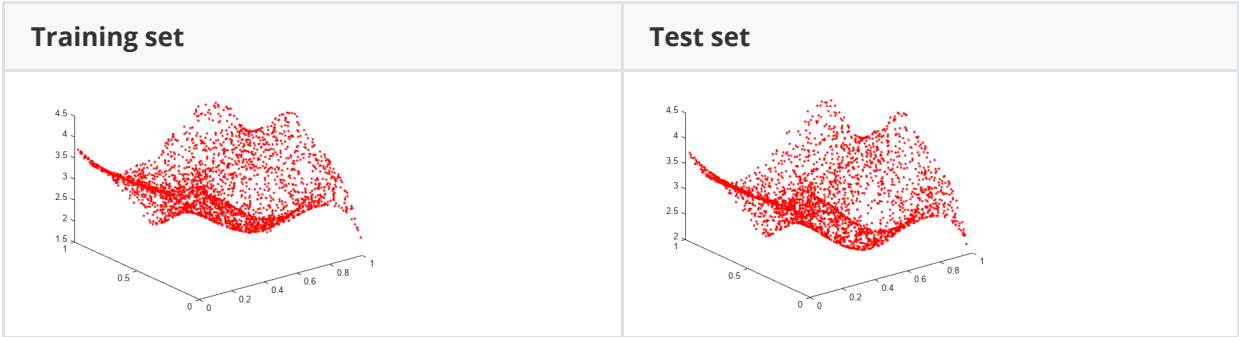


Report 1

Backpropagation in feedforward multi-layer networks

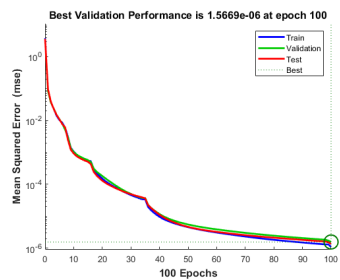


Using the datasets that can be seen above, we will train and test 4 networks with 4 different training algorithms:

- Levenberg-Marquardt algorithm (trainlm)
- BFGS quasi Newton algorithm (trainbfg)
- Gradient descent (traindg)
- Gradient descent with momentum and adaptive learning rate (traingdx)

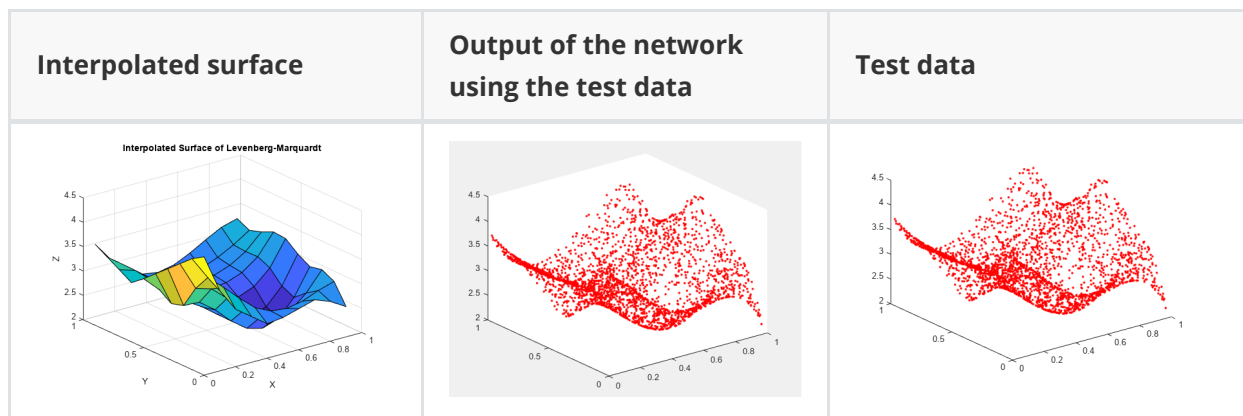
For each of these networks, we will use a hiddenlayer of size 2 with 20 and 10 neurons respectively. The networks are trained for 100 epochs. This seemed an adequate amount of epochs and neurons because the model complexity is relatively low and we would risk overfitting. Higher amounts of epochs seemed to pass a point of diminishing returns after 100.

Levenberg-Marquardt



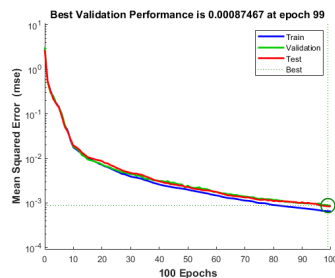
The image above shows us the MSE over the amount of epochs. The algorithm quickly converges to a satisfactory error of 1.5569×10^{-6} .

Interpolated surface	Output of the network using the test data	Test data
----------------------	---	-----------

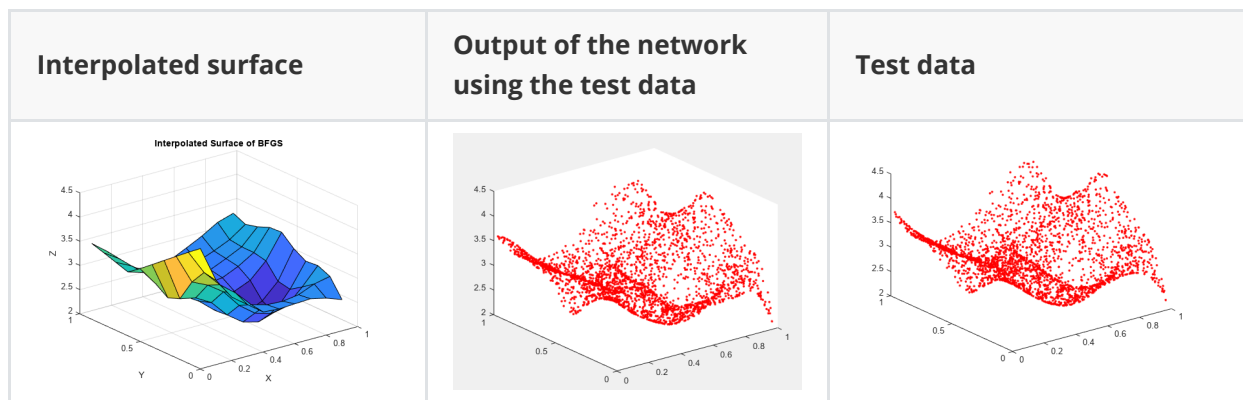


The network training appears successful as the output closely matches the test data. However, a drawback of the Levenberg-Marquardt algorithm is its reliance on the Hessian matrix, which can introduce computational complexity and memory requirements. Fortunately, this is not an issue for our relatively small dataset.

BFGS quasi Newton

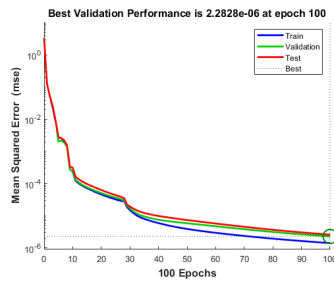


Here the algorithm converges to an error of 0.00087467.

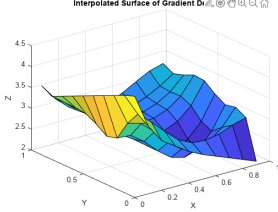
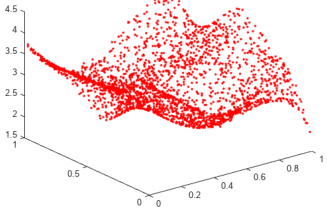
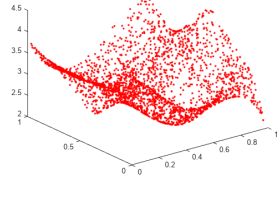


Again, this looks like a success because the output closely resembles the test data although if we take a closer look we can spot some differences. This could allow us to conclude that BFGS may not be the most optimal algorithm for the problem.

Gradient descent

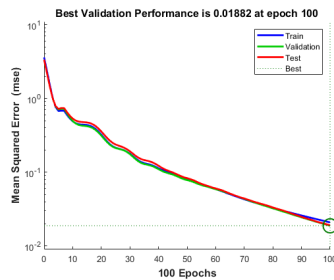


The algorithm converges to an error of $2.2828 * 10^{-6}$.

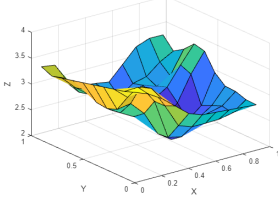
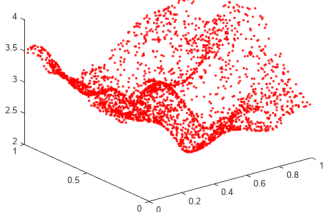
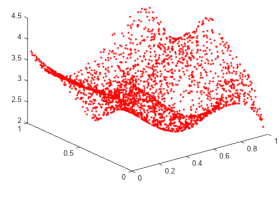
Interpolated surface	Output of the network using the test data	Test data
		

Gradient descent also does very well, a small error and a good prediction. Since our model is relatively simple we also don't have the problem of the vanishing gradient.

Gradient descent with momentum and adaptive learning rate



The error is relatively large at 0.01882

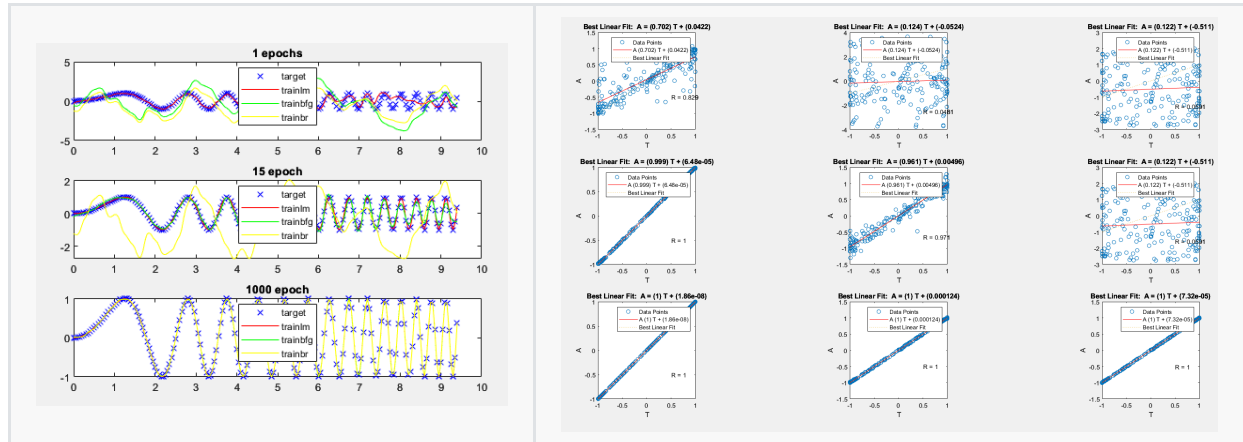
Interpolated surface	Output of the network using the test data	Test data
		

Of all the algorithms used, this is by far the worst. The reason that it is so much worse than its counterpart Gradient Descent might be because of the fact that it has far more hyperparameters to tune and that it is overshooting the data because of the momentum used.

Summary

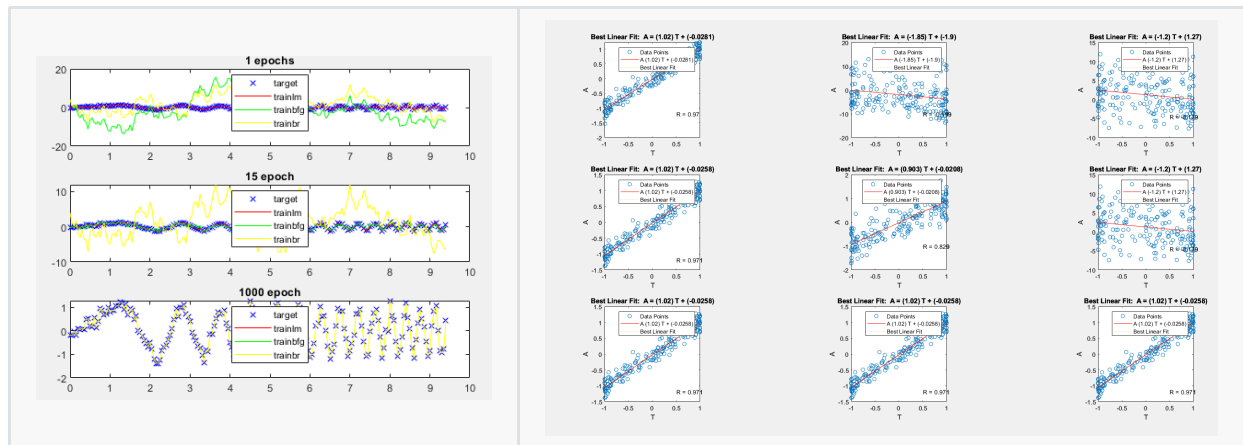
Based on the analysis, it can be concluded that the Levenberg-Marquardt (LM) algorithm demonstrates superior performance, closely followed by Gradient Descent (GD). However, it should be noted that in some instances, GD has shown better results compared to LM. It is worth mentioning that the performance of all algorithms may improve on larger datasets, but eventually, GD and GD with momentum and adaptive learning rate will end up outperforming LM and BFGS because these two do not perform well on larger datasets. Therefore, it can be reasonably extrapolated that Gradient Descent is the most suitable algorithm for this specific problem.

Regression with Bayesian regularization



(1) The figure on the right gives us the data in column order for: LM, BFGS and BR.

Initially, Bayesian regularization shows lower performance compared to other algorithms with a small number of epochs. However, as the number of epochs increases, Bayesian regularization gradually improves its performance and becomes comparable to Levenberg-Marquardt. This behavior can be attributed to the incorporation of prior knowledge about the parameter distribution in Bayesian regularization. In the early stages of training, when the network has limited information about the underlying patterns in the data, the prior knowledge may not perfectly align with the true parameter distribution, leading to a slower start.



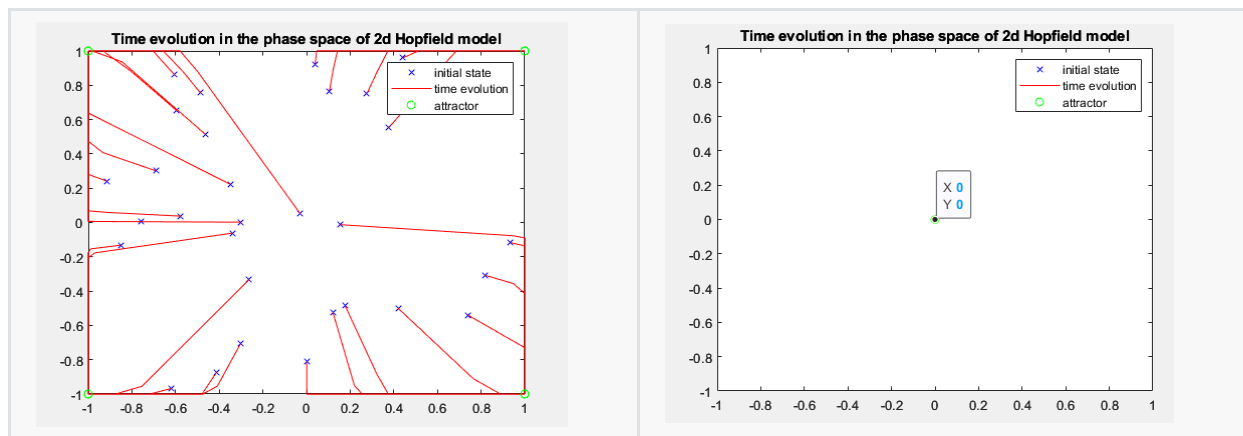
When we then add noisy data, it clearly negatively impacts the performance of all algorithms, including Bayesian regularization. When we further increase the number of neurons to 200 or even 500, there are no significant drastic changes observed. Despite its initially poor performance, Bayesian regularization exhibits a remarkable improvement over time, highlighting its ability to adapt and learn from the data.

Report 2

2D Hopfield

Using the specified attractors $[1 \ 1; -1 \ -1; 1 \ -1]^T$, we constructed a Hopfield network. During the simulation, we observed that approximately 30 randomly entered data points converged within around 30 timesteps. However, an unintended attractor, $[-1 \ 1]$, emerged during this process. It is not uncommon for Hopfield networks to generate spurious states, which are patterns that were not explicitly included as attractor states. In the case of the additional attractor $[-1 \ 1]$, it exhibits a point of high symmetry. This means that it can be reached by flipping the signs of the third attractor. The symmetry property of the network allows it to converge towards this unintended attractor.

Considering the storage capacity of a Hopfield network, it depends on the number of neurons and the symmetry of the attractors. If we assume perfect recall, the maximum capacity can be estimated using the formula $p_{max} = \frac{N}{(4 \cdot \log(N))}$, where N represents the number of neurons. In our Hopfield network with 2 neurons, the capacity is calculated to be 1.660. This capacity is clearly not sufficient to store all 3 attractors, which also contributes to the generation of spurious states.

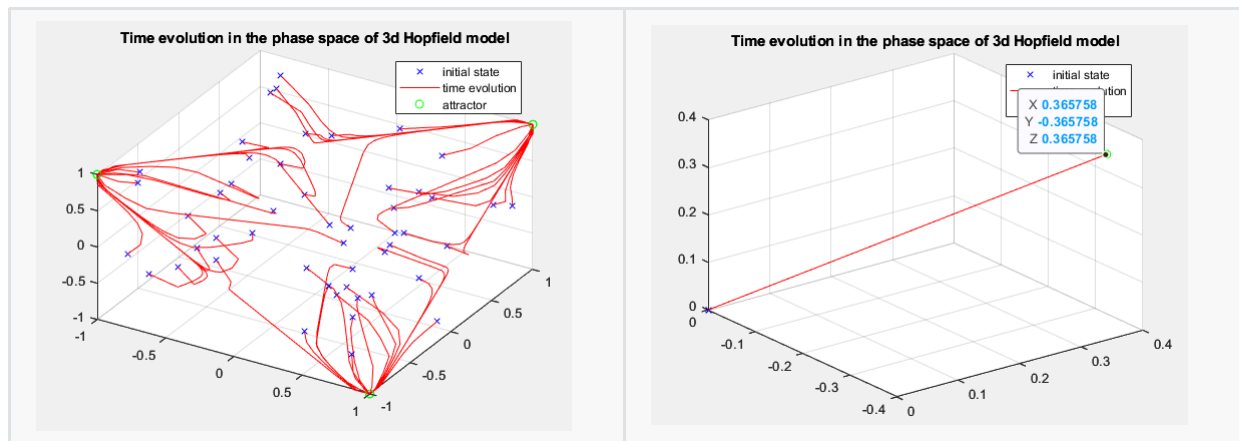


The attractors seem to be quite stable because they are symmetric and we see no oscillations after reaching convergence. One of the interesting initial values in this case is the point $[0 \ 0]$. The point will not converge in one of the given attractors and instead create a spurious state in the same place as can be seen above on the right. This is probably because the point $[0 \ 0]$ does not closely resemble any of the specified attractors and thus it will not converge to any of them and seek out a local minima which might be exactly in the point $[0 \ 0]$.

3D Hopfield

We proceed to construct a 3D Hopfield network with the attractors $[1 \ 1 \ 1; -1 \ -1 \ 1; 1 \ -1 \ -1]^T$. As we enter 50 data points into the network, we observe convergence within approximately 45 timesteps. However, considering the storage capacity of the network using the previously mentioned formula, we find a capacity of 1.572. This indicates that the network does not have sufficient capacity

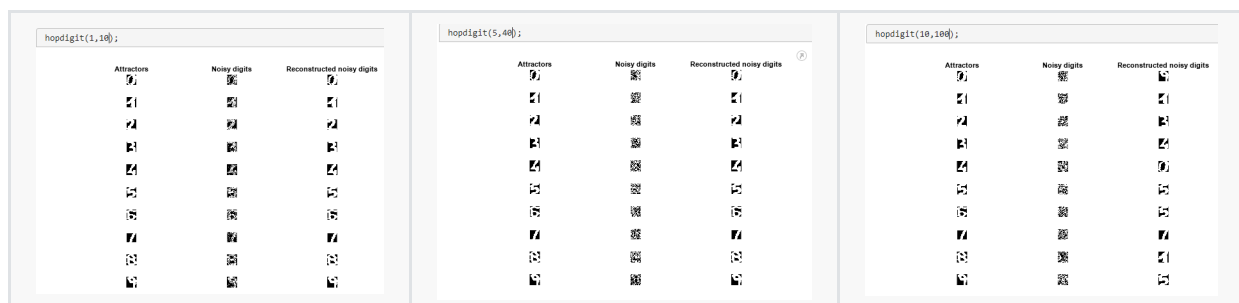
to store all the attractors without introducing spurious states. Interestingly, contrary to our expectations, no additional attractors are generated despite the indications that suggest otherwise. The existing attractors demonstrate stability and do not exhibit any oscillations. Furthermore, they possess a high degree of symmetry.



There is some other curious behaviour. When we enter the point $[0 \ 0 \ 0]$, a seemingly random attractors is created at the point $[0.365758 \ -0.365758 \ 0.365758]$, as can be seen above on the right. It is plausible that the energy landscape of the network contains a local minimum that was unintentionally created during the network's construction. This local minimum may be influencing the convergence of our extra point towards the unintended attractor.

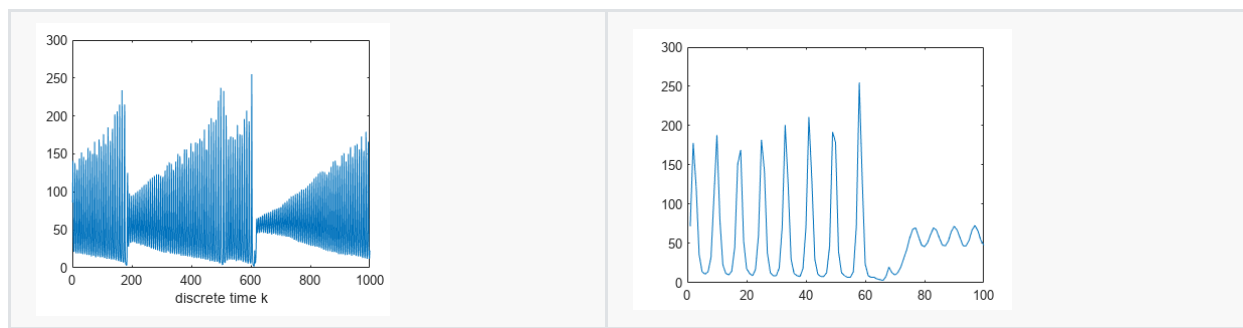
Hopdigit

Hopfield networks excel at recognizing numbers when the data is noise-free. However, as we introduce noise into the input, the network's accuracy in digit classification starts to decline. To mitigate this issue, we can increase the number of iterations during the network's learning process. This helps improve the network's ability to handle moderately noisy data. Nevertheless, there is a limit to the effectiveness of iteration-based improvement. As the level of noise reaches a certain threshold, the Hopfield network struggles to classify the digits correctly. This outcome is not surprising, considering that digits with a noise level of 10 lack distinct features, as illustrated below.



MLP with sequential data

Upon examining the training data, we can observe patterns, albeit with some inconsistencies, particularly in the occurrence of crashes. This irregularity may pose challenges for our MLP model in accurately approximating the data. Furthermore, a similar pattern is evident in the test data, indicating that the MLP will need to predict crashes that occur infrequently, which could result in less accurate predictions. The limited amount of data available is likely a significant factor influencing the model's accuracy.



I trained the MLP using the BFGS training algorithm for 250 epochs. I explored other algorithms that are recommended for sequential data, but they yielded significantly poorer results. As shown below, the performance of these alternative algorithms varied widely, with some coming close to the desired outcome while others falling far off the mark. Notably, when comparing the MLP with 30 neurons to its counterpart with 20 neurons, we observed a significant drop in performance. This could potentially be attributed to overfitting, where the model becomes too specialized to the training data.

In our case, the term "lag" refers to how far back we look to compare our current datapoint. Increasing the lag has a positive impact on performance, as demonstrated by the results. When we reached a lag of 50, the approximation came remarkably close to the desired outcome, indicating the effectiveness of considering a larger historical context in the sequence.

	20 neurons	30 neurons
10 Lag		
20 Lag		
50 Lag		

LSTM

As expected, the LSTM outperformed the MLP in handling sequential data. LSTM networks are specifically tailored for sequential tasks, and their superiority is evident in the results obtained. In this case, we trained the network using 500 neurons and 250 epochs. By increasing the lag, we observed an improvement in the root mean square error (RMSE), which is intuitive. As the lag increases, the

LSTM can capture more contextual information and better model the sequential patterns, leading to enhanced performance. During experimentation, adjusting the learning rate and the solver did not yield better results. Therefore, we opted to stick with the default values of 0.0005 for the learning rate and the Adam solver, as they provided the most favorable outcome.



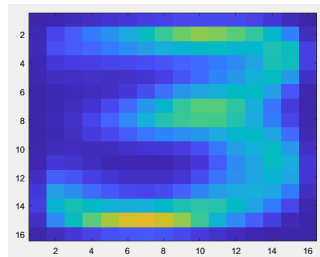
The superiority of the LSTM over the MLP is evident from the results, making the choice between the two quite clear. Not only did the LSTM outperform the MLP, but it also exhibited faster training times. This combination of superior performance and efficient training makes working with the LSTM a much more favourable experience.

Considering all the evidence and observations, it's clear that the LSTM is the preferred choice. Its effectiveness and efficiency align with our expectations and make it the recommended option.

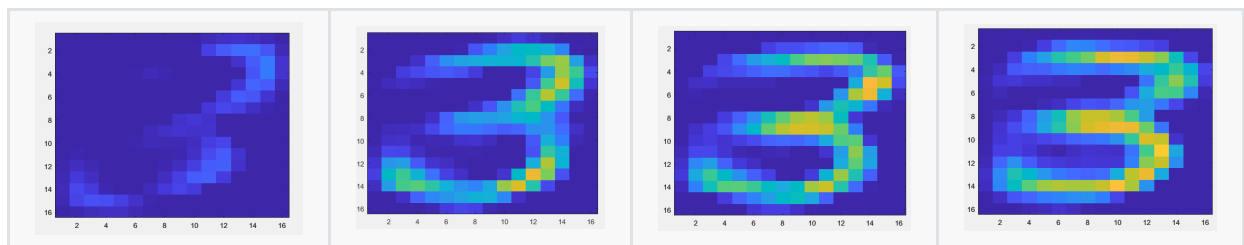
Report 3

PCA

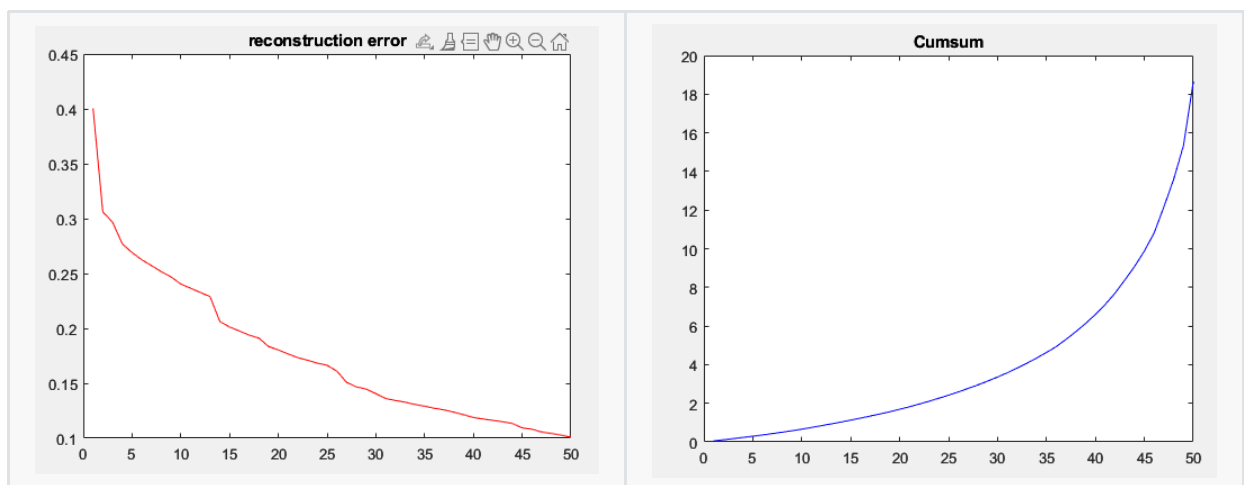
PCA aims to find a lower-dimensional representation of the data by capturing the maximum amount of variance in the original dataset. Here we use PCA on the threes dataset to see how close we can get to reconstructing it with as small as error as possible. Below you can see what the mean three of the dataset looks like



Below you can see what the threes dataset looks like when projected on 1, 2, 3 and 4 principal components. Because we take the eigenvectors with the largest magnitude as principal component we can still notice a distinct 3 even with only 1 principal component. We see some major improvements with each extra principal component which is only logical as this increases the quality of the reconstruction.



When observing the image below and on the left, we can clearly observe a rapid decrease in the reconstruction error. By incrementally adding principal components, starting from 1 to 50, we witness significant improvements in the error. As we include the eigenvectors with the largest magnitudes, the error drops substantially. In theory, if we were to compute the reconstruction error between the original data and the reconstructed data using all 256 principal components, the error should ideally be zero. However, in practical scenarios, the reconstruction error may not reach absolute zero but rather converge to a very small value, approaching zero. In our specific case, the obtained error was $6.0287 * 10^{-16}$, which aligns with our initial expectations.



The insight that the reconstruction error is proportional to the eigenvalue of the unused principal components plays a crucial role in PCA for dimensionality reduction. By selecting principal components with substantial eigenvalues, we can reduce the dimensionality of the data while minimizing the loss of information. The cumulative sum of the eigenvalues provides a useful measure to determine the number of principal components to retain. Our goal is to retain enough components to capture a significant portion of the total variance while disregarding components with negligible impact on the data representation.

Stacked autoencoders

Stacked auto encoder class accuracy = 99.7200

Neural net with 1 hidden layer class accuracy= 95.8400

Neural net with 2 hidden layers class accuracy= 96.5800

Neural net with 3 hidden layers [100 50 50] class accuracy = 97.1400

Neural net with 3 hidden layers [100 50 25] class accuracy= 97.2200

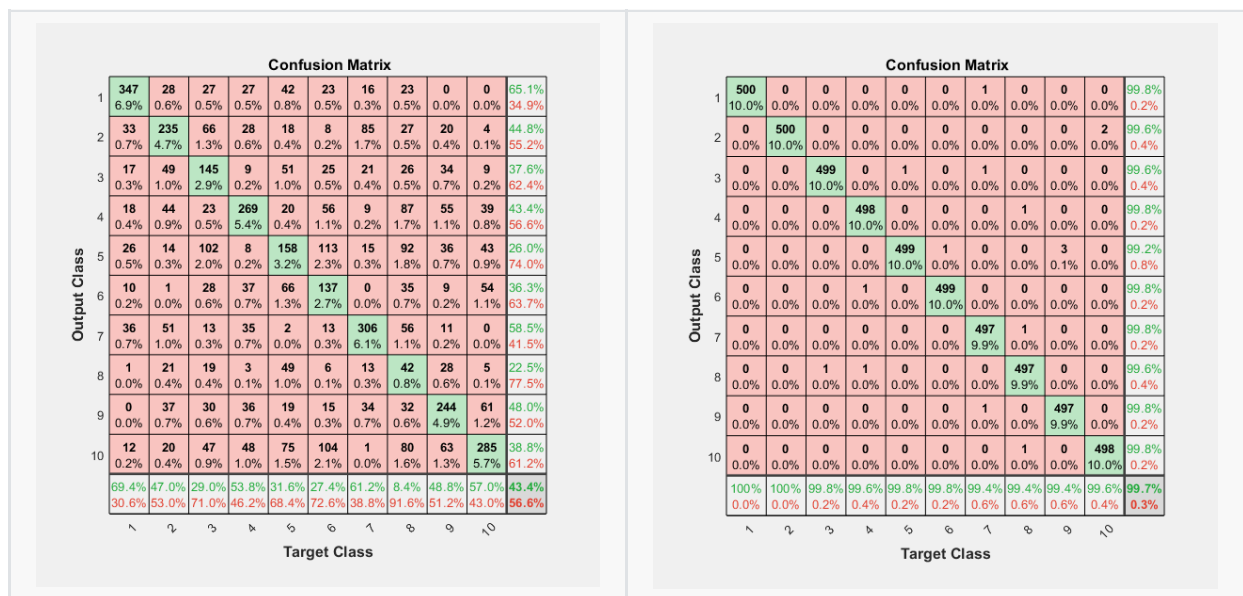
Neural net with 3 hidden layers [100 50 25] without validation class accuracy = 98.4400

After conducting experiments with different configurations such as varying epochs, neurons, and hidden layers, we have determined that our regular neural network achieves its best performance with three hidden layers consisting of 100, 50, and 25 neurons respectively. However, even with this optimal configuration, the performance of the regular neural network still falls short of the stacked autoencoder. It appears that the capabilities of our shallow neural network are limited compared to the stacked autoencoder. Interestingly, the number of epochs does not seem to have a significant impact on performance, as the desired gradient target value is reached relatively quickly even with a limit set higher than 100 epochs.

The process of finetuning the deep neural network, which includes the autoencoder layers and the softmax layer, proves to be beneficial for optimizing the overall model. During finetuning, the parameters of all the layers are adjusted together, enabling better alignment and coordination among them. This collaborative optimization allows the model to function more effectively as a cohesive unit, leading to improved performance.

Both the unsupervised training of the stacked autoencoder and the subsequent supervised finetuning are crucial for achieving optimal results. The unsupervised training initially captures deeper patterns and representations, while the finetuning phase refines and enhances these representations. Together, these stages contribute to the overall improvement in performance.

Below, we can observe the accuracy comparison between the stack without finetuning on the left and the accuracy with finetuning on the right. This highlights the significant impact of finetuning on the model's accuracy, demonstrating the importance of both unsupervised training and finetuning in achieving the best possible performance.



CNN

Weight representation

In the CNN we used, the second layer or convolution layer has a few weights, namely [96 11 11 3]. The "96" in this case means the amount of filters used. The 2 elevens are the dimensions of the filter so in this case 11 x 11 pixels. The 3 defines in what color space we are working. 3 is for RGB and 1 is for grayscale.

Input dimension

After layer 5 in our CNN the dimension will be 27×27 . We found this by using the following formula $(W - F + 2P) / S + 1$ where

- W is the dimension of the input
- F is the size of the filter
- P is the size of the padding
- S is the size of the stride

We apply the formula twice because it goes through a convolution layer and a max pooling layer. We could also just use the command `analyzeNetwork` but this would not give us an explanation as to why that is.

Input dimension of fully connected layer

The input dimension right before it enters the fully connected layer is 1×1 . This reduction in dimensionality occurs as the original 227×227 input undergoes a series of convolutional and pooling operations. By reaching a dimension of 1×1 , the input size for the fully connected layer becomes significantly smaller. This reduction is beneficial as it allows the fully connected layer to efficiently process and classify the data.

If we were to retain the original input size, the fully connected layer would require an enormous number of weights to balance, resulting in a computationally expensive and impractical model. This is one of the reasons why convolutional neural networks (CNNs) outperform fully connected networks in image classification tasks. The CNN architecture, with its convolutional and pooling layers, enables

effective feature extraction and dimensionality reduction, leading to improved performance and efficiency in image classification.

CNNdigit

Based on the provided code, we can observe that the model with an additional convolutional layer and ReLU activation performs significantly poorer compared to the model with a single convolutional layer. This drop in performance could be attributed to the increased complexity of the model, which can lead to overfitting. The model's performance can be visualized on the left.

Furthermore, when we replace the max pooling layer with an average pooling layer, we observe a slight improvement in performance. This could be attributed to the fact that average pooling retains more information compared to max pooling, which can result in better representation of the underlying patterns.

