KU LEUVEN

LEUVEN STATISTICS RESEARCH CENTRE

Data Mining and Neural Networks

Aug 2021

Student: Shiqi Wang (r0815604)

Assignment 1: Training Algorithms and Generalization

1.1 THE PERCEPTRON AND BEYOND

1.1.1 1) How can you perform a linear regression with a perceptron? 2) Describe the link between those two models.

1) A perceptron can be used for regression if no nonlinear activation function is applied to the sum of the dot product of the input features and weights.
2) Firstly, the parameters of both models are found by minimizing their cost functions. In linear regression, the parameters w and β are found by minimizing the sum of squared errors using the least squares method. In a perceptron, the weights are updated in each iteration by using the perceptron rule until the cost function is minimized to zero, which indicates that all the data is classified correctly. It is important to note that the perceptron algorithm will converge and the cost function will be zero if the data is linearly separable, while the cost function of linear regression is usually not equal to zero unless all the data exactly locates on the fitted line.
   Secondly, the objectives of the linear regression and perceptron are different. The linear regression model tries to model the relationship between the dependent variable and the independent variables. In other words, the linear regression model tries to find the best line to fit the data points. In contrast, the perceptron is a binary/linear classifier, which aims to classify the input. However, when there is no nonlinear activation function in the perceptron, it can also perform as a linear regression.

1.1.2 Plot the function. 1) Will a linear model adequately capture the relationship between the input and output data? 2) Can you relate this to over- or underfitting?

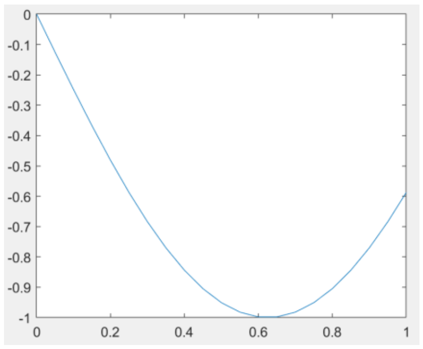


Figure 1.1

1) From Figure 1.1 we can see that the shape of the function is parabolic, so a linear model cannot adequately capture the relationship between the input and output data.
2) A linear model may underfit the data as there is an obvious parabolic trend in the plot.

1.1.3 Train a neural network with one hidden layer containing two neurons on the data-set of the previous question.

It is found from the training results that the fitnet() function randomly divides the data into 3 sets (training set, validation set and test), trains the neural network with Levenberg-Marquardt method and the measures the performance using mean squared error by default. Besides, a nonlinear transfer function tansig is used in the hidden layer and a linear transfer function purelin is used in the output layer. After training the neural network for several times, it is found that the result differs each time. This is because the initial weights and biases of the network is different every time. The regression result shows the model has a perfect fit with R=1 and all datapoints lying on the 45-degree line in the regression plots for training set, validation set and test.

1.2 BACKPROPAGATION IN FEEDFORWARD MULTI-LAYER NETWORKS

1.2.1 Backpropagation

For a shallow neural network with two layers (neural network with one hidden layer), the equations of the general delta rule are shown below.

$$\Delta w_{ij}^2 = \eta\delta_{i,p}^2 x_{j,p}^1 \qquad \Delta w_{ij}^1 = \eta\delta_{i,p}^1 x_{j,p}^0 \qquad \delta_{i,p}^2 = \left(x_{i,p}^{desired} - x_{i,p}^2\right)\sigma'\left(\xi_{i,p}^2\right) \qquad \delta_{i,p}^1 = \left(\sum\nolimits_{r=1}^{1} \delta_{r,p}^2 w_{rj}^2\right)\sigma'\left(\xi_{i,p}^1\right)$$

Feedforward pass for the first datapoint (x_1, x_2, y)=(1,1,2):

$$\xi_{1,1}^1 = \sum\nolimits_{j=1}^{3} w_{1j}^1 x_{j,1}^0 = w_{11}^1 x_{1,1}^0 + w_{12}^1 x_{2,1}^0 + w_{13}^1 x_{3,1}^0 = 1 + 3x_{2,1}^0 + 5x_{3,1}^0 = 9 \qquad x_{1,1}^1 = \sigma\left(\xi_{1,1}^1\right) = \frac{1}{1+\exp\left(-\xi_{1,1}^1\right)} \approx 1$$

$$\xi_{2,1}^1 = \sum\nolimits_{j=1}^{3} w_{2j}^1 x_{j,1}^0 = w_{21}^1 x_{1,1}^0 + w_{22}^1 x_{2,1}^0 + w_{23}^1 x_{3,1}^0 = 2 + 4x_{2,1}^0 + 6x_{3,1}^0 = 12 \qquad x_{2,1}^1 = \sigma\left(\xi_{2,1}^1\right) = \frac{1}{1+\exp\left(-\xi_{2,1}^1\right)} \approx 1$$

$$\xi_{1,1}^2 = \sum\nolimits_{j=1}^{3} w_{1j}^2 x_{j,1}^1 = w_{11}^2 x_{1,1}^1 + w_{12}^2 x_{2,1}^1 + w_{13}^2 x_{3,1}^1 = 7 + 8x_{2,1}^1 + 9x_{3,1}^1 = 24 \qquad x_{1,1}^2 = \sigma\left(\xi_{1,1}^2\right) = \frac{1}{1+\exp\left(-\xi_{i,1}^2\right)} \approx 1$$

Backpropagate pass for the first datapoint (x_1, x_2, y)=(1,1,2):

$$\sigma'\left(\xi_{1,1}^2\right) = \sigma\left(\xi_{1,1}^2\right)\left(1-\sigma\left(\xi_{1,1}^2\right)\right) = 0 \Rightarrow \delta_{1,1}^2 = \left(x_{1,1}^{desired} - x_{1,p}^2\right)\sigma'\left(\xi_{1,1}^2\right) = 0 \Rightarrow \Delta w_{1j}^2 = \eta\delta_{1,1}^2 x_{1,1}^1 = 0 \;\; for\ j = 1,2,3$$

$$\sigma'\left(\xi_{1,1}^1\right) = \sigma\left(\xi_{1,1}^1\right)\left(1-\sigma\left(\xi_{1,1}^1\right)\right) = 0 \Rightarrow \delta_{1,1}^1 = \left(\sum\nolimits_{r=1}^{1} \delta_{r,1}^2 w_{rj}^2\right)\sigma'\left(\xi_{1,1}^1\right) = \delta_{1,1}^2 w_{1j}^2 \sigma'\left(\xi_{1,1}^1\right) = 0 \Rightarrow \Delta w_{1j}^1 = \eta\delta_{1,1}^1 x_{1,1}^0 = 0 \;\; for\ j = 1,2,3$$

$$\sigma'\left(\xi_{2,1}^1\right) = \sigma\left(\xi_{2,1}^1\right)\left(1-\sigma\left(\xi_{2,1}^1\right)\right) = 0 \Rightarrow \delta_{2,1}^1 = \left(\sum\nolimits_{r=1}^{1} \delta_{r,1}^2 w_{rj}^2\right)\sigma'\left(\xi_{2,1}^1\right) = \delta_{1,1}^2 w_{1j}^2 \sigma'\left(\xi_{2,1}^1\right) = 0 \Rightarrow \Delta w_{2j}^1 = \eta\delta_{2,1}^1 x_{1,1}^0 = 0 \;\; for\ j = 1,2,3$$

We can see that the gradients will be zero and the weights will stay the same as before.

Similarly, we can derive the computed outputs for datapoints (2,-1,4) and (3,0,3), which are 22.88 and 24. By applying backpropagation algorithm, it can be derived that the gradients for datapoints (2,-1,4) and (3,0,3) are also zero.

1.2.2 1) How does gradient descent perform compared to other training algorithms? 2) What are the advantages or disadvantages of the algorithms you compare? 3)What is the influence of noise of the dataset size? 4) Discuss the difference between epochs and timing to assess the speed of the algorithms.

1) Due to local minima, each training produces different results. To compare the general performance of gradient descent and other algorithms, 20 trainings for each algorithm were conducted. For a shallow neural network with 35 neurons in the hidden layer and epoch set to 250 and noise set to 0, it is found that the rank of the average training time is trainbfg(1.18s) > trainbr(1.07s) > traingd(0.98s) > trainlm(0.52s) > traingda(0.44s) and the rank of the average correlation between targets and outputs is trainlm(1) > trainbfg(0.98) > trainbr(0.95) > traingda(0.74) > traingd(0.24).

   Figure 1.2 shows one of the fit plots for each algorithm. From the results of the training time and fit plots we can see that gradient descent with adaptive learning rate takes shortest time to train the model but the fit is not very good. The batch gradient descent algorithm holds a moderate training time and fits the data poorly. In general, the second-order methods have better fitting abilities than the first-order gradient descent methods.

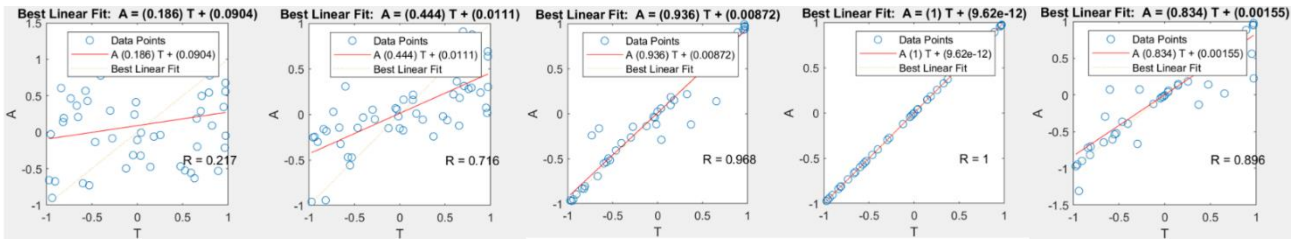


Figure 1.2 The fit plots between model output (A) and target value (T) using different

algorithms. Algorithms from left to right: traingd, traingda, trainbfg, trainlm, trainbr

2) In theory, the batch gradient algorithm (traingd) produces a rather stable gradient descent convergence and error gradient than the gradient descent algorithm (traingda) with adaptive learning rate because the batch gradient algorithm calculates the average of all training samples instead of the value of a single sample. However, it can take a long time to process all the training samples as a batch, which is also found in the average training time results from the previous question. The average training time for traingd is more than twice that of traingda.

   The gradient descent with adaptive learning rate algorithm (traingda) computes the gradient using a single sample at a time, which is computationally fast and is easier to fit into memory. However, single samples are really noisy, which can cause oscillations and lead the gradient descent into other directions.

   The BFGS Quasi Newton algorithm (trainbfg) uses gradient information to approximate Hessian matrix and results into a superlinear speed of convergence. However, the lack of precision in the Hessian calculation can lead to slower convergence in terms of steps, which often occurs for simple problems where the extra computation time to actually compute the Hessian inverse is low. Another disadvantage of this algorithm is that when there are many interconnection weights in the neutral network, it’s hard to store the matrixes into computer memory.

   The Levenberg Marquardt algorithm (trainlm) performed the best in the previous example. It trained the model fast and produced a perfect correlation between the output and target. A potential drawback of this algorithm may be that it converges slower when λ is large.

   The Bayesian regularization algorithm (trainbr) did not show a very big advantage in our example. The fit is reasonably good but the training time is nearly twice that of the Levenberg Marquardt algorithm. But for some noisy and small problems Bayesian Regularization may indeed take longer but obtain a better solution than the Levenberg Marquardt algorithm.

3) By adjusting the noise level in the [0, 0.5] range with other settings held unchanged, it is found that the training time of all algorithms were generally decreasing and the fits were generally increasing when the noise level increased. This phenomenon is reasonable since we have a rather small dataset and adding random noise can make the network less likely to memorize training sample and produce a more robust network with lower generalization error. Besides, adding noise to small datasets can smooth the structure of the input space and thus enable the network to learn the mapping function easier and produce a better and faster learning.

4) The number of epochs defines the number of times that the learning algorithm will work through the entire training dataset. When the number of epochs is small, it usually takes less time for the network to be trained. If the number of the epochs is the same for different algorithms, an algorithm with shorter training time may indicate a faster training speed. However, the feedforward() function in Matlab uses the early stopping technique by default, so an algorithm may not reach the number epochs we set and the training time will be shorter if the training stops early. So, when measuring the training speed of different algorithms, it is necessary to take both epochs and training time into account.

1.3 PERSONAL REGRESSION EXAMPLE

1.3.1 Define your datasets and plot the surface of the training set.

To minimize the chances of having a biased dataset and increase the chances of having a representative dataset for training, the original dataset is first randomized before being divided into training, validation and test set. Then the training set is created by selecting the first 1000 datapoints from the randomized dataset, the validation set is selected as the second 1000 points from the randomized dataset and the test set is produced with the third 1000 datapoints in the randomized dataset. The surface plot of the training set is shown as Figure 1.3.

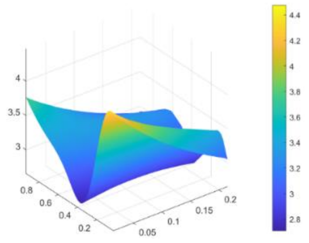


Figure 1.3 Surface of the training set.

1.3.2 Build and train your feedforward neural network.

To better compare with different algorithms, the initial weights were set the same for every algorithm. First, a shallow neural network with two hidden neurons and 1000 epochs was tried. From Figure 1.4 we can see that trainbr has the best fit, followed by trainlm and trainbfg. When the number of hidden neurons was increased, it is found that the RMSE and correlation criteria of all algorithms were generally getting better. When there were 6 neurons in the hidden layer, the correlations of trainlm and trainbr both reached one, which is shown in Figure 1.5. To have a further look at the performance of the two algorithms, the RMSE was computed. The RMSE of trainlm was 2.9029e-04and the RMSE of trainbr was 4.5075e-05, which suggested that trainbr performed better.

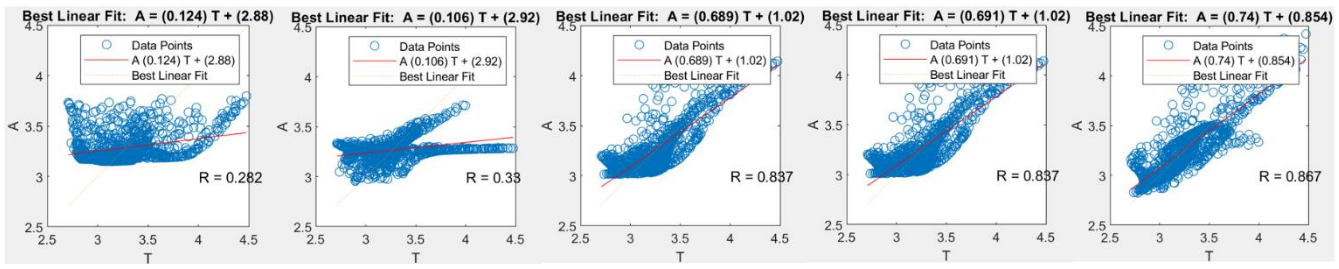


Figure 1.4 A shallow neural network with 2 hidden neurons (tansig activation function). Algorithms from left to right: traingd, traingda, trainbfg, trainlm, trainbr

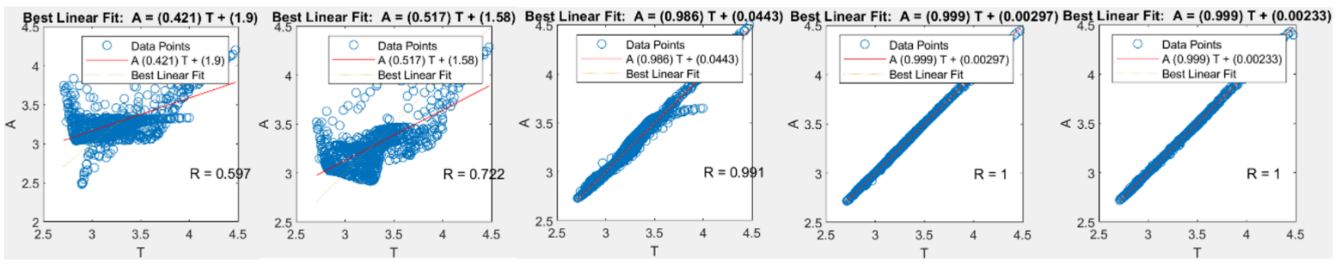


Figure 1.5 A shallow neural network with 6 hidden neurons (tansig transfer function). Algorithms from left to right: traingd, traingda, trainbfg, trainlm, trainbr

If we continue increase the number of hidden neurons, the fit will get better but also takes longer for the network to learn. Since the tainlm and trainbr have already showed great performance at 6 hidden neurons, it would be interesting to explore some other parameters.

The next step was adjusting the transfer function. Apart from the default transfer function tansig, two other transfer functions logsig (Figure 1.6) and softmax (Figure 1.7) were tried. It is found that tansig still produced the best fit compared to logsig and softmax. So, the transfer function tansig will be retained.

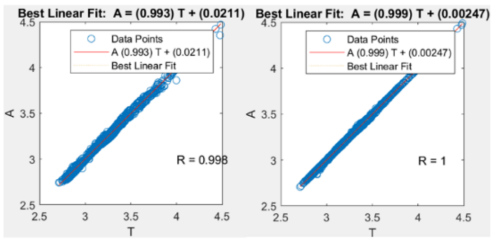


Figure 1.6 trainlm and trainbr with logsig transfer function

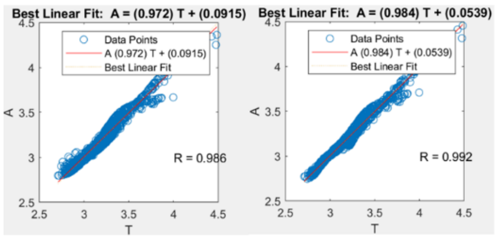


Figure 1.7 trainlm and trainbr with softmax transfer function

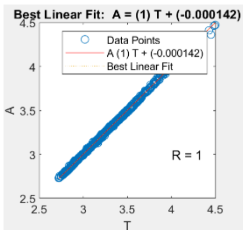


Figure 1.8 trainbr with tansig transfer function on validation set

In the trials of different learning methods and activation function, trainbr showed better performance than other algorithms in general. So, trainbr was chosen as the best algorithm for our example.

Then multilayer neuron networks with two hidden layers were tried. It is found that when there are 4 neurons in first hidden layer and 3 neurons in the second hidden layer, the correlation value of trainlm and trainbr both reached 1. However, compared to the previous neuron network with only one hidden layer, the training time of the network with two hidden layers was longer. For model simplicity and shorter training time, the one hidden layer neuron network with 6 hidden neurons and tansig transfer function was finally chosen.

To validate the model, the network was simulated on the validation set. From Figure 1.8 we can see that there is a perfect correlation between model output and target results, which suggests the neuron network performed well on the validation set.

1.3.2 Performance assessment.

Figure 1.11 shows the correlation between the network approximated result and the target result for test set, which equals 1, showing the network has a perfect fit. By comparing the surface plot of the test set (Figure 1.10) and the network approximated surface plot, we can see the two plots look extremely similar, which also indicates the model has a very good fit.

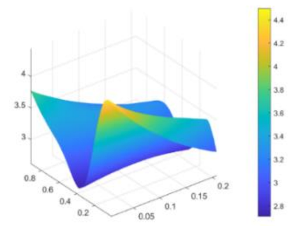


Figure 1.9 Approximated surface plot of the test set

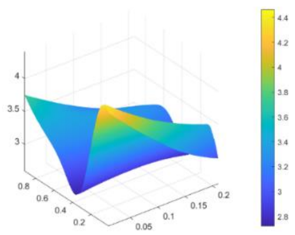


Figure 1.10 Surface plot of test set

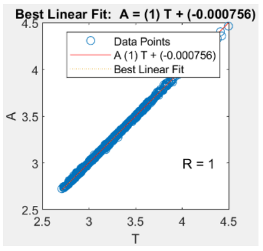


Figure 1.11 Correlation result for test set

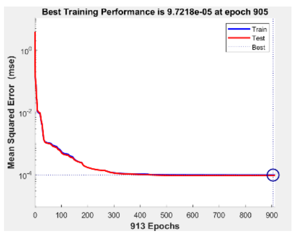


Figure 1.12 Loss curves of selected neural networks

From Figure 1.12 we can see that there are only two loss curves in the plot. It is because the function "trainbr" in matlab that performs Bayesian regularization algorithms disables validation stops by default. The plot also suggests that MSE of the test set is a slightly lower than the MSE of the training set at the end of the epochs, which can also be confirmed by computing the RMSE of the training set (4.5075e-05) and the test set (2.3213e-05). Theoretically, the test error is usually higher than the training error, but the test error can be smaller than the training error in practice when a method generalizes well. To improve the model, it may be sensible to normalize the original data.

1.4 BAYESIAN INFERENCE

1.4.1 Similarly as you did in part 1.2, use trainbr and compare it with other relevant training functions seen. Try with and without noise. Consider overparametrized networks (many neurons): do you see any improvement with trainbr?

Trainlm is a training function which is closely related to trainbr. Trainlm uses Leuvenberg-Marquardt optimization to update the weights and bias. Trainbr uses Leuvenberg-Marquardt optimization as well, but it also uses Bayesian regularization that minimizes a combination of squared errors and weights. To compare the two algorithms, a neuron network with one hidden layer will be constructed to see if noise and neurons have an effect on the performance of the algorithms.

Figure 1.13 show that when the data is noise free and there are 10 neurons in the hidden layer, the fit of trainlm is better than trainbr. When we increase the noise level to 0.3, the correlation of network approximated results and target results for both models increased (Figure 1.14). It can be seen that when there are 10 neurons in the network, trainlm shows better performance than trainbr.

From Figure 1.15 and Figure 1.16 we can see that for an overparameterized network with 45 neurons, the fits of trainlm and trainbr were better than a network with only 10 neurons. However, as no validation set is used here, the overfitting problem should be considered with care. Normally trainbr produces a network which generalizes well, because it uses Bayesian regularization so as to produce a network with a rather stable and robust performance.

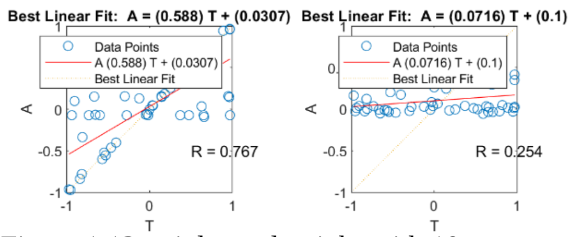


Figure 1.13 trainlm and trainbr with 10 neurons and 0 noise level

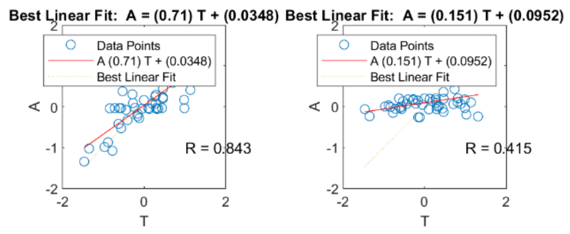


Figure 1.14 trainlm and trainbr with 10 neurons and 0.3 noise level

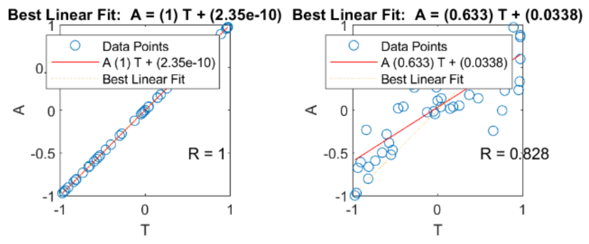


Figure 1.15 trainlm and trainbr with 45 neurons and 0 noise level

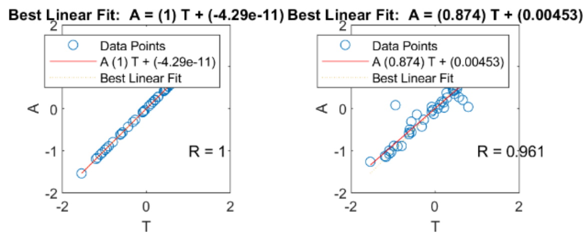


Figure 1.16 trainlm and trainbr with 45 neurons and 0.3 noise level

1.4.2 Compare results before and after regularization.

The biggest difference of trainlm and trainbr was about Bayesian Regularization. Trainbr is an algorithm that uses Leuvenberg-Marquardt method with a regulation term while trainlm only applies Leuvenberg-Marquardt method to update the weights and bias. In our example, after regularization the correlation between the approximated and target results decreased, which is reasonable because here we use all the datapoints to train the network and regularization can prevent the network from overfitting the data when we have a large number of neurons and a high noise level.

Assignment 2: Time-series Prediction and Classification

2.1 TIME-SERIES PREDICTION

2.1.1 (Santa Fe Dataset) Investigate the model performance with different lags and number of neurons so as to select the most appropriate hyperparameters. Note that the performances can be assessed by the mean squared error (MSE) or root-mean-square error (RMSE).

To find out which hyperparameters produces the best network performance, a neural network with one hidden layer is constructed. Different combinations of lags (20, 25, 30, …, 125) and neurons (20, 30, 50) were tried to select the best pair of hyperparameters. Each network with a certain pair of hyperparameters are trained for 10 times to get the average MSE value. As shown in Table 2.1, the neural network with 95 lags and 50 neurons has the smallest average MSE, which means the network performs better than a neural network with other hyperparameters.

Lag	Neuron	MSE	Rank
95	50	1686.5485	1
115	30	1695.2448	2
120	30	2068.9414	3

Table 2.1 Average MSE with different hyperparameters

2.1.2 (Santa Fe Dataset) Would it be sensible to use the performance of this recurrent prediction on the validation set to optimize hyperparameters? If it is, please conduct the hyperparameters selection on the validation set. How is the performance of your feedforward neural network with the chosen hyperparameters on test set?

Logically, it is sensible to use the performance of the recurrent prediction on the validation set to optimize hyperparameters. To conduct that, the training set is split into an 80% sub-training set and a 20% validation set. To select the most appropriate hyperparameters on the validation set, a network of each choice of hyperparameters was iterated 10 times to compute the average MSE. As shown in Table 2.2, 120 lags and 20 neurons are chosen as the best hyperparameters for the validation set.

Lag	Neuron	Avg MSE	Rank
125	20	5427.1824	1
120	20	7381.889	2
110	20	9222.4742	3

Table 2.2 Average MSE with different hyperparameters on validation set

To evaluate the performance of the neural network with the chosen parameters on test set, we can calculate the MSE of test set. Again, the MSE of test set is computed as the average MSE of 10 iterations, which is 3060.1938. Figure 2.1 visualizes the prediction and target value for test set. It seems that the network predicted the first few datapoints pretty well but not for the later datapoints, especially the datapoints after discrete time 60 where the fluctuation shrinks at a sudden. By looking at the training set we can see that there are trends of sudden shrinks around time 200 and 600 and it seems that the network did not capture the shrink pattern very well.

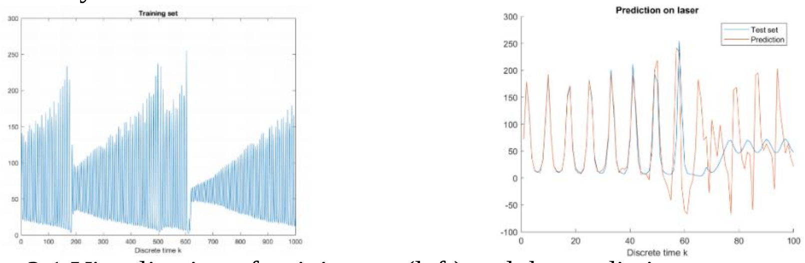


Figure 2.1 Visualization of training set (left) and the prediction on test set (right)

2.1.3 (Climate Change) Build and train your feedforward neural network for the city temperature data set.

The analysis of temperature is conducted on two cities. One is Rio De Janeiro in Brazil, the other is Toyota in Japan. To find out which hyperparameters produce the best network performance, shallow neural networks with different combinations of lags (20, 25, 30, …, 125) and neurons (30, 50, 70) were tried and iterated for 5 times each. The MSE of neural networks with different hyperparameters are shown in Table 2.3 and Table 2.4. For temperatures in Rio De Janeiro from 1870 to 2012, a neural network with 125 lags and 50 neurons produces the smallest MSE, which means the network has the best performance compared to networks with other hyperparameters. Similarly, a neural network with 125 lags and 30 neurons performed best for temperatures in Toyota from 1870 to 2012.

Lag	Neuron	Avg MSE	Rank
125	50	3.735714	1
125	30	4.304293	2
125	70	4.518545	3

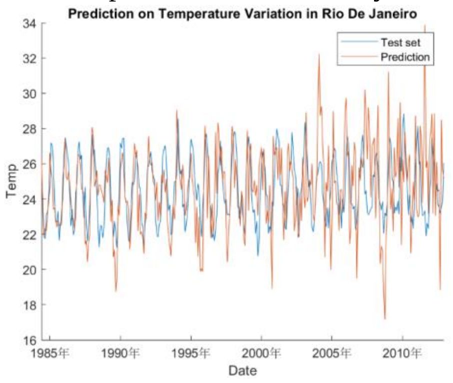
Table 2.3 MSE of networks with different hyperparameters for Rio De Janeiro

Lag	Neuron	Avg MSE	Rank
125	30	215.6991	1
120	30	225.0933	2
105	50	237.7543	3

Table 2.4 MSE of networks with different hyperparameters for Toyota

2.1.4 (Climate Change) Performance Assessment: evaluate the performance of your selected network on the test set. Please refer to global_temperature_test.m. What else could you do to improve the performance of your network?

To evaluate the performance of the selected networks. The MSE of test set for Rio De Janeiro and Toyota are computed, which are 4.956229 and 216.148432. The prediction of temperatures in Rio De Janeiro and Toyota is shown in Figure 2.2. Both of the two networks seem to fit the test set pretty well at the beginning, but less satisfactory for the last few datapoints. From the choice of hyperparameters we can see that, 125 lags are suggested to choose for both networks, which are the maximum lags in the lag selection range. So, it may be realistic to increase the number of lags to improve the performance of the networks. A deeper neural network may also be tried to see if it contributes to the network performance.



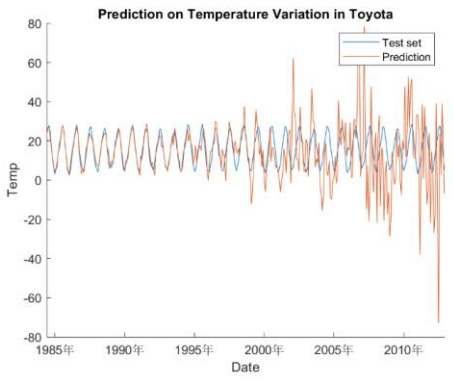


Figure 2.2 Prediction on temperature variation in Rio De Janeiro (left) and Toyota (right)

2.1.5 (Climate Change) What conclusion can you derive based on the prediction results of different cities?

Different cities have different temperature variation. A city with smaller temperature variation produces lower MSE for both validation and test set. In contrast, a city with higher temperature variation produces higher MSE for both validation and test set. In other words, when there are more variation and more complex pattern in the data, it is harder for the network to learn.

2.2 CLASSIFICATION

2.2.1 Visualize the breast cancer data set.

The visualization of the training set is conducted by implementing t-SNE in Matlab. The tsne function in Matlab returns a matrix of two-dimensional embeddings of the high-dimensional rows of the dataset. Then the two-dimensional result is plotted and colored with their label values, as shown in Figure 2.3.

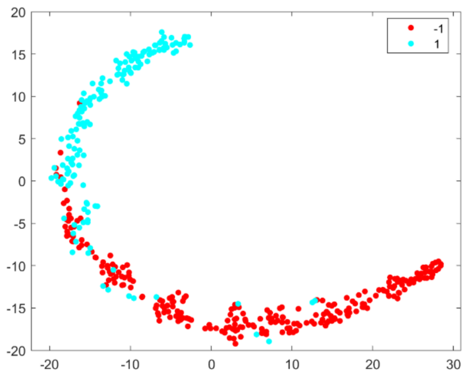


Figure 2.3 Breast cancer training set visualization

2.2.2 Train an MLP on the breast.mat with tuned hyperparameters

For breast.mat, we have a training set and test set. The training set will be used to conduct 5-fold cross validation to see model accuracy under different hyperparameter sets, and the test set will be further used to see how the network performs on it.

There are 30 variables in the dataset. Before training the network, the data is first normalized to make sure they have the same scale. By tuning the hyperparameters, it is found that neural networks with one hidden layer can already provide really good results. Different training algorithms were tried with maximum 1000 epochs. Table 2.5 shows the cross-validation accuracy for certain training algorithms with specific number of neurons in the hidden layer. It turns out that trainbfg with 7 neurons can result in 99.25% accuracy, which also resulted in a 95.86% classification accuracy in the test set.

	1	3	5	7	9
trainbfg	98.50%	98.75%	98.25%	**99.25%**	98.00%
trainbr	95.75%	96.50%	97.50%	97.00%	97.00%
traingd	96.00%	97.50%	97.25%	97.75%	96.50%
traingda	97.00%	98.00%	97.50%	98.25%	98.25%
trainlm	98.75%	98.75%	98.75%	98.25%	98.50%

Table 2.5 5-fold cross validation accuracy for different algorithms with different number of neurons in the hidden layer

2.3 AUTOMATIC RELEVANCE DETERMINATION

2.3.1 Script demard.m, a toy example of a three-dimensional input selection task.

In the script demard.m, a synthetic data set is set up with three input variables x1, x2 and x3, where x1 is designed to be very relevant for determining the target value, x2 to be of some relevance and x3 to be irrelevant in principle. The prior over weights is given by the ARD Gaussian prior with a separate hyper-parameter for the group of weights associated with each input. Then a multi-layer perceptron is trained on this data, with re-estimation of the hyper-parameters using EVIDENCE. The final values for the hyperparameters reflect the relative importance of the three inputs. Table 2.6 shows the hyperparameter values for each input variable, which corresponds to the inverse variance for each input variable. So, we see that the posterior variance for weights associated with input x1 is large, that of x2 has an intermediate value and the variance of weights associated with x3 is small.

Hyperparameter	Value
alpha1	0.17949
alpha2	32.95388
alpha3	311177.40683

Table 2.6 Estimated alpha values

2.3.2 Implementing demard.m on breast.mat, which inputs are most relevant? Now, extract the input variables of breast.mat that are most relevant, train a MLP once again for classification on this data set and evaluate the classification results on the test set. How is its performance when compared with the results of breast.mat on test data in Section 2.2?

Table 2.7 shows 5 inputs with the smallest alphas when implementing ARD on breast.mat. The alphas are the hyperparameters for the corresponding input to hidden unit weights. Since each alpha corresponds to an inverse variance, we can see that the posterior variance for weights associated with input x10, x15, x18, x19 and x20 are rather large (alpha<1), which means these 5 inputs are most relevant.

Hyperparameter	Value
alpha10	0.56190
alpha15	0.16472
alpha18	0.68502
alpha19	0.55741
alpha20	0.69199

Table 2.7 Top 5 smallest alpha values

Then the MLP in Section 2.2 is trained with the 5 selected inputs variable, which resulted in an 82.00% classification accuracy on the training set and a 75.15% classification accuracy on the test set. Compared with the result in Section 2.2, we can see that the classification accuracy using the 5 selected input variables with ARD is lower than the classification accuracy with all input variables, which makes sense. However, in Section 2.2 all the 30 input variables are used to train the network and the accuracy for the test set is 95.86%, but here only 5 input variables are used to train the network and the accuracy for the test set is 75.15%, which suggest the 5 variables are quite useful in determining the target value.

Assignment 3: Unsupervised Learning and Data Visualization

3.1 SELF-ORGANIZING MAP

3.1.1 (Banana Dataset). Check how the prototypes distribute in the feature space before training. Then execute training and observe how the competitive rule determines the final placement according to the data distribution.

Figure 3.1 shows the prototype distribute in the feature space before training, in which the SOM is initialized with hextop as topology function, linkdist as distance function and grid size 10x10. Each neuron (red dots) in the map space is associated with a weight vector. When a training example is fed to the network, its distance to all weight vectors is computed. The neuron with the shortest distance is chosen as the winner and then the weight vector of this winner neuron and its neighbor neurons will be updated to further shorten their distances from the training sample. Figure 3.2 shows the final placement of the neurons where many neurons end up in the green area.

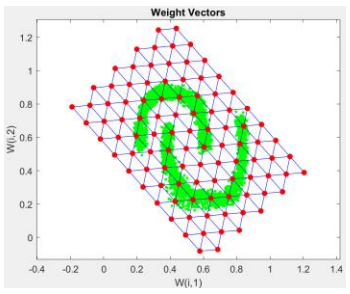


Figure 3.1 Neuron placement before training

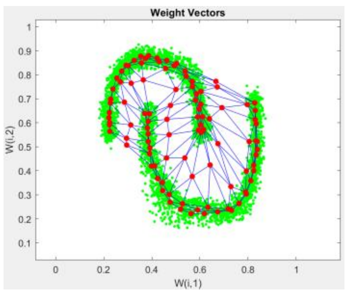


Figure 3.2 Neuron placement after training

3.1.2 (Covertype Dataset) Try a few different values for the grid size, topology and number of epochs. Evaluate performance.

Table 3.1 shows that Covtype is an imbalanced dataset, class 1 and class 2 contain 85.22% datapoints of the whole dataset, while the other 5 classes only contain less than 15% datapoints in total. ARI and RI are both criteria for measure of the similarity between two data clusterings. Higher ARI and RI imply higher similarities. However, ARI is not a suitable criterion for evaluating the performance of SOM when a dataset is imbalanced, so the RI criterion will be used instead.

Class	Count	Percent
1	211840	36.46%
2	283301	48.76%
3	35754	6.15%
4	2747	0.47%
5	9493	1.63%
6	17367	2.99%
7	20510	3.53%

Table 3.1 Numbers and percentage of samples for each class

Several SOM with different epochs, grid sizes and topology functions are tried. Table 3.2 shows the RI of SOM with different grid sizes when epochs is 200 and topology is hextop. We can see that the RI keeps increasing when the grid size becomes larger. Theoretically, a higher RI suggest a better classification result. However, since the grid size of SOM should be similar as the number of classes of the data set, the 3x3 grid size is already big enough in this case, and the RI of the 3x3 SOM is 0.5957.

Grid-size of SOM	1x2	2x2	2x3	3x3	3x4	4x4
Rand Index	0.4911	0.5588	0.5792	0.5957	0.6021	0.6068

Table 3.2 Rand Index for SOM with different grid size

According to SOM Neighbor Distances, one can perform clustering by the colors. A darker color suggests a larger distance between two neurons and a lighter color suggests a shorter distance between them. Figure 3.3 visualize the neighbor distances for the SOM with 3x3 grid size.

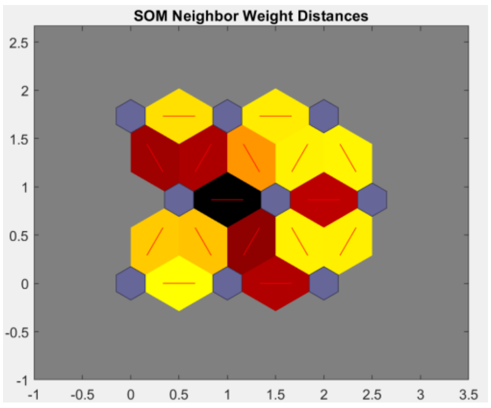


Figure 3.3 SOM neighbor distances visualization

3.2 Principal Component Analysis

3.2.1 Generate a 50 × 500 matrix of Gaussian random numbers (randn(50,500)) and try to reduce dimensions with PCA (interpret this as 500 datapoints of dimension 50). Examine different reduced datasets for different dimensions. Try to reconstruct the original matrix. Estimate the error, e.g. by calculating the root mean square difference between the reconstructed and the original data (sqrt(mean(mean((x-xhat).^2)))).

Table 3.3 shows the proportion of variance that are explained by choosing different number of principal components, or reducing data to different dimensions. We can see that more principal components explain more variance and the 35 highest principal components can explain 80.95% total variance. By reconstructing the original matrix, it is found that when more principal components are used, the RMSE is smaller. When the dimension was reduced to 35, the RMSE was 0.44.

Numbers of Components	Proportion of variance explained
20	53.19%
25	63.43%
30	72.66%
35	80.94%

Table 3.3 Variance explained proportion with different number of components with random data

3.2.2 Do the same using the data file choles_all (standard in Matlab, can be loaded with load ...choles_all). For the exercise, use only the p component, which is a 21 × 264 matrix (so the dimension of the data equals 21). How does the reduction of random data compare to the reduction of highly correlated data?

Compared with the reduction of random data, the highly correlated data can be reduced into a really small dimension but with high variance explained. In 3.2.1, the 35 principal components explained 80.94% total variance, while in this correlated dataset, the first principal components can already explain 95.44% total variance and two principal components can explain 99.74% total variance (Table 3.4). When the dimension was reduced to 1, the RMSE is 0.02.

Numbers of Components	Proportion of variance explained
1	95.44%
2	99.74%
3	99.88%
4	99.95%

Table 3.4 Variance explained proportion with different number of components with correlated data

3.2.3 Now experiment with the functions mapstd(x) and processpca(x,maxfrac) (try for example maxfrac = 0.001), and see whether you can obtain similar results.

For the random data set in 3.2.1, no record was deleted by using the processpca(x,0.001) function, which means every row in the data matrix contributes at least 0.001 to the total variation and no data was deleted. The RMSE of the reconstructed data and original data is nearly 0 which means the reconstructed data is very close to the original data. It is reasonable because the dimension was not reduced and all principal components were retained when using processpca.

For the correlated dataset choles_all, the dimensions were reduced from 21 to 4, which means the 4 principal components can explain a large proportion of the total variation and each of other 17 principal components can only contribute less than 0.001 to the total variation. The RMSE of the reconstructed data and original data is 0.03, which is close to the RMSE we get in 3.2.2 when we use the first 4 principal components.

3.3 AUTOENCODER

Image reconstruction on synthetic handwritten digits dataset using an autoencoder.

To have an understanding of the appropriate epochs for training, an autoencoder with 50 neurons in one hidden layer is first trained with different number of epochs. Table 3.5 shows the reconstructing MSE. When epochs reached 350, the decrease of MSE is rather small ($< 10^{-4}$) when epochs kept increasing.

Epoch	50	100	150	200	250	300	350	400	450
MSE	0.0379	0.0244	0.222	0.0216	0.0213	0.0212	0.0211	0.0211	0.0211

Table 3.5 MSE of neural networks with 50 neurons and different epochs

Figure 3.4 shows the actual image and the reconstructed image produced by an autoencoder with 50 neurons and 350 epochs. Compared with the actual image, the reconstructed image was rather vague.

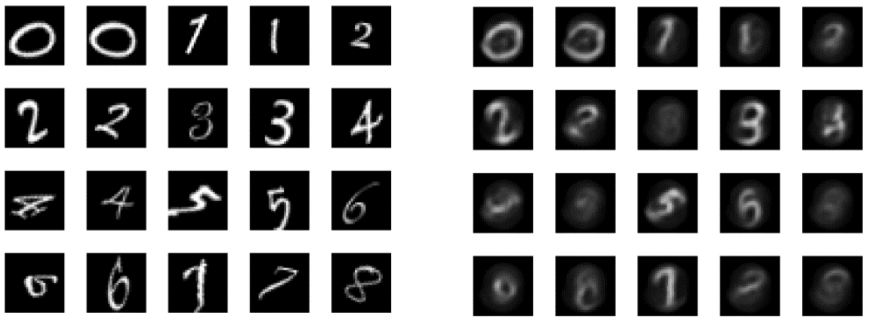


Figure 3.4 Actual image (left) and reconstructed image (right)

Then more neurons are tried to have a clearer reconstructed image. Table 3.6 shows the reconstructing MSE of autoencoders with different neurons and 350 epochs. The MSE kept decreasing with the neurons were increased. When neurons increase to 500, the decrease of MSE is rather small and we cannot see a big difference in the reconstructed images produced by 500 neurons and 700 neurons (Figure 3.5). So, an autoencoder with 500 neurons and 350 epochs is final selected. However, more combinations of epochs and neurons can be tried if we want to further improve the reconstructed results.

Neurons	100	200	300	400	500	600	700
MSE	0.0172	0.0146	0.0135	0.0129	0.0126	0.0124	0.0122

Table 3.6 MSE of neural networks with 350 epochs and different neurons

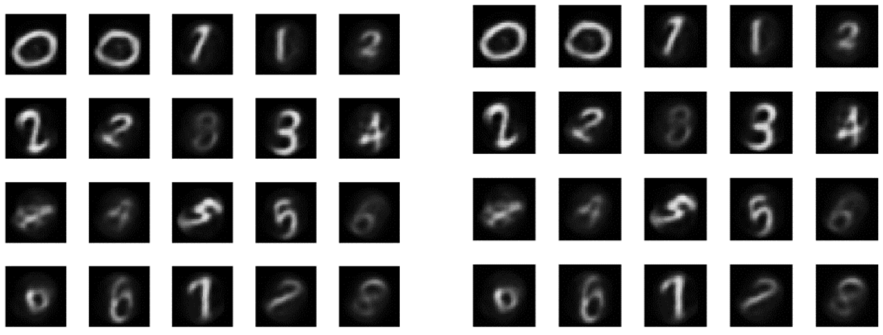


Figure 3.5 Reconstructed image with 500 neurons (left)
and reconstructed image with 700 neurons (right)

3.4 STACKED AUTOENCODER

3.4.1 Are you able to obtain a better result with different parameters (wrt to the default ones)? 2) How many layers do you need to achieve a better performance than with a normal neural network? 3) What can you tell about the effect of finetuning?

1) In the script DigitClassification.m, the default neurons for the stacked autoencoder are 100 (first layer), 50 (second layer) and 10 (softmax layer). And the default epochs are 400 (first layer), 100 (second layer) and 400 (softmax layer). To see if the accuracy can be increased, different combinations of neurons and epochs are tried. Stacked autoencoders with each parameter set are trained three times to get the average accuracy, as shown in Figure 3.7. This default choice of hyperparameters resulted in 83.83% accuracy before finetuning and 99.73% after finetuning. By comparing accuracy results obtained by different hyperparameter sets, it is found that when neurons and epochs increase, the accuracy before fine tuning will increase and the accuracy will increase after finetuning. Especially, the power of finetuning seems very powerful when the number of neurons and epochs are small.

Neurons (first layer)	Neurons (second layer)	Epochs (first layer)	Epochs (second layer)	Epochs (output layer)	Accuracy (before finetuning)	Accuracy (after finetuning)
50	25	200	50	200	27.47%	99.60%
100	50	200	50	200	53.67%	99.73%
200	100	200	50	200	77.67%	99.87%
50	25	400	100	400	44.50%	99.57%
100	50	400	100	400	83.83%	99.73%
200	100	400	100	400	85.60%	**99.80%**
50	25	500	200	500	71.53%	99.33%
100	50	500	200	500	94.90%	98.90%
200	100	500	200	500	98.17%	98.60%

Table 3.7 Accuracy of stacked autoencoder with different hyperparameters

2) To see if normal neural networks can achieve the same or better performance, different layers and hidden neurons are tried. First, normal neural networks with one hidden layer were trained. By trying different number of hidden neurons (10, 20, 30, … , 200), it is found that a neural network with 140 hidden neurons has the highest 97.76% accuracy, which is smaller than the accuracy of autoencoder (99.80%). Then, neural networks with two hidden layers were investigated. By trying different combinations of neurons in each layer (10, 20, 30, … , 200), it was found that when a neural network with 180 neurons in the first hidden layer and 30 neurons in the second hidden layer has the highest accuracy, which was still lower than the accuracy of autoencoder. Neural networks with three hidden layers were also tried. Again, by trying different combinations of neurons in each layer (10, 20, 30, … , 200), a neural network with 140, 140 and 30 neurons produced the highest 98.3%. The result suggests that a three hidden layer network still cannot do not have a better performance than a fine-tuned autoencoder. To achieve a better accuracy than the fine-tuned autoencoder, a normal neural network should at least have more than 3 hidden layers if every hidden layer is constrained to 200 neurons.
3) From accuracy result in Table 3.7, we can see that finetuning is powerful. When experimenting different neurons and epochs, it is found that the accuracy of an autoencoder generally increases after finetuning. Especially when original the autoencoder has a low accuracy, finetuning can increase the accuracy to a large extend. For example, when there are 50 neurons in the first layer and 25 neurons in the second layer with 200 epochs in the first layer, 50 epochs in the second layer and 200 epochs in the output layer, the accuracy are 27.47% and 99.6% before and after finetuning.

4.1 Variational Auto-Encoders and Convolutional Neural Networks

4.1.1 How does the scale of weight initialization affect models with/without batch normalization differently, and why?

From Figure 4.1 we can see that when weight initialization scale is small, the batch normalization performs better with validation accuracy, training accuracy and final training loss. In general, batch normalization is more tolerant with weight initialization scale, it is because batch normalization will normalize the output in each layer, decreases the importance of initial weights and speed up training.

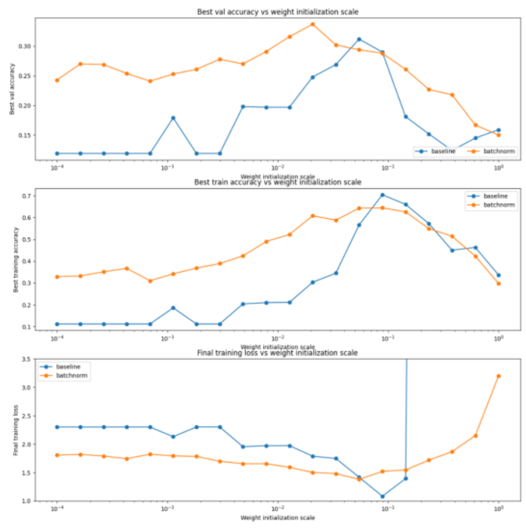


Figure 4.1 Relationship accuracy and weight initialization scale with/without batch normalization

4.1.2 What does this imply about the relationship between batch normalization and batch size? Why is this relationship observed?

Figure 4.2 shows the training and validation accuracy of batch normalization with different batch size in different epochs. Generally, the training accuracy of small batch size are lower. It is reasonable because batch normalization calculates the statistics (e.g. mean and variance) in a batch and the results in smaller batches differ more significantly. With larger batch size, the statistics are be more stable and accuracy is higher. However, the validation results did not show a significant relationship between batch size and validation accuracy. This is because batch normalization uses the pre-calculated running statistics (e.g. running mean and running variance) that are similar to the global statistics of the data.

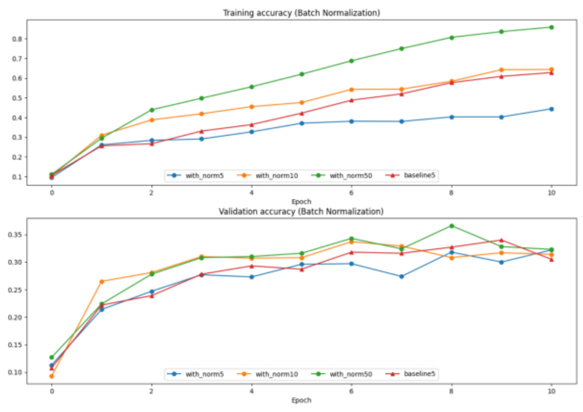


Figure 4.2 Relationship between accuracy and batch size with different batch sizes

4.2 VARIATIONAL AUTOENCODERS

4.2.1 Stacked Autoencoder vs Variational Autoencoder

The similarity between stacked autoencoder and variational autoencoder is that they both contain an encoding and a decoding process.

The difference is that stacked autoencoder tries to reduce dimensions of the original data and then reconstruct the original image (data), but it cannot generate new image (data). In contrast, variational autoencoder can produce new image (data). Besides, instead of mapping input data to a fixed vector in stacked autoencoders, variational autoencoders maps input data to a distribution. In other words, the bottleneck vector in stacked autoencoders is replaced by the mean vector and standard deviation vector of the distribution.

The loss functions are different for stacked autoencoder and variational autoencoder. As a neural network that contains multiple layers of sparse autoencoders, stacked autoencoder tries to minimize the reconstruction loss (the mean squared error). Since the structure of variational autoencoder is similar to stacked autoencoder except it maps the input to a distribution instead of a fixed vector, the variational autoencoder tries to minimize a combination of reconstruction loss and latent loss, in which the reconstruction loss is the expected negative reconstruction error and the latent loss refers to the Kulback-Liebler divergence between the approximate posterior and the prior distribution, which acts as a regularizer in the loss function.

4.2.2 Which optimizer is used in the demo file of stacked autoencoders and in variational autoencoders? Write a short note on each method and compare the pros and cons.

The scaled conjugate gradient method is used in the stacked autoencoder and the Adam opitimizer is used in the variational autoencoder.

The scaled conjugate gradient algorithm is a backpropagation algorithm used for feed forward neural networks and is a member of the class of conjugate gradient methods.

Pros: It can avoid the time-consuming line search that other conjugate gradient methods do. The line search is computationally expensive, since it requires that the network response to all training inputs be computed several times for each search.

Cons: It adjusts the weights in the steepest descent direction (the most negative of the gradient), which is the direction where the loss function decreases most rapidly. However, it has been shown that although the function decreases most rapidly along the negative of the gradient, this does not necessarily produce the fastest convergence.

The Adam optimizer is a method for stochastic optimization and uses momentum and adaptive learning rates to converge faster. Different from classical stochastic gradient descent which keeps the learning rate unchanged for weights update during training, Adam optimizer computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.

Pros: It is effective and can achieve good results fast in the field of deep learning. The Adam optimizer combines the good properties of two other extensions of stochastic gradient descent named Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp), which makes it able to provide an optimization algorithm that can handle sparse gradients on noisy problems.

Cons: In some areas, Adam does not converge to an optimal solution. It has been shown that for some tasks like the image classification on popular CIFAR datasets, the best results are still only achieved by applying SGD with momentum.

4.3 CONVOLUTIONAL NEURAL NETWORKS

4.3.1 Toy-example.

i) Instead of multiplying the kernel with input vector iteratively, the convolution operation could be written as matrix multiplication. In our case, y is a 4x1 matrix, A is a 4x6 matrix and x is a 6x1 matrix. When y and x is known, the Toeplitz matrix A can be constructed, as shown in Figure 4.3.

$$\overset{y}{\begin{bmatrix} 6 \\ 6 \\ 7 \\ 8 \end{bmatrix}} = \overset{A}{\begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}} \text{ x } \overset{x}{\begin{bmatrix} 2 \\ 5 \\ 4 \\ 1 \\ 3 \\ 7 \end{bmatrix}}$$

Figure 4.3 Convolution operation as matrix multiplication

ii) Simply transposing the matrix A and multiplying with y cannot recover x. As shown in Figure 4.4, z does not equal to x. It is reasonable because matrix A is a singular matrix and cannot be inverted.

$$\overset{\text{A transpose}}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \text{ x } \overset{y}{\begin{bmatrix} 6 \\ 6 \\ 7 \\ 8 \end{bmatrix}} = \overset{z}{\begin{bmatrix} 6 \\ 6 \\ 13 \\ 14 \\ 7 \\ 8 \end{bmatrix}}$$

Figure 4.4 Result of A transpose multiply with y

4.3.2 Script CNNex.m

i) The code size(convnet.Layers(2).Weights) returns 11, 11, 3, 96, which means the first convolutional layer (layer 2) has 96 kernels of size 11x11x3. The weights are represented as the multiplicative factor of the filters. Intuitively, the weight values live inside the filters and the dot product will be calculated between the filter and the receptive field (a local region of the input image with the same size as that filter). The weights are initialized randomly and will be updated through the backpropagation process.

ii) By checking the CNN architecture, it can be confirmed that the size of the input image is 227x227x3 and the kernels are convolved with the input by stride [4 4] and padding [0 0 0 0] in layer 2. Using this information, the size of the feature map can be calculated as [(227+2*0-11)/4+1, (227+2*0-11)/4+1, 96], which is [55 55 96].

Then the third layer is ReLU and the fourth layer is Cross Channel Normalization. Both of the two layers do not affect the dimension of the feature map. So, the output size of fourth layer is still [55 55 96].

The fifth layer is 3x3 max pooling with stride [2 2] and padding [0 0 0 0]. The size of the pooled image can be calculated as [(55+2*0-3)/2+1 (55+2*0-3)/2+1 96], which is [27 27 96] and is also the dimension of input at the start at layer 6.

iii) The number of neurons used for the final classification task is 1000, which is much smaller than the dimension of the original input data (227x227x3).

4.3.3 Script CNNDigits.m

The default CNN architecture in the script produces an 82.0% accuracy on test set. To see if accuracy can be improved, architectures with different layers and dimensions of weights are tried. It turns out that neural network with one convolution layer can already be very powerful. Table 4.1 shows the accuracy for test set with different parameter combinations when there is an image input layer [28 28 1], convolution layer, ReLU layer, max pooling layer, fully connected layer, softmax layer and classification layer. The result shows that wen the convolution size is 5x5 with 20 kernels, max pooling size is 4x4 with 1 stride, the classification accuracy for the test set reached 99.3%.

convolution2dLayer	maxPooling2dLayer	Accuracy
(5,20)	(2,'Stride',2)	95.0%
(5,20)	(3,'Stride',2)	95.5%
(5,20)	(4,'Stride',2)	96.0%
(5,20)	(5,'Stride',2)	95.4%
(5,20)	(2,'Stride',1)	97.0%
(5,20)	(3,'Stride',1)	98.0%
(5,20)	(4,'Stride',1)	**99.3%**
(5,20)	(5,'Stride',1)	98.0%
(5,30)	(2,'Stride',1)	98.0%
(5,30)	(3,'Stride',1)	99.1%
(5,30)	(4,'Stride',1)	98.8%
(5,30)	(5,'Stride',1)	98.7%

Table 4.1 Accuracy of test set under different architectures