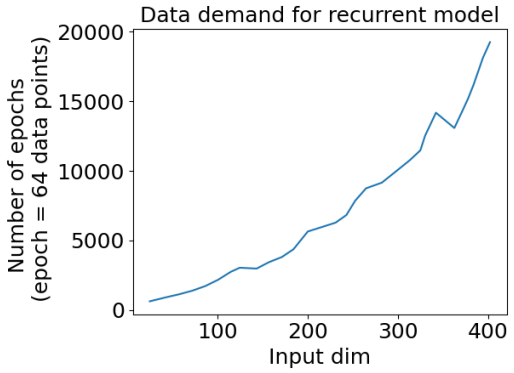

Figure 14: Data demand for training the recurrent network.

4.3. Data-Based Distance Metrics

In this work, we also wanted to investigate the so-called "Sample-Based Distance Metrics" where distances are defined a priori by humans; however, it is difficult to find publicly available datasets that we would need, and gathering such data exceeded our organizational and logistical capabilities. Nevertheless, to take some initial steps in this area for future research, we tried to simulate this type of situation using algorithms that create distance metrics based on data. For this purpose, we used the metric-learn library [8] and the algorithms contained therein. The research was conducted on many algorithms (LMNN, RCA, NCA, etc.), but in each case, similar dependencies could be observed, therefore, we present only the results of using the LMNN algorithm.

Experiment 8 (Subexperiment 8.1) The experiment was conducted by running the algorithm on the Iris dataset, calculating all distances between pairs of instances in the Iris set using this metric, and then using part of these as a training set for the selected neural network. In this way, a Siamese network (Fig. 15), a linear network (Fig. 16), and a recurrent network (Fig. 17) were trained for 200 epochs.

Experiment 8 (Subexperiment 8.2) Additionally, we then performed classification using the KNN algorithm and trained neural networks, resulting in accuracies of 53% for Siamese, 49% for recurrent, and 37% for linear networks. These results lead to an interesting conclusion that Siamese networks, despite achieving lower accuracy in distance measures, achieve better results in classification.

5. Discussion

5.1. Comparison of distance approximation results

The results of the experiments showed that recurrent networks generally achieve better results in approximating the distances between the vectors compared to linear networks, and they work better than Siamese networks. However, when using networks in KNN classifications, the Siamese architecture outperforms both linear and recurrent networks.

5.2. Topologies

5.2.1 Topology of reccurent model

In the research on the number of recurrent layers, it was observed that increasing the number of LSTM layers up to a certain point improves the quality of the approximation. The optimal solution for most of the tested cases was an architecture with two recurrent layers. A larger number of layers led to problems related to overfitting the model, which slowed the learning process.

Similar effects were observed when analyzing the number of neurons in the recurrent module. The larger the number of neurons, the longer the model learned, but at the same time, the effect of overfitting was delayed. The optimal number of neurons in the tested configurations ranged from 32 to 64, which allowed a good balance between learning time and model accuracy. It is possible that the number of neurons will need to be increased for extremely large input sizes to improve the ability of the recurrent module to remember long dependencies.

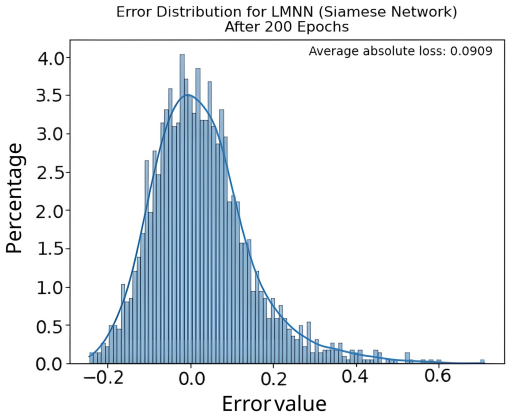

Figure 15: Error distribution for the Siamese network trained for 200 epochs for the LMNN algorithm.

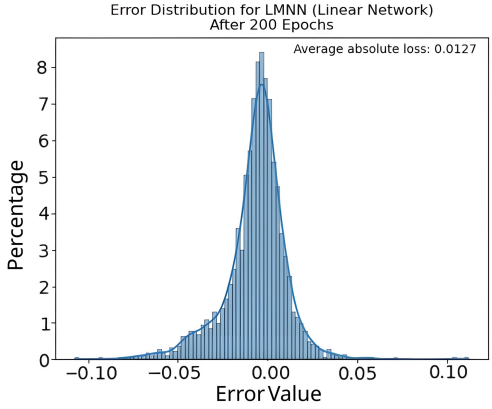

Figure 16: Error distribution for the linear network (feed forward) trained for 200 epochs for the LMNN algorithm.

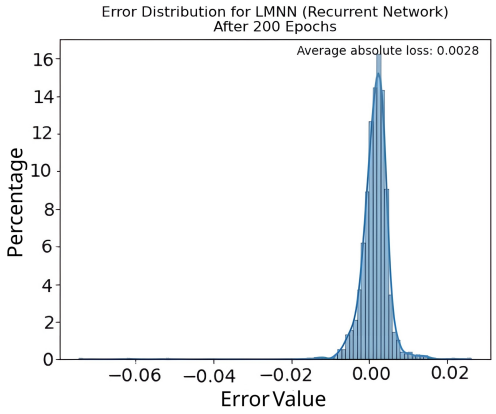

Figure 17: Error distribution for the recurrent network trained for 200 epochs for the LMNN algorithm.

5.2.2 Topology of linear model

Linear models appear to function best when they include two layers, adding more of them did not appear to improve accuracy of the network; this may be due to overfitting, or the fact that more complex networks may need more time (i.e., more epochs) to reach the same level, while in our case each model was trained for the same number of epochs.

Networks that converge smoothly (Q factor of 0.75 or higher) consistently outperform those that converge faster. This could be caused by the last layers of models that converged fast being too small to contain all the necessary information.

The necessary complexity to accomplish given accuracy appears to be increasing exponentially with the increase in dimensionality.

5.3. Influence of Data Quantity on Model Efficiency

Experiments have also confirmed that the amount of input data has a significant impact on the quality of the approximation. Models with fewer data dimensions achieved satisfactory results faster and were more resistant to overfitting. However, for data with a higher number of dimensions, a significantly larger number of training examples was required for the model to achieve an acceptable level of generalization.

Graphs showing the relationship between the amount of data and model quality (loss function) indicate a linear relationship between data dimensionality and the number of needed samples for feed-forward architecture and an exponential relationship for recurrent architecture.

This discrepancy suggests that the feedforward architecture is better suited for higher-dimensional data.

5.4. Classification Based on Distance Approximation

An interesting result of the experiments was the analysis of classification results based on approximated distances, especially when using Siamese neural networks. Although these networks achieved slightly worse results in approximating the distance itself, they proved to be more effective in the classification task. Siamese networks, thanks to their architecture designed to compare vector pairs, achieved the highest classification accuracy of 53%, outperforming both recurrent networks (49%) and linear networks (37%).

Acknowledgements

We would like to express our gratitude to Professor Julian Szymański for his guidance and continuous support throughout the duration of this research.

References

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[2] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng, "Deep speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[3] S. Wolfram, "What is chatgpt doing … and why does it work?," 2023. Accessed: 2024-06-20.

[4] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, 2020.

[5] R. Coulom, *Reinforcement Learning Using Neural Networks, with Applications to Motor Control*. PhD thesis, Institut National Polytechnique de Grenoble - INPG, Jun 2002.

[6] J. Wan, D. Wang, S. Hoi, P. Wu, J. Zhang, and J. Li, "Deep learning for content-based image retrieval: A comprehensive study," 2014.

[7] S. Amidi, "Cheatsheet - recurrent neural networks." Stanford University CS 230 Course Website, December 2019. Accessed: June 21, 2024.

[8] W. de Vazelhes, C. Carey, Y. Tang, N. Vauquier, and A. Bellet, "metric-learn: Metric Learning Algorithms in Python," *Journal of Machine Learning Research*, vol. 21, no. 138, pp. 1–6, 2020.