Artificial Neural Networks and Deep Learning

Exercise session 1

Backpropagation in feedforward multi-layer networks

Function approximation

The target for approximation is the function $y = \sin(x^2)$ for x ranging from 0 to 3π, with an increment of 0.05. Three different backpropagation algorithms were utilized to train three neural networks, each comprising one hidden layer with 50 neurons. The algorithms used were gradient descent (traingd), Levenberg-Marquardt (trainlm), and BFGS quasi-Newton (trainbfg).

To compare the speed of convergence, three different epochs (1, 14, 985) were selected to observe the learning progress of each network. Additionally, clear comparisons among the three networks during the training process were provided. Since the primary goal is to approximate the target function as quickly as possible, all data points were used as the training data, and the concern about the "overfitting" problem was disregarded.

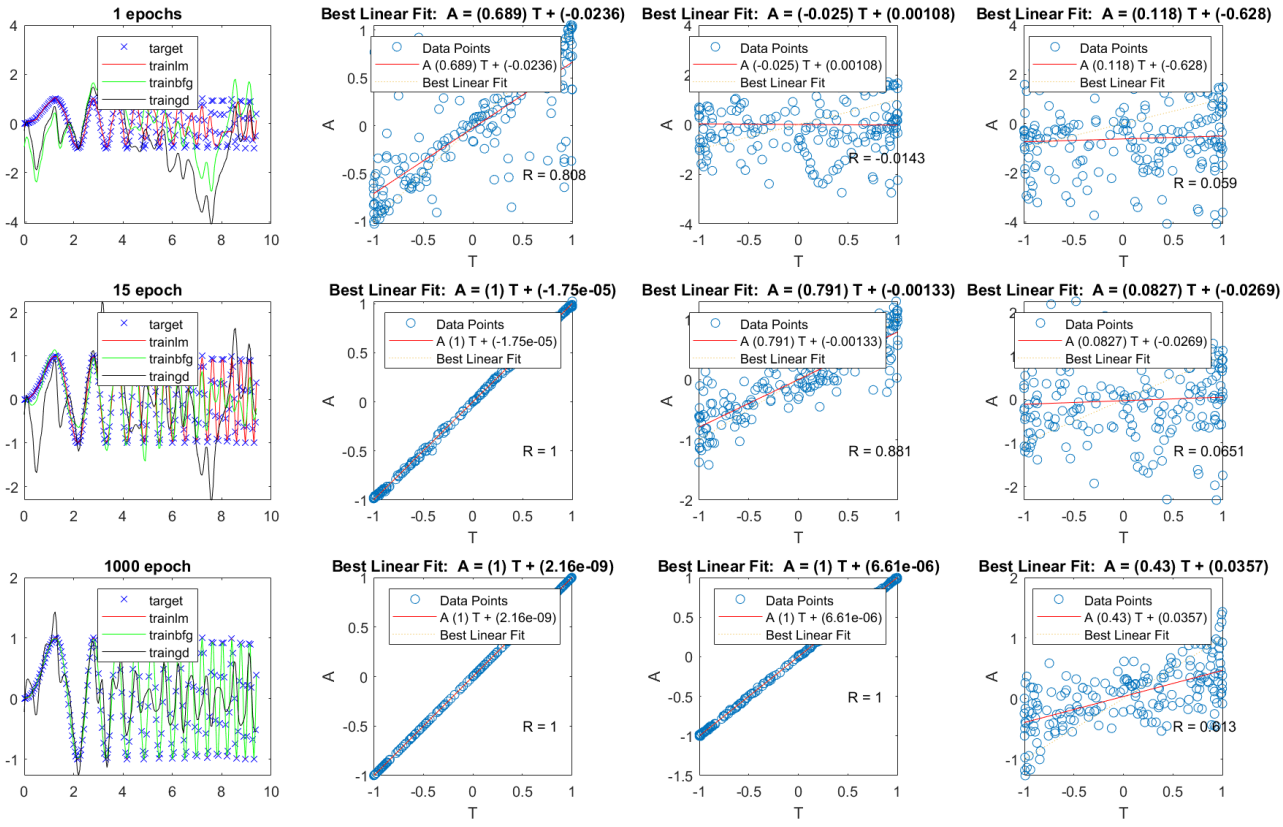


Figure 1. The left column: target function without noise overlapped with 3 trained network model. Right: Comparison of Correlation (R) between Target Function and Outputs of Trained Network Models in 3 check points during 1000 epochs training for 3 networks models. From left to right, they are: models based on Levenberg-Marquardt, models based on BFGS quasi-Newton and models based on gradient descent.

The Levenberg-Marquardt (LM) algorithm demonstrated exceptional performance on this dataset compared to the other two parameter refining algorithms. After a single training session, it achieved a correlation of 0.808 between the outputs and targets. Conversely, the network based on the quasi-Newton algorithm required 15 training sessions to achieve a similar level of performance. At this point, the network based on the LM algorithm closely matched the target function, yielding an impressive correlation coefficient of 1. On the other hand, the gradient descent algorithm performed poorly in comparison to the other two methods. Even after 1000 training sessions, it still failed to capture the underlying pattern in the data.

Learning from noisy data: generalization

The noise in the data was generated using the random function (randn) with a standard deviation of 0.2. It was then added to the target function ($y = \sin(x^2)$ for x ranging from 0 to 3π). Similar results were observed, with the Levenberg-Marquardt

algorithm quickly capturing the underlying data pattern even after a single training session. However, due to the presence of noise, there were no significant differences between the network based on LM after 15 and 1000 training sessions. The correlation value (R) remained around 0.96, indicating that the network did not overfit excessively to the noise. This suggests good generalization capabilities, but it should be further confirmed using testing sets.

The quasi-Newton algorithm also facilitated effective learning of the dataset, although it exhibited slightly slower convergence compared to LM. On the other hand, the gradient descent algorithm performed poorly once again in the presence of noise. It requires more data to improve its ability to learn the underlying data pattern.

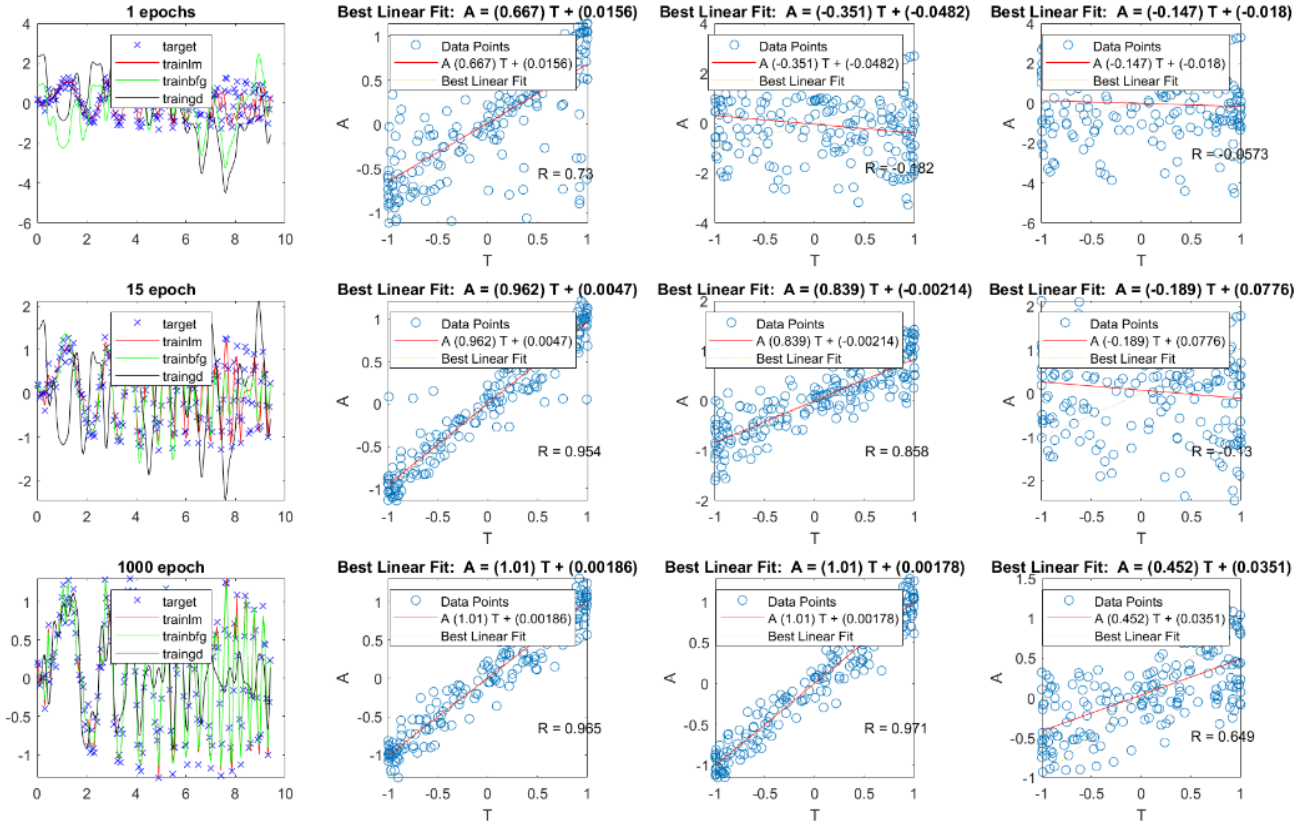


Figure 2. The left column: target function withnoise overlapped with 3 trained network model. Comparison of Correlation (R) between Target Function with Noise and Outputs of Trained Network Models for 3 check points during 1000 epochs training for 3 networks models. From left to right, they are: models based on Levenberg-Marquardt, models based on BFGS quasi-Newton and models based on gradient descent.

Overall, all models performed slightly worse when noise was introduced. However, both LM and quasi-Newton algorithms outperformed the gradient descent algorithm in terms of convergence speed and accuracy. This can be attributed to the advantages offered by these algorithms, combining the strengths of Gauss-Newton and gradient descent methods, allowing them to converge more rapidly towards the optimal solution. The relatively poor performance of gradient descent may be due to getting stuck in local minima, while LM is not affected by this problem as it can adjust the step size based on the curvature of the error surface. Furthermore, the superior performance of LM can be attributed to the properties of the data itself. As a nonlinear function, LM excels in capturing the second-order information of the error surface.

Personal Regression

Following the instructions, I created a new nonlinear function, denoted as Tnew, using the formula: Tnew = (8T1 + 7T2 + 7T3 + 6T4 + 4T5)/(8 + 7 + 7+ 6 + 4). To ensure consistency in the training, validation, and testing sets, a seed of 87764 was utilized. Based on previous experiments, it was found that a neural network with 50 neurons in a single hidden layer could approximate the nonlinear function very effectively after 1000 training iterations. As the new function is simply a linear combination of five nonlinear functions, the model configuration remains the same.

In comparison to the quasi-Newton and gradient descent algorithms, the Levenberg-Marquardt algorithm demonstrates rapid convergence in the initial rounds of training. Hence, it is expected that LM will outperform the other algorithms in terms of convergence speed and learning accuracy. Given the slow learning speed of gradient descent, it will not be employed in this case. However, I am interested in exploring the use of conjugate gradient, which is well-known for handling high-dimensional data (although it is not applicable in this particular case). For comparison purposes, both the Fletcher-Reeves and Polak-Ribiere conjugate gradient algorithms will be utilized. Since the number of training iterations is fixed at 1000, it will also be interesting to examine the computational efficiency of each method.

Additionally, I have included a 3D dimension plot of the training data for reference.

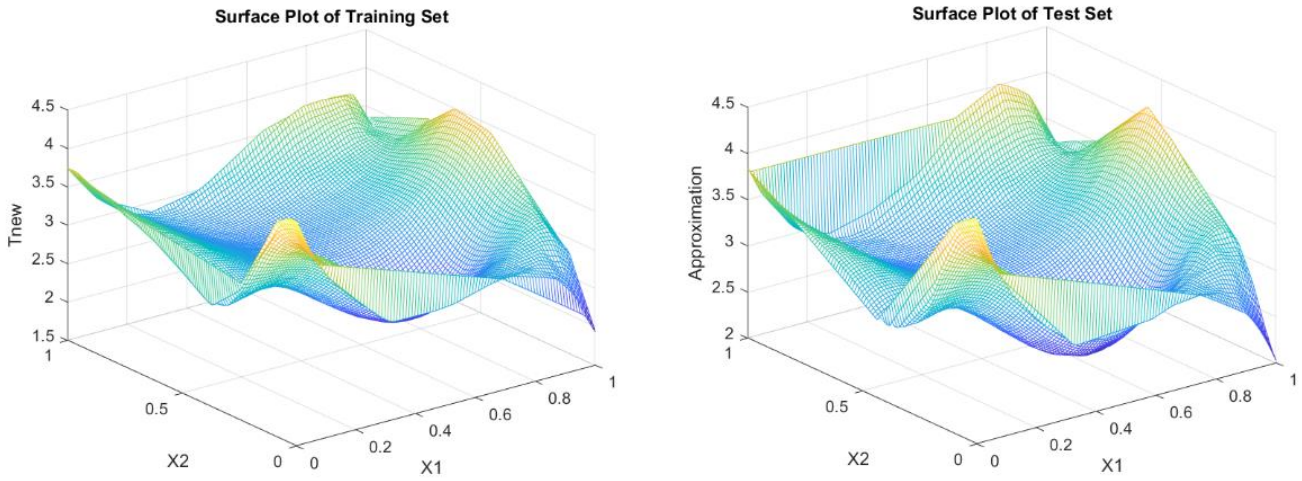


Figure 3: 3D Surface Plot of Training and Testing Data for Regression Approximation

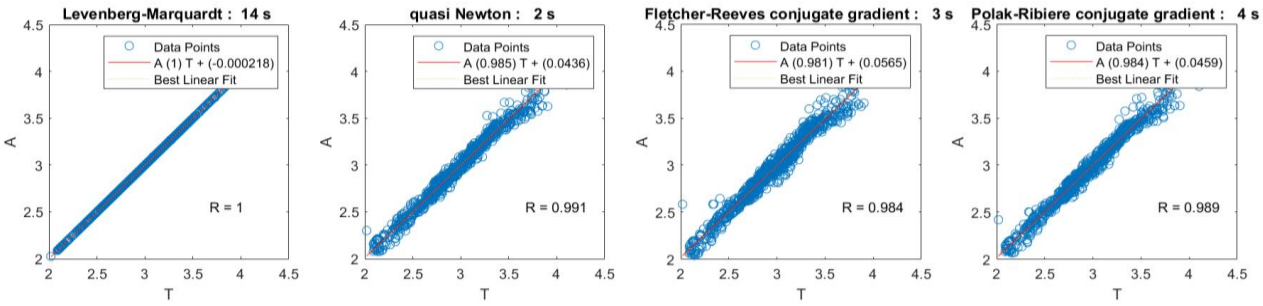


Figure 4: Correlation Results of 4 Network Models with 50 Neurons in a Single Hidden Layer based on Different Algorithms over Time for Training

After 1000 training iterations, all the networks perform well, with notable accuracy. However, it should be noted that the LM-based model, despite being the most accurate, requires the longest training time compared to the other methods. On the other hand, the remaining three methods exhibit similar performance in terms of both running speed and accuracy. Considering the running time, I would recommend choosing the model based on the quasi-Newton algorithm, as it only takes 2 seconds to complete the training process.

To assess the error levels, error curves have been plotted. As expected, the network based on the LM algorithm outperforms the others, while the rest show comparable error surfaces, albeit with small errors overall. The mean square error for the testing set has also been calculated in table 1.

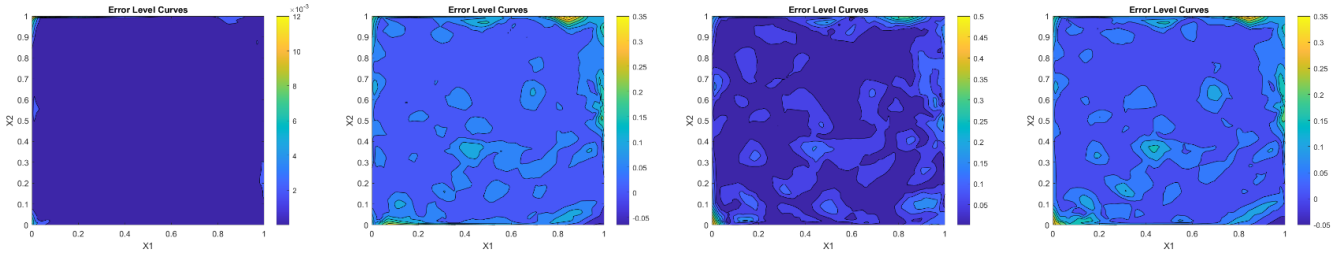


Figure 5. Error Level Curves for LM, Quasi, Fletcher-Reeves, and Polak-Ribiere Models

Table 1: Training and Testing RMSE for 4 Network Models with Neurons

RMSE	LM	Quasi	Fletcher-Reeves	Polak-Ribiere
Training	0.2995	0.29739	0.29824	0.29732
Testing	0.31715	0.31318	0.31553	0.31339

All the models effectively capture the underlying data pattern, with training and testing mean square errors reaching similar levels. Although testing errors are slightly higher than those of the training set, the difference is minimal. Considering the running time and overall performance, the networks based on the quasi-Newton, Fletcher-Reeves, and Polak-Ribiere algorithms perform well.

The LM algorithm is renowned for its fast convergence speed. However, in this case, it took the longest time to complete the 1000 training iterations. This can be attributed to the additional computations involved in calculating the Jacobian matrix and solving a system of linear equations. In contrast, the other three algorithms demonstrate higher computational efficiency.

To further enhance the model, setting more checkpoints within the 1000 epochs could be beneficial. This would allow for early identification of models that have already converged, potentially reducing the overall running time, particularly for the LM-based model, which has shown the ability to capture the nonlinear data pattern in the initial iterations.

Byesian Exercises

Table 2: Testing RMSE for 4 Network Models with Neurons using Bayesian Regularization Algorithm

test errors	50 neurons without noise	100 neurons without noise	200 neurons without noise	200 neurons with noise
Levenberg-Marquardt	0.027786	0.141799	0.309458	0.245703
quasi Newton	0.002727	0.035202	0.169181	0.279420
Bayesian Regularization	0.000024	0.000000	0.030443	0.217398

As observed in the table, when the number of neurons is small (50), all models utilizing different algorithms successfully learn the training data without experiencing overfitting. However, as the number of neurons increases, the test errors for all models also increase. Notably, in all cases, the model based on Bayesian regularization consistently outperforms the other two models. This superior performance can be attributed to the incorporation of prior knowledge in Bayesian regularization, where a regularization term is introduced to penalize complex or extreme parameter values. This regularization helps to prevent overfitting and ensures a more generalizable model.

Exercise session 2

Hopfield Network

In the case of 2 neurons with T = [1 1; -1 -1; 1 -1]T, it has been observed that after testing, 50 iterations are sufficient for any random points in this case to reach their corresponding attractors. However, there is one "fake" attractor located at (-1, 1). This unwanted attractor is a result of spurious states, indicating an overrepresentation in the weight matrix.

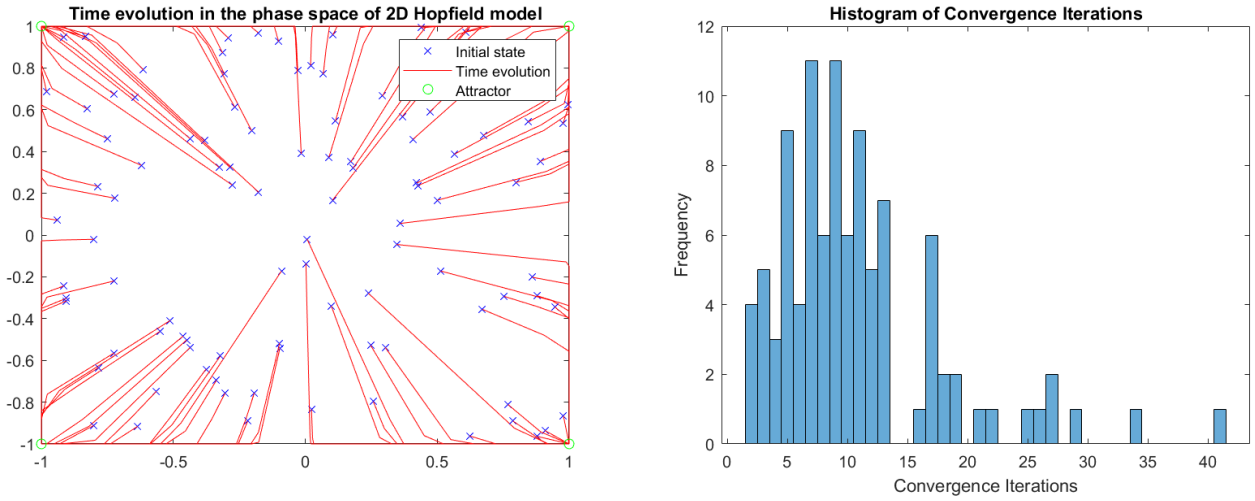


Figure 1. Left: Trajectory of 100 random points after 1000 iterations using a Hopfield model in 2D dimension. Right: Convergence iterations for the 100 points.

In this case, the weight matrix is calculated as W = TT' = [1 1; -1 -1; 1 -1] * [1 -1 1; 1 -1 -1] = [2 0; 0 2]. Let's consider an initial point [-1, 1]. When multiplied by the weight matrix, W*P = [2 0; 0 2] * [-1; 1] = [-1, 1]. This causes the point to become "stuck" in its current location, leading to it being perceived as an attractor because it does not change its position.

Fortunately, the stability in this case is satisfactory, as every randomly generated point eventually reaches its corresponding attractor.

In the case of 3 neurons with T = [1 1 1; -1 -1 1; 1 -1 -1]', testing has shown that 50 iterations are adequate for any random points to reach their corresponding attractors. Additionally, no unwanted attractors are present in this scenario, and the stability of the system is maintained.

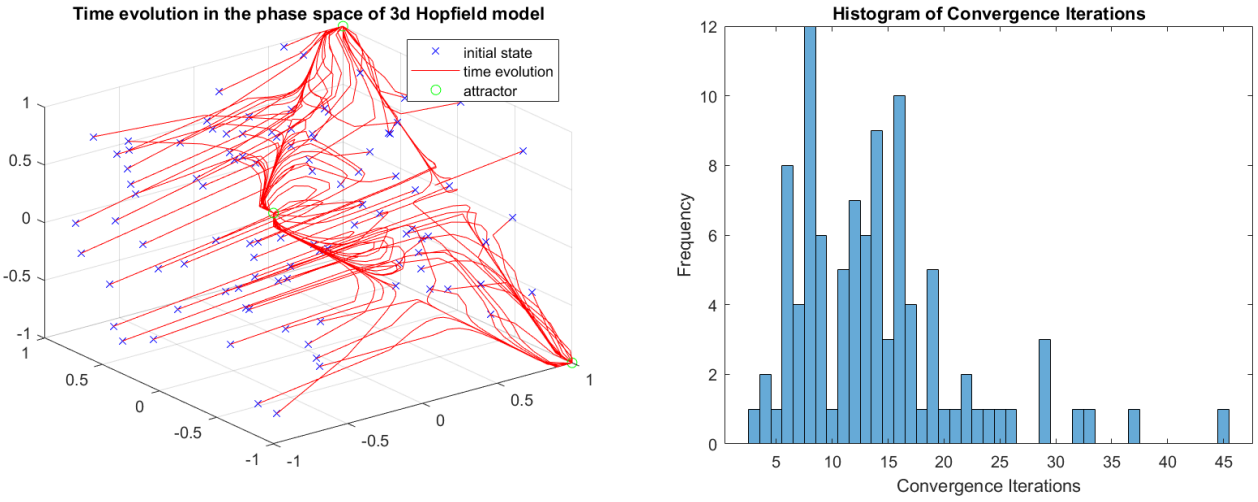


Figure 2. Left: Trajectory of 100 random points after 1000 iterations using a Hopfield model in 3D dimension. Right: Convergence iterations for the 100 points.

The Hopfield model demonstrates the ability to reconstruct noisy digits within a reasonable range of noise levels that do not

overlap with the patterns of the reference data. However, it is crucial to have a sufficient number of iterations to allow the input states to reach the attractors.

Through trial and error, it has been observed that when the noise level reaches 4 and the model is run for 1000 iterations, the reconstruction process becomes highly random. However, by increasing the number of iterations to 10000, the model can still successfully reconstruct the digits even at this noise level.

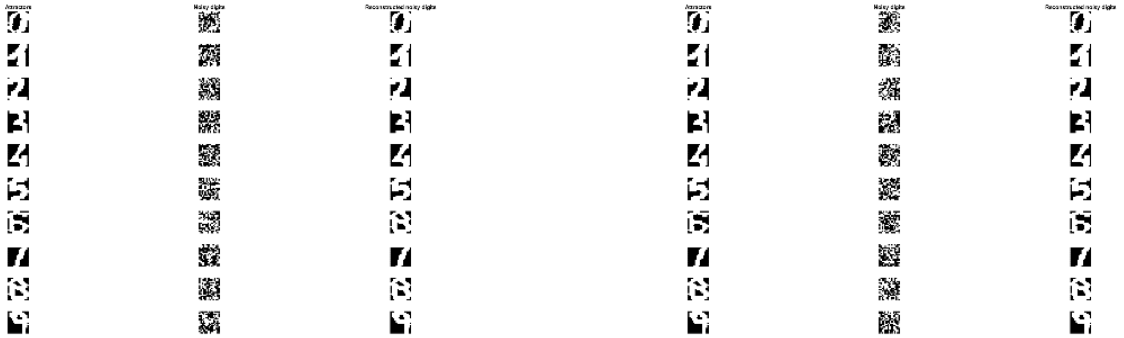


Figure 3. Left: Digitals without noise (left column), digitals with level 4 noise added (middle column), and reconstructed digitals after applying the Hopfield model for 1000 iterations (right column). Right: Same as before, but after 10000 iterations.

The figures illustrate that after 1000 iterations, the digit that was initially supposed to be a "6" is reconstructed as an "8". However, when the noise level is increased to 5, not only one but several digits are reconstructed incorrectly. Even after 10000 iterations or even 100000 iterations, some digits continue to be reconstructed erroneously. In general, a higher noise level requires more iterations for accurate reconstruction.

These findings highlight the importance of both the noise level and the number of iterations in achieving reliable digit reconstruction using the Hopfield model.

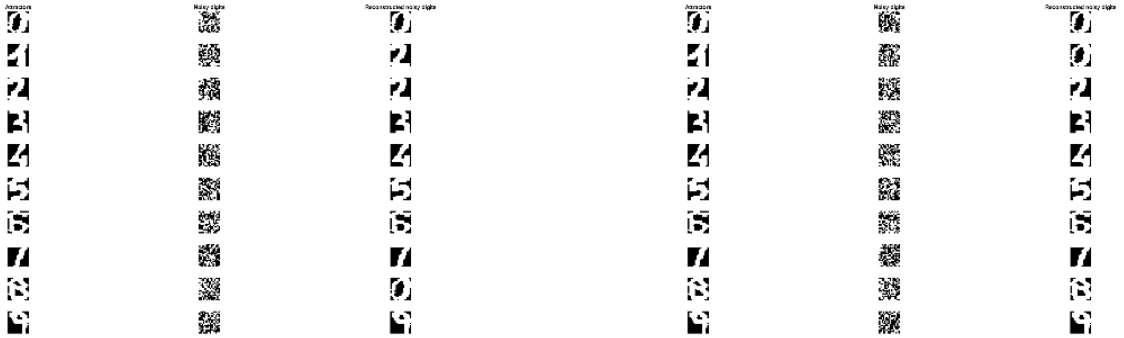


Figure 4. Left: Digitals without noise (left column), digitals with level 5 noise added (middle column), and reconstructed digitals after applying the Hopfield model for 10000 iterations (right column). Right: Same as before, but after 100000 iterations.

Neural network

To assess the model's performance in predicting the next 100 points based on the previous 1000 time points in a time series, 12 different combinations of neurons (10, 20, 50, and 100) and lags (1, 5, and 10) were selected, as shown in the table. All models underwent 200 iterations and utilized the Bayesian Regularization algorithm.

The results reveal that the model with 50 neurons and 5 lags successfully learns the training dataset. However, when it comes to prediction, particularly for the testing data, this model fails to capture the pattern adequately, especially in the range of the next 60 to 100 measurements. Conversely, the model with 20 neurons and 5 lags performs better in matching the pattern of the 60 to 100 part, but exhibits larger variations in the 30 to 60 measurements.

Overall, the simple MLP (Multi-Layer Perceptron) models are capable of accurately capturing the pattern of the training data. However, in terms of prediction performance, there is room for improvement as the models do not yield satisfactory results.

Table 1. Root Mean Square Error (RMSE) of MLP models with different combinations of neurons numbers and lags.

RMSE	10 Neurons	20 Neurons	50 Neurons	100 Neurons
1 lags	68.8125	59.9912	64.3277	68.116
5 lags	57.8268	58.6861	**55.2528**	1967.1711
10 lags	441.7451	515.1958	79.3029	394.9321

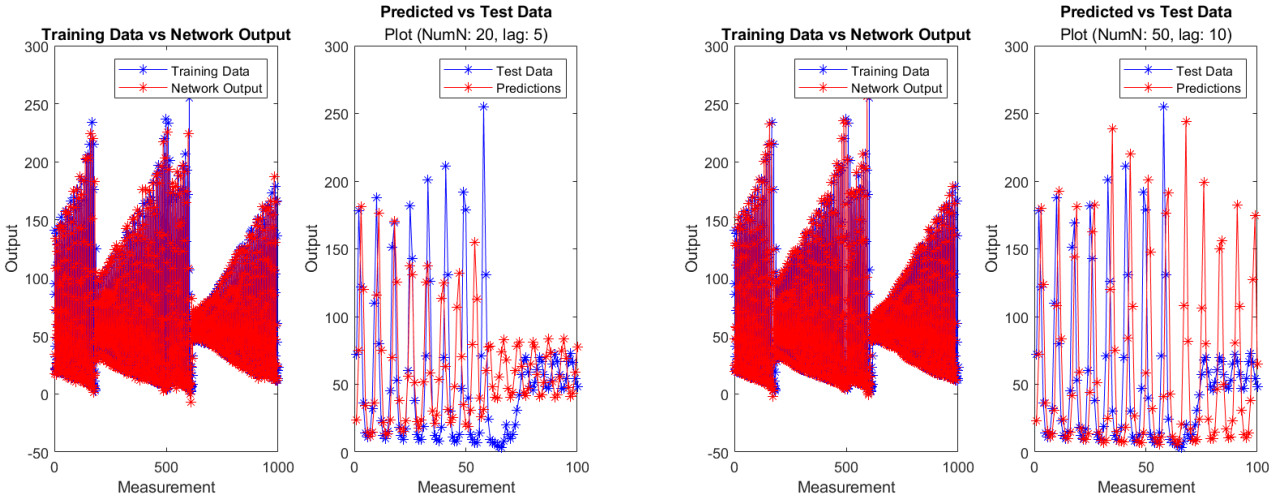
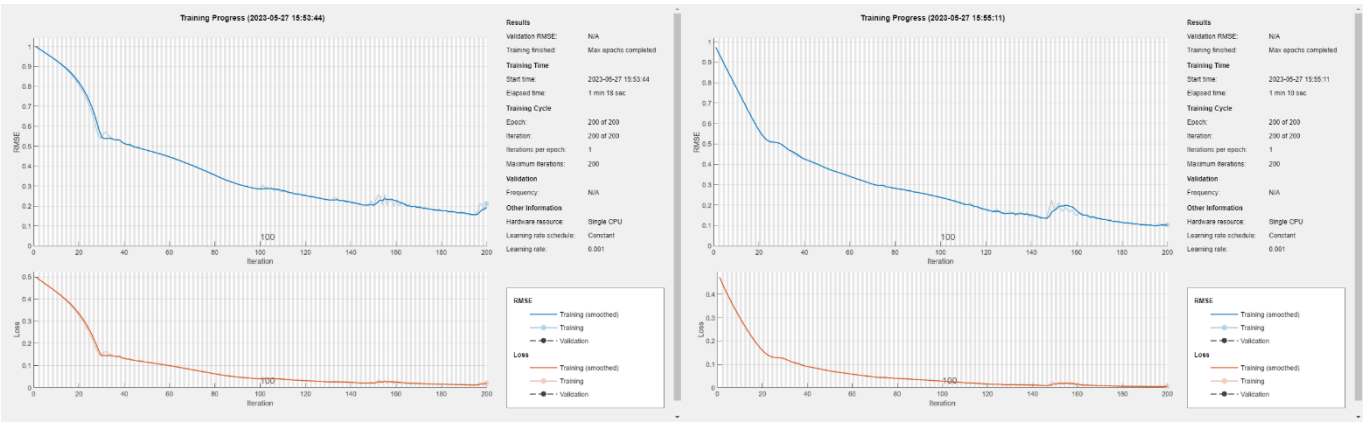


Figure 5. Mapping of both training data and testing data using the learned MLP models. Left: MLP model with 20 neurons and lag set to 5 after 200 iterations. Right: MLP model with 50 neurons and lag set to 10 after 200 iterations.

Long short-term memory network

Based on the previous network configuration, it was found that the network with 50 neurons and 5 lags achieved the best performance after 200 iterations. However, considering that Long Short Term Memory (LSTM) networks are renowned for their effectiveness in handling time series data, I decided to experiment with different lag values (1, 5, 10, and 20) while using 100 neurons. This allowed me to compare the performance of the LSTM network with the normal recurrent neural networks (RNNs) we previously constructed. For optimizing the loss, the Adaptive Moment Estimation (Adam) algorithm was used as the default choice.



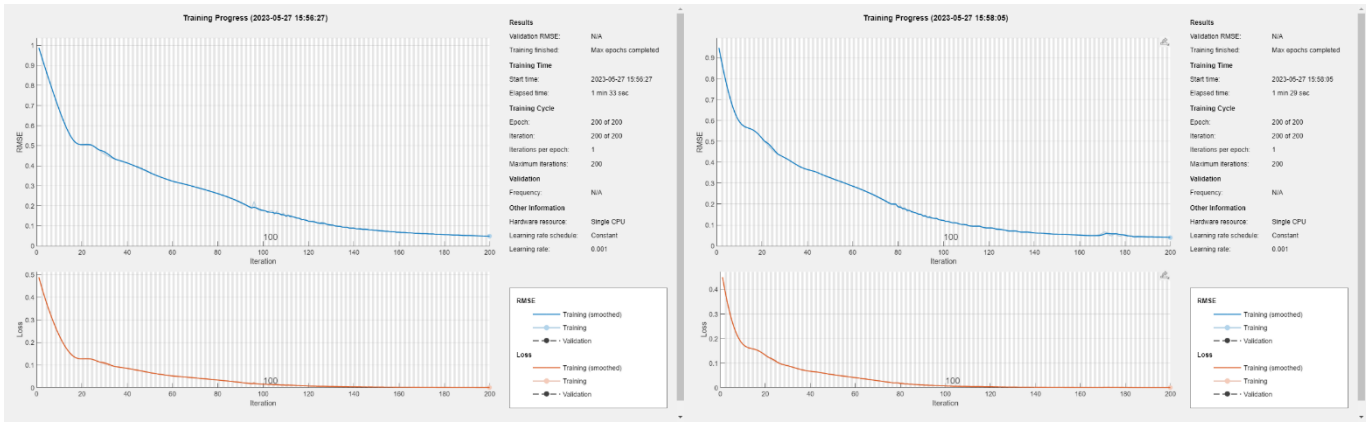


Figure 6. Training progress of four models:Upper left: Model 4 with 20 lags. Upper right: Model 3 with 10 lags. Bottom left: Model 2 with 5 lags. Bottom right: Model 1 with 1 lag.

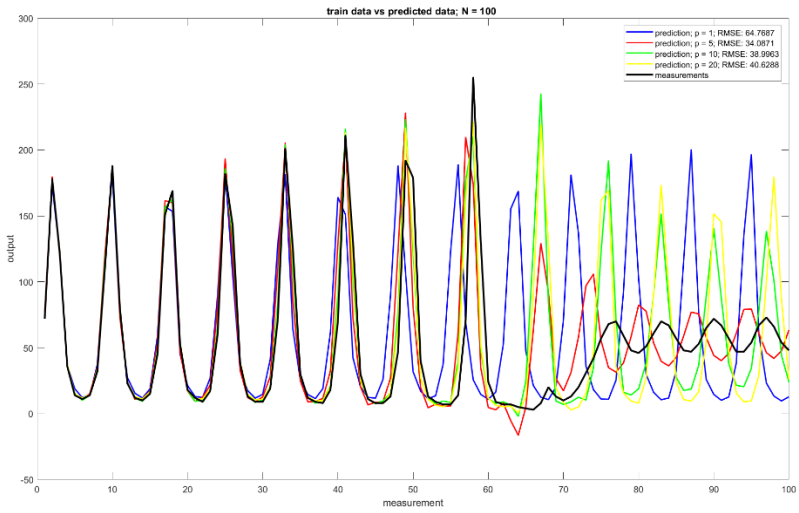


Figure 7. Prediction results from four trained models with RMSD (Root Mean Square Deviation) evaluation.

From the training process figures, it is evident that all models were able to achieve nearly 0 loss and RMSE values ranging from 0.1 to 0.2. This indicates that the models successfully captured the patterns in the training data. As the number of lags increased, the models exhibited lower RMSE values, which is expected since more input data was provided for predicting the next point.

The prediction result plot, along with the corresponding RMSE values, further confirms that all models performed significantly better than traditional recurrent neural networks in terms of RMSE. Particularly, for the next 60 points, all models closely matched the target data. However, beyond that point, the fluctuations deviated from the true pattern.

The model with 5 lags outperformed the other models, especially those with higher lags. This suggests that models with higher lag values may suffer from overfitting the training data. Additionally, the MLP model with the same architecture (100 neurons and 5 lags) exhibited the highest RMSE value of 1967.1711, indicating that the LSTM model is particularly effective in handling time series data.

In summary, both conventional recurrent neural networks and LSTM models were able to capture the patterns in the time series training data effectively. However, when it comes to prediction, the LSTM model outperformed the RNN due to its specialized configuration. Among the LSTM models, higher lag values did not necessarily result in better prediction performance due to the overfitting issue.

Exercise session 3

Handwritten Digits PCA and reconstruction

The 'threes.mat' dataset was loaded, consisting of 500 samples with a size of 256. Eigenvalues and eigenvectors were computed, and the eigenvalues were plotted in descending order. Starting from 31.12, the second eigenvalue drops to 15.8497, followed by 14.2758, 12.7420, 11.7322, and 9.0008. The drop rate is relatively slow, indicating that the initial eigenvectors do not explain a significant portion of the variance. This observation is further supported by reconstructing the digit 'three' using one, two, three, and four principal components. Even with four principal components, the reconstructed 'three' in the plot still exhibits low resolution.

To quantify the reconstruction quality, the reconstruction errors were calculated using different numbers of principal components, ranging from 1 to 50. As shown in the figure, increasing the number of principal components reduces the reconstruction error. When all principal components are used, the reconstruction error is calculated to be 1.7003e-15, which is nearly zero and can be considered negligible. This implies that all principal components effectively represent the entire dataset in this case.

Additionally, the squared reconstruction error induced by excluding a specific principal component has also been plotted alongside the reconstruction error. As the index of the excluded principal component increases, the reconstruction error approaches 1, indicating that the excluded principal component plays a diminishing role in explaining the variance of the data.

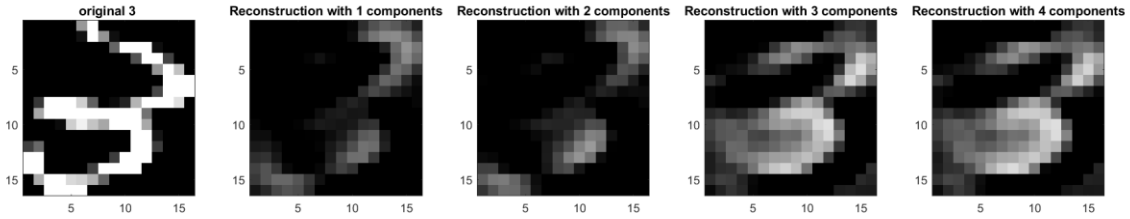


Figure 1. Reconstruction of the original digit "3" using different numbers of principal components (PCs). From left to right: original digit "3", reconstructed digit "3" using one PC, reconstructed digit "3" using two PCs, reconstructed digit "3" using three PCs, and reconstructed digit "3" using four PCs.

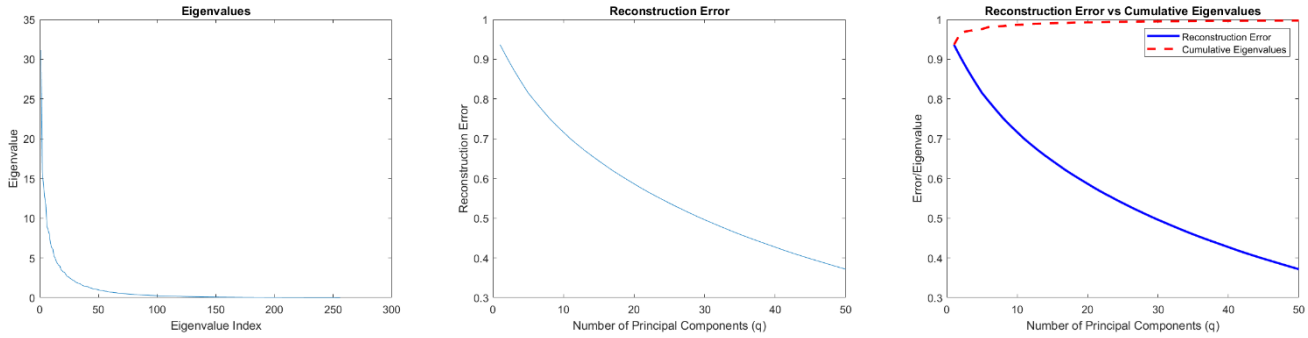


Figure 2. Left: Eigenvalue plotted against the eigenvalue index. Middle: Reconstruction error using different numbers of principal components (PCs). Right: Squared reconstruction error induced by excluding a specific principal component for reconstruction.

Digit Classification with Stacked Autoencoders

Similar to PCA, stacked autoencoders are employed to achieve dimension reduction and extract the essence of the dataset in a more efficient manner. By mapping the entire dataset into a lower-dimensional space represented by a set of neurons, stacked autoencoders can capture the underlying structure effectively. In this exercise, the compressed information obtained from the autoencoders is utilized as input to a supervised learning network for digit classification. The architecture of different network models is described in the table.

The default settings yield an accuracy of 99.66% using the following construction. The input data is mapped to the first layer consisting of 100 neurons for 400 epochs. Subsequently, the first layer is mapped to the second layer comprising 50 neurons

for 200 epochs. Finally, the supervised learning labels undergo 400 epochs for training.

Table 1. Different architectures of stacked autoencoders to explore and understand the relationship between hyperparameters and model performance.

	Default	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6
First layer: neuron number(N1)	100	100	50	20	20	20	10
First layer: Epoch(E1)	400	200	200	200	100	100	100
Second layer: neuron number(N2)	50	50	25	10	10	NA	NA
Second layer: Epoch(E2)	100	50	50	50	10	NA	NA
Epoch for supervised learning(e3)	400	200	200	200	200	100	100
accuracy before supervised learning	81.4%	60.2%	30.8%	17.3%	24.0%	63.1%	42.5%
accuracy after supervised learning	99.66%	**99.72%**	99.40%	58.46%	99.00%	99.24%	97.88%

After conducting trial and error, I was surprised to find that higher accuracy can be achieved by reducing the number of epochs for each layer. By reducing E1 from 400 to 200, E2 from 200 to 100, and E3 from 400 to 200, I obtained a model with an accuracy of 99.72%. This suggests that excessive epoch times may lead to overfitting, and the improvement from 98.66% to 99.72% cannot be considered "significant."

To further investigate the effects of the number of neurons, I reduced N1 from 100 to 50 and N2 from 50 to 25. In this case, the model achieved an accuracy of 99.40%. However, it is important to note that the initial performance of the model without tuning was poor, with only around 30% accuracy. I continued to reduce N1 from 50 to 20 and N2 from 25 to 10, which resulted in a model with an accuracy of 58.46%. Inspired by the success of reducing the number of epochs, I adjusted E1 from 200 to 100 and E2 from 50 to 10, leading to model 4 with 99% accuracy, which met our expectations. Therefore, the number of training times should be adjusted based on the number of neurons, as the model's performance tends to worsen otherwise (overfitting problem).

To investigate how many layers are needed to achieve better performance than a normal neural network, I directly removed the second layer from model 4 and trained model 5. Surprisingly, it performed even better than model 4, particularly with around 63% accuracy before supervised learning. The accuracy of model 5 was 99.24%, which surpassed the accuracy of the two normal neural networks at 95.8400 and 96.5800, respectively. I also attempted to reduce the number of neurons in the first layer to 10, which still outperformed the two normal networks.

The final fine-tuning step of training the stacked autoencoders in a supervised way is crucial for the accuracy of the model, as demonstrated by the significant improvement seen in models 2 and 4, with an increase of approximately 70% in accuracy. The reason behind the substantial improvement through fine-tuning is that before this step, each layer of the stacked autoencoders attempts to capture the patterns of the input data in an unsupervised manner, which may not be specific to the classification task. However, by utilizing the labels to update the parameters, the learned features are tailored to match the targets as closely as possible. As a result, the predictions become more accurate, similar to other classification models. Moreover, the dimensionality reduction of the input data in stacked autoencoders helps eliminate noise, leading to better classification performance compared to other models.

Digit Classification with CNN

1. The size command provides information about the dimensions of the convolved features or filters in a convolutional

neural network (CNN). In this case, the result [11 11 3 96] indicates that each filter has a size of 11x11, and there are 96 filters. The number 3 represents the three color channels (red, green, and blue) of the input images.

2. The dimension of the output at layer 5 is 27x27x96. Starting with an input size of 227x227x3, the second layer of the CNN provides 96 filters with a size of 11x11x3. The stride, which determines the step size of the filter when convolving over the input, is 4x4. As a result, the output of the second layer is a new matrix with a size of 55x55x96. The ReLU (Rectified Linear Unit) and Cross Channel Normalization layers do not affect the dimension of the input, so we can move directly to layer 5. In layer 5, a 3x3 pooling matrix with a stride of 2x2 is applied, reducing the size of the output to 27x27x96.
3. The dimension of the output at the end of the CNN architecture is 6x6x256. This is a significant reduction compared to the initial dimension of the input, which was 227x227x3. By applying various convolutional and pooling layers, the CNN progressively reduces the dimensionality of the data while extracting relevant features. In this case, about 95% of the data has been discarded or compressed before reaching the classification task, resulting in a more computationally efficient process.

Table 2. Different architectures of CNN to explore and understand the relationship between hyperparameters and CNN model performance for supervised digit recognition.

	Default	Model1	Model2	Model3	Model4	Model5	Model6	Model7
First layer: Filter(F1)	5*5	5*5	5*5	5*5	5*5	5*5	5*5	5*5
First layer: Number of channels (nF1)	12	12	24	24	24	24	24	24
Second layers: Filter(F2)	5*5	5*5	5*5	5*5	5*5	5*5	5*5	5*5
Second layers: Number of channels(Nf2)	24	24	24	24	24	24	48	48
Third layers: Filter(F3)	NA	NA	NA	5*5	5*5	NA Dropout	NA	NA dropout
Third layers: Number of filter(nF3)	NA	NA	NA	6	24	NA	NA	NA
ConnectedLayer Neurons(N)	10	10	10	10	10	10	10	10
MaxEpochs(M)	15	30	30	30	50	40	30	30
Accuray	0.8212	0.9408	0.9648	0.1428	0.9544	0.9736	0.9844	0.9876
Time(seconds)	53.4	77.5	113.4	111.8	252.7	195.9	152.8	158.8

- Model1: Increased the number of epochs (M) from 15 to 30, resulting in improved accuracy of 0.94.
- Model2: Increased the number of channels (nF1) from 12 to 24, leading to increased accuracy of 0.96.
- Model4: Added additional layers with 24 filters (5x5) but observed a decrease in accuracy and longer training time. The model did not improve.
- Model5: Identified overfitting in Model2 and introduced a dropoutLayer with a rate of 0.1 before the fullyConnectedLayer. This regularization technique aims to prevent overfitting by randomly dropping out a fraction of the neurons during training. The accuracy improvement was not mentioned.
- Model6: Increased the number of filters in the second layer (nF2) to further enhance performance. The accuracy improvement was not mentioned.
- Model7: Introduced a dropoutLayer to Model5, but no further improvement in accuracy was observed.

Overall, increasing the number of epochs and the number of channels in the first layer resulted in improved accuracy. The introduction of additional layers did not lead to better performance, but dropout regularization and increasing the number of filters in the second layer showed potential improvements.

Exercise session 4

Restriced Boltzmann Machines

1. Reconstructing unseen test images

Table 1. Different architectures for RBM models to understand and evaluate the relationship between hyperparameters and RBM model performance for reconstructing unseen test images.

	Default	Model1	Model2	Model3	Model4	Model5	Model6	Model7	Model8
Number of components	10	100	100	200	200	500	500	1000	1000
Epochs	10	30	30	50	50	30	30	30	30
Learning rate	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
Gibbs sampling steps	10	20	50	20	50	50	500	10	1

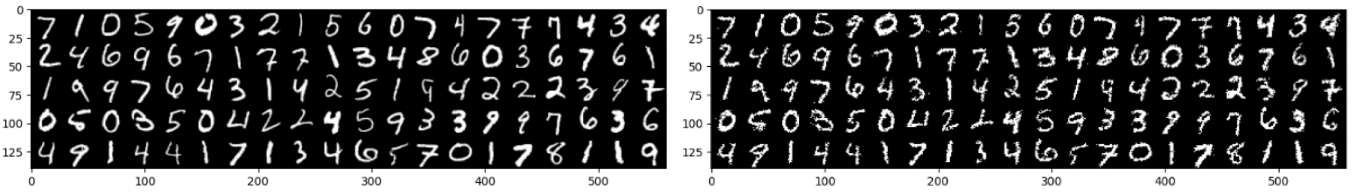


Figure 1: Reconstruction of unseen test images using different Gibbs sampling steps. The left figure is the original test images, while the right figure shows the result after just 1 Gibbs sampling step with 1000 components after 30 epochs.

Based on the performance of the models, it can be concluded that having a sufficient number of components and epochs is key to building a model that captures the pattern of the training data effectively. The plots of the models with 100, 200, and 500 components show that they have not fully learned the training data, resulting in unclear or incorrect digitals in the generated images. Consequently, increasing the number of Gibbs steps would exacerbate the issue, as these models have learned incorrect data patterns, which would be propagated during the generation process, leading to undesirable outcomes.

In contrast, the model with 1000 components and 30 epochs demonstrates a superior understanding of the data pattern. As depicted in the figures, after just one Gibbs step, the generated images closely resemble the target images. However, increasing the number of Gibbs steps in this case would actually worsen the results (not shown in figure). This occurs because the generated images progressively converge toward the training data, limiting the diversity required for generating "new data."

Hence, striking the right balance between the number of components and epochs is critical for achieving accurate and diverse image generation. Insufficient components or epochs can result in inadequate learning, while an excess of components or Gibbs steps may lead to overfitting and a loss of diversity in the generated images.

2. Reconstruct missing parts of images

Table 2: Evaluation of different architectures for RBM models in reconstructing missing parts of images and understanding the relationship with hyperparameters.

	Default	Model1	Model2	Model3	Model4	Model5	Model6
Number of components	10	1000	1000	1000	200	200	200
Epochs	10	30	30	30	50	50	50
Removed rows(start,end)	[0,12]	[0,12]	[0,15]	[12,24]	[0,12]	[0,12]	[0,12]

Learning rate	0.01	0.01	0.01	0.01	0.01	0.02	0.05
Gibbs sampling steps	10	20	100	100	10	10	10
pseudo-likelihood					-65	-70	-78

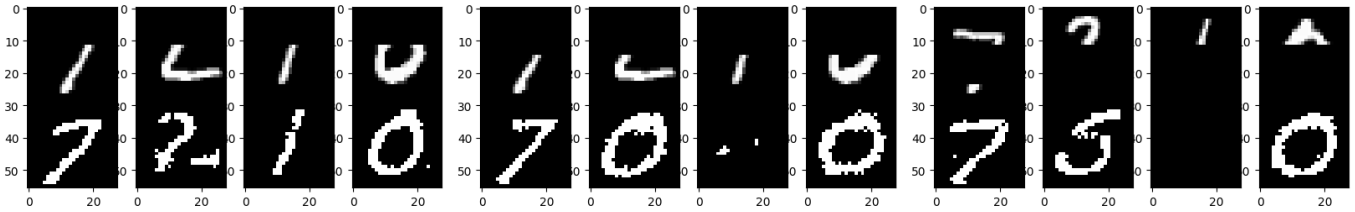


Figure 2: Results of reconstructing missing parts of images from models with different architectures. Left: Reconstruction results from Model 1. Middle: Reconstruction results from Model 2. Right: Reconstruction results from Model 3.

Based on the previous models, the model with 1000 components was selected for image repair tasks. Once again, the number of components and epochs played a crucial role in capturing the data pattern effectively. Given the importance of accurately understanding the training data's pattern, achieving a comprehensive grasp of it is essential for successful image repair. In this context, using a sufficiently high number of components and epochs becomes imperative.

Regarding the impact of the learning rate, ideally, experiments should be conducted on the model with 1000 components. However, due to the extensive training time required for this model, I opted to test different learning rates on the model with 200 components after 50 epochs. Since this model had not yet fully learned the data pattern, the performance of all learning rates was poor. Notably, model 6 exhibited even worse performance, likely because it became trapped in a local minimum, as indicated by the end of the pseudo-likelihood.

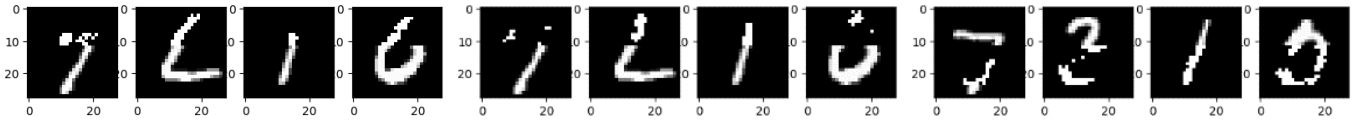


Figure 3: Comparison of the effects of learning rate on the performance of image repairing based on the model with 200 components after 50 epochs. Left: Results generated from the model with a learning rate of 0.01. Middle: Results generated from the model with a learning rate of 0.02. Right: Results generated from the model with a learning rate of 0.05.

Increasing the number of rows to be removed from 12 to 15 in the current model resulted in the inability to effectively repair the digitals. This can be attributed to the significant loss of information caused by removing more rows. The model struggled to capture the essential details necessary for accurate repair.

Furthermore, the current model faced challenges when attempting to handle the moving of the removal block to the middle of the images at coordinates (12,24). This difficulty arose due to the fact that a substantial amount of important information is typically concentrated in the middle of each image. As a result, the model struggled to accurately repair the digitals when crucial details were removed from the central region.

Deep Boltzmann Machines

1. The interconnection weights or filters between the last trained RBM and the new trained DBM are quite similar. Both models aim to learn the dependencies between the visible layer and the first hidden layer. In the second layer, the filters capture the dependencies between the first hidden layer and the second hidden layer to extract higher-level features. The figures demonstrate that the filters in the second layer show patterns of white or black blocks, rather than small edges or dots.

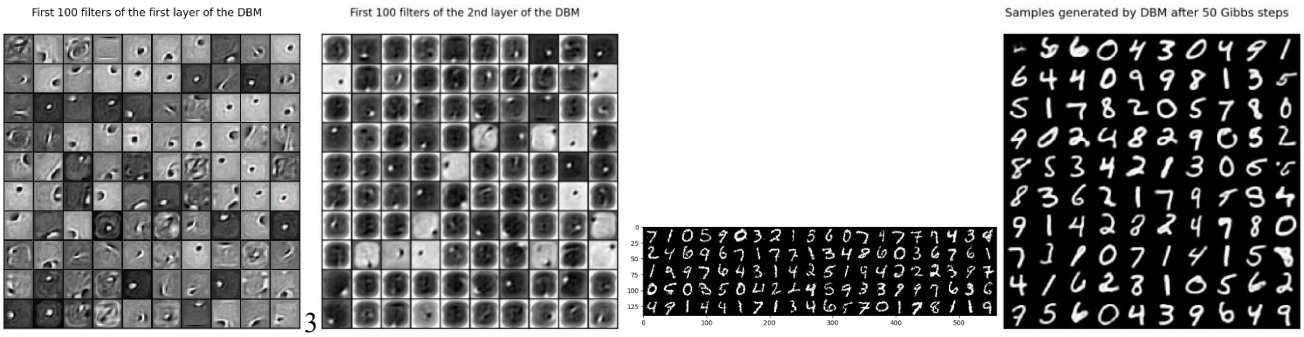


Figure 4: First 100 filters of the first and second layers of DBM. The digitals in the rectangle were generated from the RBM model after 1 Gibbs step. The digitals in the square were generated from the DBM model after 50 Gibbs steps.

2. The generated digit images from the DBM have a higher resolution (100*100) compared to those from the RBM (1000*1). This is attributed to the nature of the DBM, which has a multiple layer structure that allows it to learn more abstract representations of the training data. As a result, the image generation from the DBM is more visually appealing and realistic.

Generative Adversarial Networks

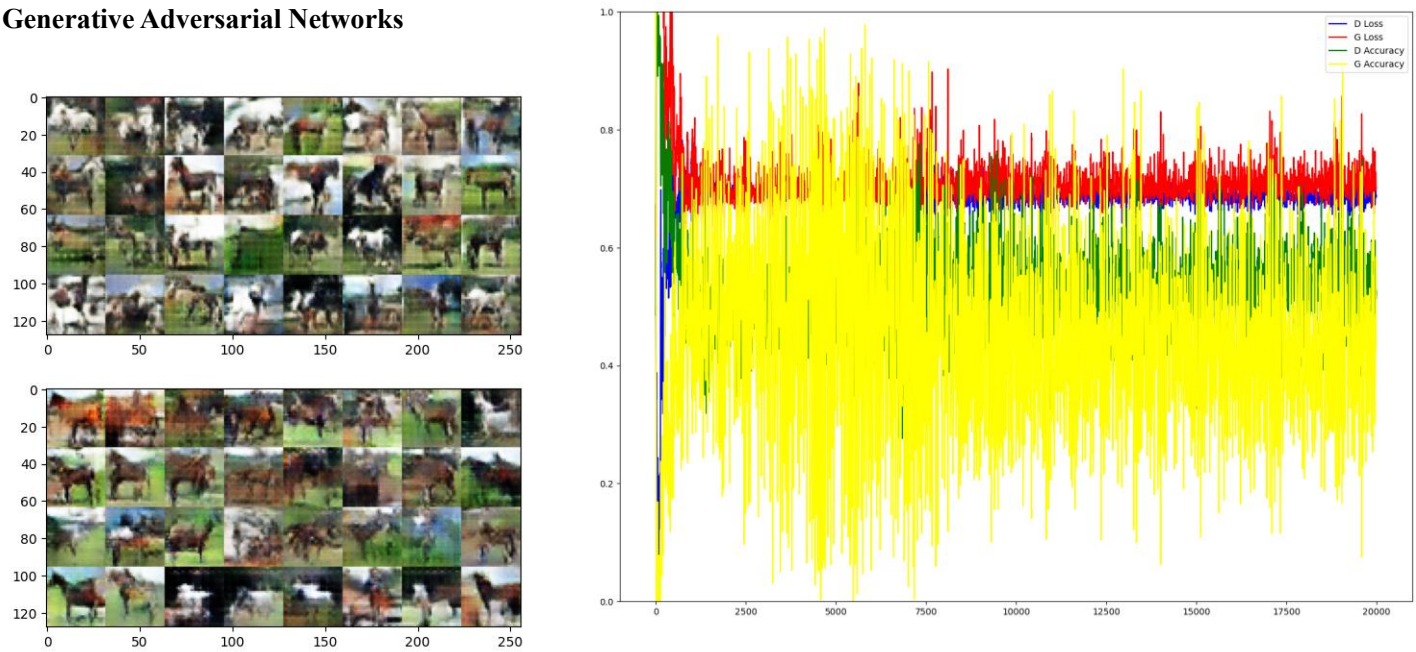


Figure 5: Generated "horses" images from the generator. The top image was generated after 19800 batches. The bottom image was generated after 20000 batches.

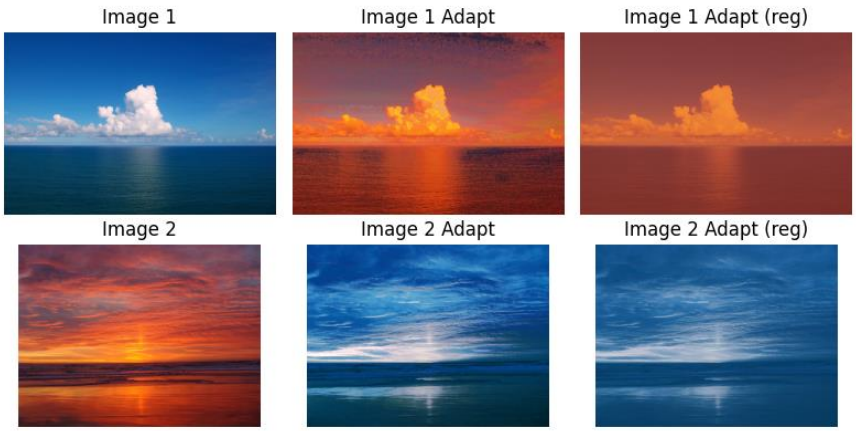
The Deep Convolutional Generative Adversarial Network (DCGAN) architecture, was downloaded from the GitHub repository: https://github.com/hannesdm/gan-tools2 directly. I selected class 7, which corresponds to "horses" pictures from the CIFAR dataset.

After training the model with 20,000 batches, I plotted the loss and accuracy for both the generator and discriminator. From the plot, it can be observed that the model converged around 9,000 batches. The loss for both the discriminator and generator stabilized at around 0.7. The accuracy of the discriminator fluctuated around 0.55, while the accuracy of the generator showed more severe fluctuations, but with an average around 0.4 towards the end. To present the dynamics of the loss and accuracy more clearly, we adjusted the y-axis range from 0 to 1.

Regarding the stability of the training process, as mentioned earlier, the dynamics of the four indicators indicate that the model reached a stable stage after 9,000 batches. The loss and accuracy values for both the generator and discriminator fluctuated

around their means. It is important to note that unlike other deep learning models, the fluctuations in this model are quite large. This is because the generator aims to generate realistic samples to fool the discriminator, while the discriminator strives to distinguish between real and generated samples. This dynamic interplay between the two components leads to the observed fluctuations. I also included two figures containing pictures generated by the generator. Upon visual inspection, most of the images resemble horses, but some may appear closer to cows. However, it is worth mentioning that all the generated pictures have a blurry quality.

Optimal Transport



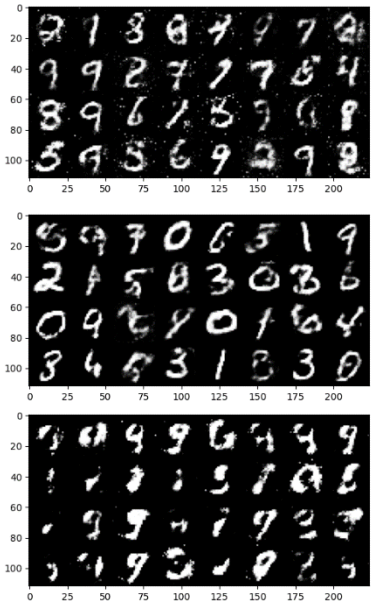


Figure 5: Image color swapping using optimal transport and generated digitals from GAN models. Left: Images with colors swapped using optimal transport.
Right: Digitals generated from different GAN models: Top: Standard GAN Middle: Wasserstein GAN with weight penalty Bottom: Wasserstein GAN with weight clipping

In this experiment, Optimal Transport (OT) was employed to transfer the colors between two images: an ocean during the daytime with a blue sky and an ocean during sunset. Initially, the colors of the two images were quite distinct, with one having a blue tone and the other an orange hue. After applying OT, the daytime ocean appeared to have an orange filter applied to it instead of achieving an optimal color swap. Some remnants of the blue sky were still visible. Similarly, the ocean during sunset had certain dark regions that became milder after undergoing the OT process.

Comparing the performance of two versions of Wasserstein Generative Adversarial Network (GAN) with the standard GAN, it can be observed that both versions of Wasserstein GAN exhibit worse results in terms of visual quality. This could be attributed to the simplistic architecture design, specifically a simple fully connected architecture, and possibly suboptimal choices for hyperparameters in the Wasserstein GAN. However, it is worth noting that the stability of the Wasserstein GAN during training is significantly better than that of the standard GAN, as depicted in Figure 6. This improved stability is a result of the underlying principles of the Wasserstein GAN, which utilizes Wasserstein distance instead of Jensen-Shannon Divergence or Kullback-Leibler Divergence used in the standard GAN. Furthermore, the introduction of the Lipschitz Constraint on the discriminator's weights in Wasserstein GAN allows for more stable training dynamics and gradients.

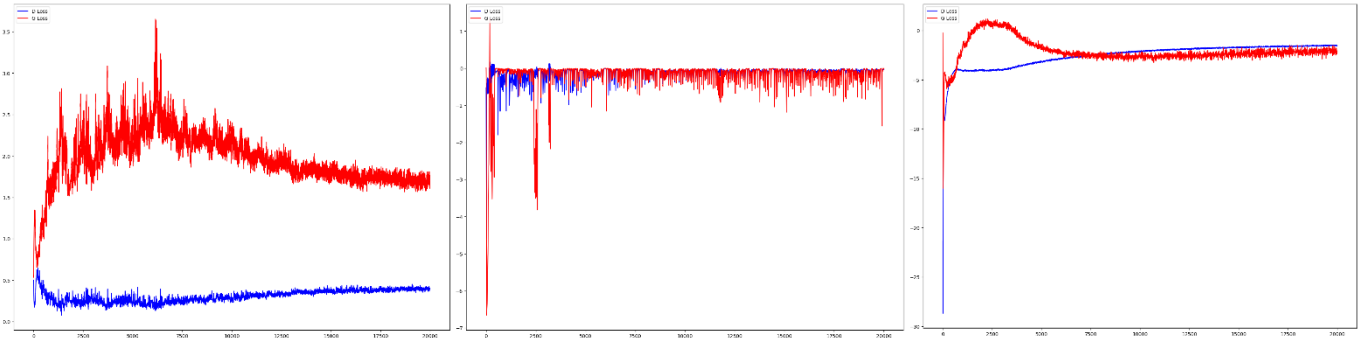


Figure 6: Loss of discriminator (D) and generator (G) during training for three GAN models.
Left: Standard GAN; Middle: Wasserstein GAN with weight clipping; Right: Wasserstein GAN with gradient penalty