

Artificial neural networks - Exercise session 1

Supervised learning and generalization

0685182 Yi Zhao

1. Function Approximation: Comparison of Various Algorithms

In this section, a neural network with one hidden layer was trained to approximate a nonlinear function $y = \sin(x^2)$ for $x = 0:0.05:3\pi$ using different training algorithms.

The training algorithms discussed include:

gd: gradient descent;

gda: gradient descent with adaptive learning rate;

cgf: Fletcher-Reeves conjugate gradient algorithm;

cgp: Polak-Ribiere conjugate gradient algorithm;

bfg: BFGS quasi Newton algorithm (quasi Newton);

lm: Levenberg-Marquardt algorithm (adaptive mixture of quasi Newton and steepest descent algorithms).

The function is noise free and I set the number of neurons 50 and number of training epochs 800, which generally guarantees the network works well. Table 1 lists the meaning of the evaluating indicators.

Table 1: Evaluating indicators

Training Time	Time costed for training the network.
Best Epoch	The epoch when it stops iteration.
Vperf MSE	The best validation performance (MSE) during training.
R	The correlation between targets and results.
MSERT	The MSE between targets and results.
Speed	Best Epoch/Training Time

For each case, 10 repeated simulations were performed to obtain the average of the evaluating indicators, see Table 2.

Table 2: Performance of various training algorithms

Algorithm	gd	gda	cgf	cgp	bfg	lm
Training Time	1.8158	0.4296	0.5184	0.4537	0.7314	0.3849
Best Epoch	721.5	76.4	40.1	33	52.5	8.1
Vperf MSE	0.5320	0.5832	0.1390	0.1493	0.0449	0.0336
R	0.4838	0.3763	0.8907	0.8527	0.9705	0.9859
MSERT	0.4292	0.4990	0.0952	0.1275	0.0275	0.0133
Speed	397.3	177.8	77.35	72.73	71.78	21.04

To observe it more clearly, we can refer to Figure 1.

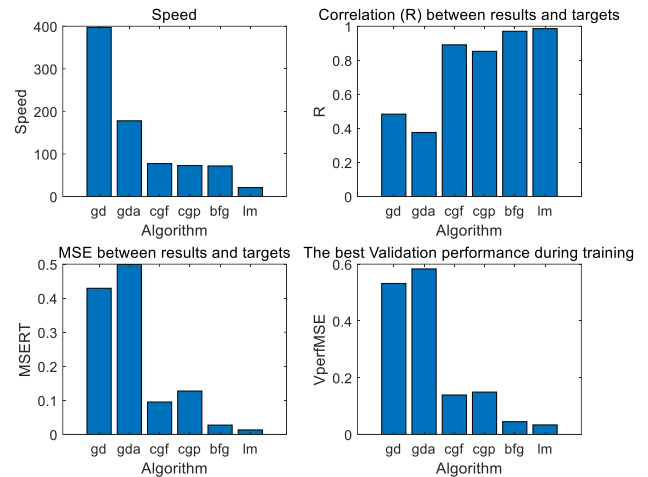


Figure 1 Comparison of various algorithms (when noise free).

From Table 2 and Figure 1, we can get the following conclusions:

- Gradient Descent (gd) is primitive, it has the largest speed (slowest), and it doesn't approximate the target function well.
- Gradient Descent with Adaptive learning rate (gda) lowers the training time, but as for accuracy (MSERT, VperfMSE, R, etc.), it doesn't help much.
- The performances of Fletcher-Reeves Conjugate Gradient algorithm (cgf) and Polak-Ribiere Conjugate Gradient algorithm (cgp) are similar. Both make the approximation better, compared with gd and gda.
- BFGS quasi Newton algorithm (bfg) costs more time than Levenberg-Marquardt algorithm (lm) and lm also has the lowest error rate. The Levenberg-Marquardt algorithm (lm) is the best among these algorithms.

2. Learning from Noisy Data

Now we add noise to the input data and compare the performance of different training algorithms. Originally, the input is $x = 0:0.05:3\pi$. The interval of input datapoints is 0.05, so I made the amplitude of the added noise to be 0.05, too. The input is: $\text{input} = x + 0.05 * \text{rands}(1, \text{length}(x))$ now but the target keeps the same: $\text{target} = y = \sin(x^2)$. The input and the target are used to train the network. After that, the network can be simulated by: $\text{result} = \text{sim}(\text{net}, x)$. It means that the test input is x rather than $(x + \text{noise})$, if the network has good generalization, the result would be close to the target y .

Figure 2 shows the noise free and noise included cases, both with neuron number 50, epoch number 80 and training function trainlm.

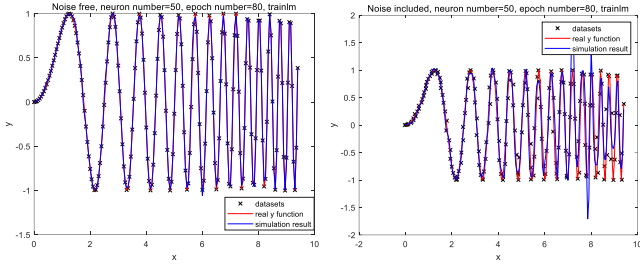


Figure 2: Noise free and noise included cases, neuron number =50, epoch number=80, trainlm

From Figure 2, we can see when it is noise free, the learned model fits the target well, but when it is noise included, the approximation is not good especially when x is large.

3. Bayesian Regularized Network

Bayesian regularized artificial neural networks are more robust than standard back-propagation nets. In Matlab, we use the command `trainbr` to achieve it, adopting Bayesian regularization and updating the weights and bias according to Levenberg-Marquardt algorithm. It minimizes a combination of squared errors and weights, and then determines the correct combination (tradeoff between bias and variance) to produce a network that generalizes well. The regularization parameters are related to the unknown variances associated with these distributions. Using regularization will cause the network to have smaller weights and biases, and this will force the network response to be smoother and less likely to overfit. In what follows, `br` is short for `trainbr`.

Now we compare `br` with other algorithms (`gd`, `cgf`, `lm`) when noise included and the indicating term is test error. Test error is measured by MSERT (the MSE between targets and results) and can describe the generalization ability of a network. The smaller MSERT is, the learned model closer to the target is and the better generalization it has. Similar with Section 1, for each case, 10 repeated simulations were performed to obtain the average of test error (MSERT).

3.1. When number of neurons varies

At first, let's see how the number of neurons affects test error. The number of epochs is set with value of 80. The

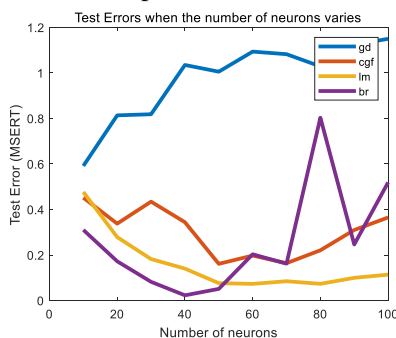


Figure 3: Test errors when the number of neurons varies

number of neurons varies from 10 to 100 with step length 10. Figure 3 shows the simulation result, from which we can see:

- 1) When neuron number is small, performance of `br` is better than others. But when neuron number is large, it doesn't hold.
- 2) Performance of `br` is influenced a lot by neuron number. As neuron number increases, test error first decreases and then increases.

To explain it, in the Bayesian framework, the weights and biases of the network are assumed to be random variables with specified distributions. When neuron number is large, the network would make the distribution more complicated, which leads to overfitting.

3.2. When number of epochs varies

Now we consider how epoch number influences test error. From Figure 3, we can find when the number of neurons is 50, test errors of `cgf`, `lm`, `br` are similar. So, I set the neuron number as 50 in this experiment. The number of epochs varies from 20 to 200 with step length 20. Figure 4 shows the simulation result.

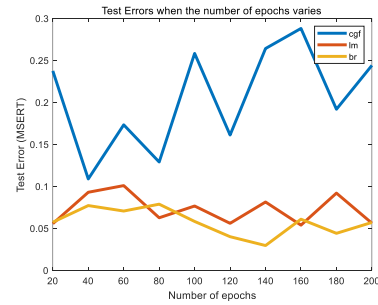


Figure 4: Test errors when the number of epochs varies

From Figure 4, we can see:

The performance (test error) of `br` and `lm` is similar when neuron number is 50 and epoch number varies. We can get the conclusion that epoch number doesn't influence the performance of `br`.

Therefore, when we use `trainbr` to train a network, we can pay less attention on epoch number but focus on neuron number.

Artificial neural networks - Exercise session 2

Supervised learning and generalization

0685182 Yi Zhao

1. Hopfield Network on the Handwritten Digit

Dataset

This section describes how a discrete Hopfield network is working on the handwritten digits dataset. A Hopfield neural network is a dynamical system and consists of a single layer which contains fully connected recurrent neurons. It also requires the data to be ± 1 .

1.1. Attractors

A Hopfield network is initially trained to store several patterns or attractors. During training, weights will be updated (according to the Hebb rule). It is then able to recognize any of the learned patterns by exposure to only partial or even some corrupted information about that pattern, i.e., it eventually settles down and returns the closest pattern or the best guess.

In this case, the attractors are ten handwritten-digit matrices representing 0, ..., 9. Each of the matrices can be shown in the form of an image. See the left column in Figure 1.

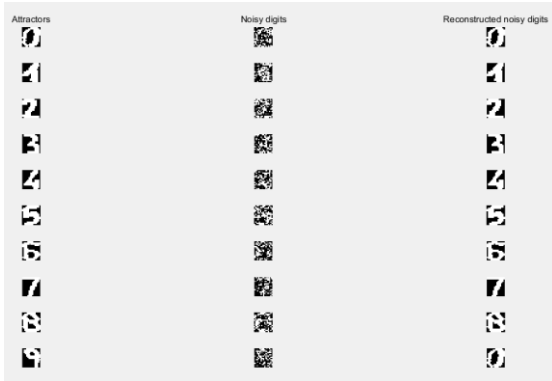


Figure 1 Result of `hopdigit_v2(4,20)` with noise level 4 and iteration times 20.

The middle column represents the input datasets which are corrupted by noise. After training, they converge into corresponding attractors in the right column.

1.2. Storage capacity

The storage capacity p_{max} in a Hopfield network with N nodes can be calculated by:

$$p_{max} = N/4\log N$$

In this case, N equals to 240, and p_{max} is around 10.9. Therefore, the network can work for 10 patterns, though the number of patterns is large.

1.3. Spurious states

When a Hopfield network stores a set of patterns, also additional unwanted patterns (spurious states) are stored:

- 1) for each stored pattern, its negation is also an

attractor.

- 2) unwanted mixture states are stored.
- 3) local minima that are not correlated to any linear combination of stored patterns might cause spurious states.

To get potential spurious states of the network for digit dataset, we first introduce an example. Executing script `rep2` and `rep3`, we can get the results in Figure 2.

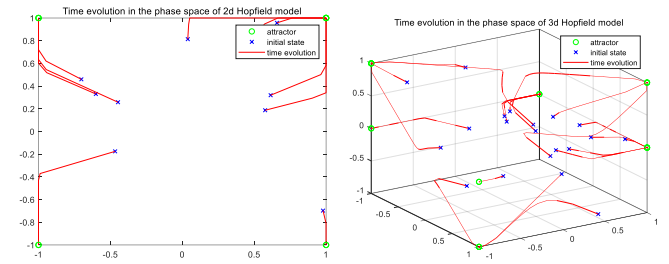


Figure 2: Time evolution in the phase space of 2d and 3d Hopfield model.

For the 2d Hopfield model, the attractors we set are $T = [1 \ 1; -1 \ -1; 1, \ -1]^T$, and the 3d case with $T = [1 \ 1 \ 1; -1 \ -1 \ 1; -1 \ -1 \ -1]^T$. Both models have spurious states after training. The reason might be that the attractors are of high symmetry.

Inspired by the example, we can take a try to see what would happen if the attractors in the Hopfield network on the handwritten digit dataset are symmetry. For example,




replacing the attractor  by  (the negation of ) to make two of the attractors symmetry, and running the command `hopdigit_v2(10,300)`, we might get new reconstructed noisy digits which are not the attractors we set, as is shown in Figure 3.



Figure 3: Spurious states caused by symmetry attractors.

1.4. Other conclusions

In this part, we analyze the influence of the two parameters noiselevel and num_iter.

- To make the Hopfield network converge into stable states, when the value of noiselevel is large, we need to set the value of num_iter large, too.
- Larger noiselevel leads to larger classification error. This is because after adding noise, a data point might be closer to another attractor than to its real attractor.

2. Elman Network to the Hammerstein Data

In this section, we analyze the experimental results related to the application of the Elman network to the Hammerstein system data concerning different time-series, number of training examples, learning times (number of epochs) and architectures. When all the parameters are set, we obtained the average R value and average MSE based on 20 repeated simulations.

2.1. Different time-series

The Hammerstein system with state-space can be described

$$\text{by } \begin{cases} x(t+1) = Ax(t) + \sin(u(t+1)) \\ y(t+1) = x(t+1) \end{cases}, \text{ where } u(t) \text{ is}$$

a stochastic input, $y(t)$ is the output and $x(t)$ is the state of the system.

Different values of the coefficient A among [0:0.1:1] were used to form different time series, as is shown in Figure 4.

When A equals to 1, there will be a huge change in both and MSE. It might mean the system would be totally unstable when A is 1. So, we discard it and analyze the cases when A is among [0:0.1:0.9].

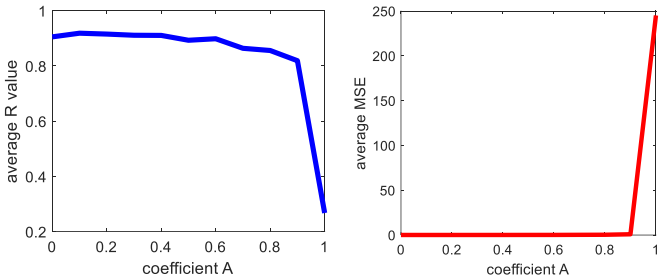


Figure 4: R and MSE when coefficient A varies among [0:1]

From Figure 5, we can conclude that generally when A increases, R value decreases and MSE increases. The reason might be that when A is small, the system try to simulate the function $y = x = \sin(u)$ which is not complicated.

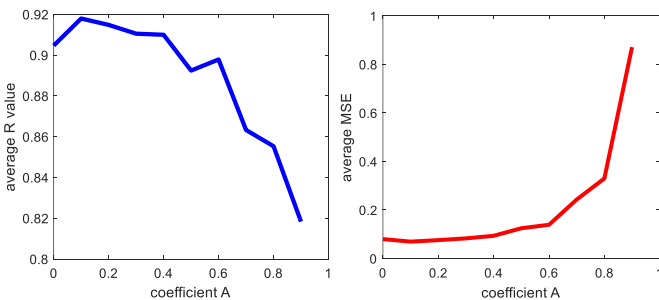


Figure 5: R and MSE when coefficient A varies among [0:0.9]

2.2. Different number of training examples

When the total number of training and validation points is 300 (in the script elmants2), the larger the percentage of training examples is, the better the performance will be. Nevertheless, if the total number of training and validation points is very large, the situation might be different. Overfitting might happen when the percentage is high.

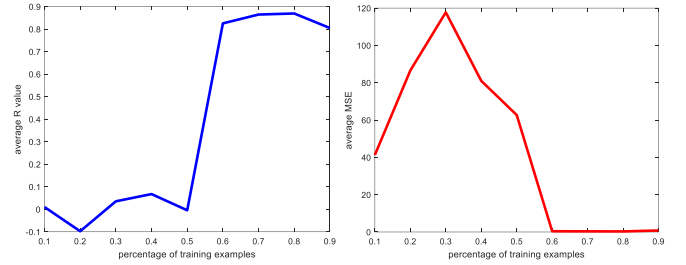


Figure 6: R and MSE when percentage of training examples varies among [0:1]

2.3. Different number of epochs

It can be seen from Figure 7 that when learning time (number of epochs) increases, the performance of the Elman network would be better with R increasing and MSE decreasing.

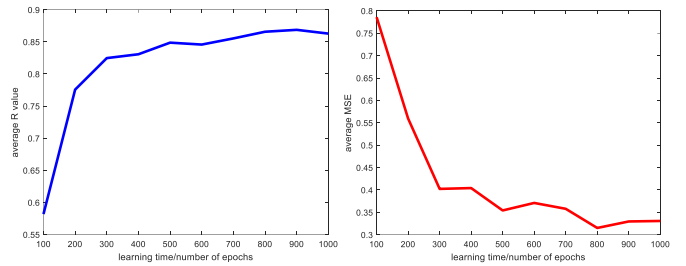


Figure 7: R and MSE when number of epochs varies among [100:100:1000]

2.4. Different architectures

From Figure 8, we can see that there isn't a pattern how different architectures (the number of neurons) influence the network performance. But in this case (the script elmants2), when other parameters are set, the Elman network would have the best performance when the number of neurons is 30. The average R value is 0.8848 and the average MSE is 0.2637.

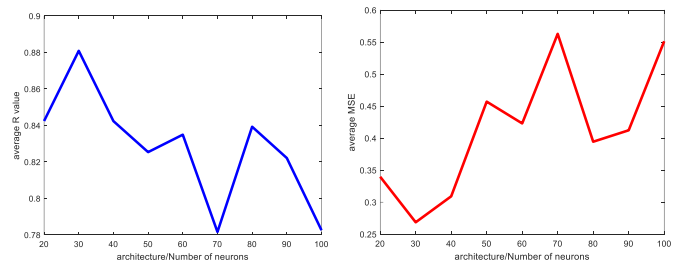


Figure 8: R and MSE when number of neurons varies among [20:10:100]

Artificial neural networks - Exercise session 3

Unsupervised learning: PCA and SOM

0685182 Yi Zhao

1 Handwritten Digits PCA and reconstruction

Principal Component Analysis (PCA) involves projecting onto the eigenvectors of the covariance matrix. The basic idea behind PCA is to map a vector $x = (x_1, x_2, \dots, x_p)$ of a p dimensional space to a lower-dimensional vector $z = (z_1, z_2, \dots, z_q)$ in a q dimensional space (where $q < p$). Now let's perform PCA on handwritten images of the digit 3 taken from the US Postal Service database. At first, a megabyte matrix called threes was loaded. Each line of this matrix is a single 16 by 16 image of a handwritten 3 that has been expanded out into a 256 long vector.

- 1) Compute the mean 3 and display it, see Figure 1.

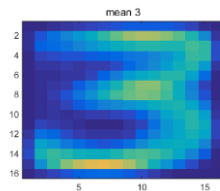


Figure 1: mean 3

- 2) Compute the covariance matrix of the whole dataset of 3s. Compute the eigenvalues and eigenvectors of this covariance matrix. Plot the eigenvalues (plot(diag(D)) where D is the diagonal matrix of eigenvalues), see Figure 2.

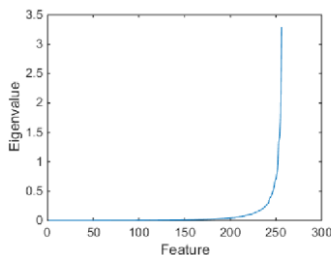


Figure 2: Eigenvalues

From Figure 2, we can see that a large percentage of eigenvalues are equal to zero. Only when the index is among (200, 256), the eigenvalues are non-zero. It also increases rapidly when the index is close to 250. Therefore, the 256 features can be reduced a lot.

- 3) Compress the dataset by projecting it onto one, two, three, and four principal components.

Take the first image of the dataset as an example, the reconstruction results are listed in Figure 3. We can see that when the number of principal components increases, the reconstruction error becomes smaller.

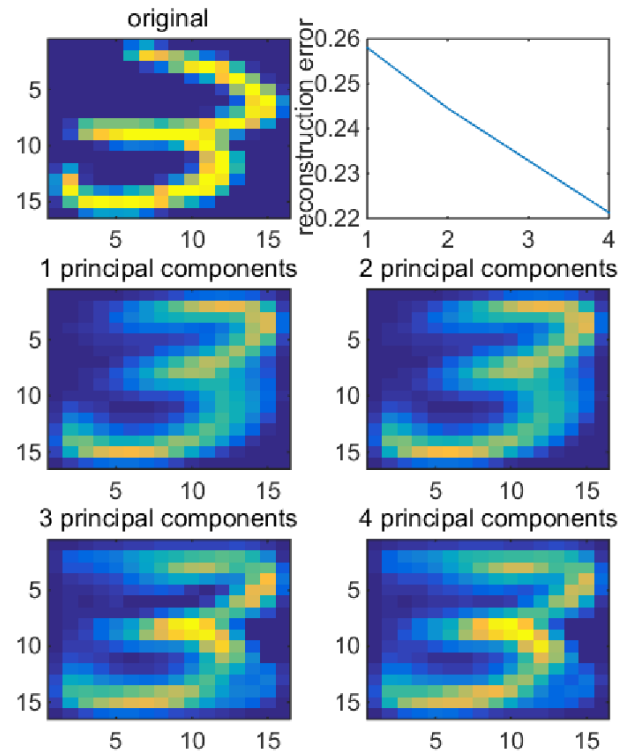


Figure 3: reconstruction of the 1st image

- 4) Write a function which compresses the entire dataset by projecting it onto k principal components, then reconstructs it and measures the reconstruction error.

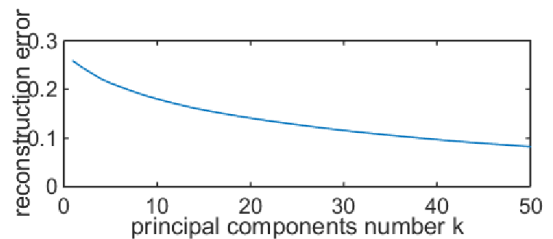


Figure 4: reconstruction error under k

From Figure 4, we can see that as k increases, from 1 to 50, the reconstruction error decreases but the decreasing rate slows.

- 5) What should the reconstruction error be if $k = 256$? If $k = 256$, the reconstruction error would be zero. The reason is that in this case, we choose all features without feature reduction.
- 6) Use the Matlab function cumsum to create a vector whose i -th element is the sum of all but the i largest eigenvalues for $i = 1: 256$. Compare the first 50 elements of this vector to the vector of reconstruction error versus k

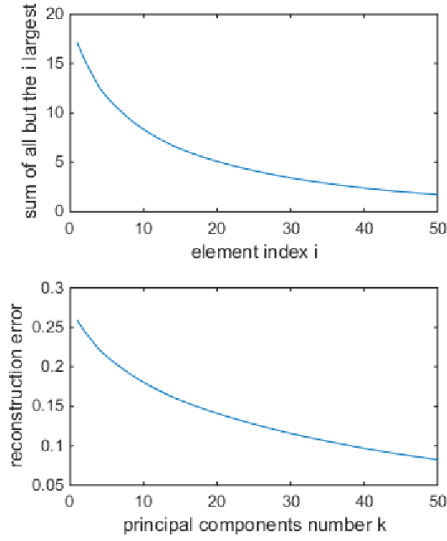


Figure 5: (1) the sum of all but the i largest eigenvalues;
(2) reconstruction error

From Figure 5, we can see these two share the same trend. It also indicates that reconstruction errors come from those eigenvectors which have not been chosen. And therefore, for those ignored eigenvectors, the sum of their corresponding eigenvalues to some extent describes reconstruction errors. And since the sum of the eigenvalues that we are not using is not very large, the eigenvalues fall off quickly when projecting onto the first few.

2 SOM applied to the Iris datasets

The purpose of Self-organizing Maps (SOMs) is to recognize groups of similar input vectors in an unsupervised fashion. The neurons in the layer of a SOM are originally arranged according to a predefined topology (gridtop, hextop, randtop). All the neurons in the neighborhood $N(i^*)$ of the neuron i^* are then updated according to the Kohonen rule. That is, for each index $i \in N(i^*)$ we have: $w_{(t)}^i = (1 - \alpha)w_{(t-1)}^i + \alpha x$, where α is a small scalar and $w_{(t)}^i$ denotes the vector of weights representing neuron i at step t .

In this section, we apply SOM to the Iris datasets. With different values for the grid size, topology and number of epochs, let's compare the performance by looking at the Adjusted Rand Index (ARI). Larger values of ARI mean the clustering results are more similar with the true situation. For each case, 10 repeated simulations were performed, and the average ARI was obtained.

1) Grid size

The number of epochs is 100 and the result about how ARI varies when grid size is changing is shown in Figure 6_ (1)-(3). We can see that:

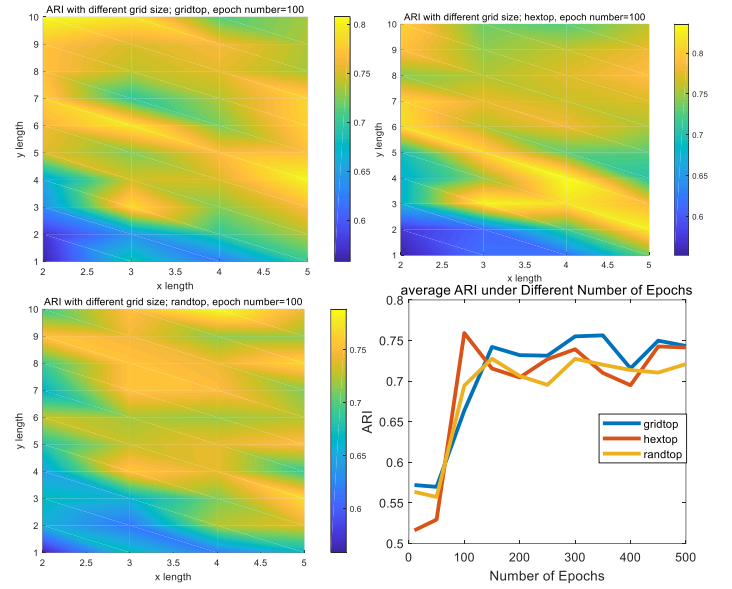


Figure 6: 1) ARI with different grid size; gridtop. 2) ARI with different grid size; hextop. 3) ARI with different grid size; randtop. 4) ARI with different number of epochs

- for all the three topologies, when grid size falls in the bottom left corner, the value of ARI is low. Therefore, properly increasing the grid size would help to make better clustering, though it won't change much after grid size is more than a certain range.
- These topologies seem to have similar effect. Besides, in these simulations, for gridtop case, the largest ARI occurs when the grid size is 3 by 6; for hextop case, the largest ARI occurs when the grid size is 4 by 4; for randtop case, the largest ARI occurs when the grid size is 5 by 3. I would use these three grid size when analyzing the influence of number of epochs.

2) Number of epochs

In Figure 6_ (4), it shows how ARI varies when the number of epochs is changing among [10, 50:50:500]. We can see: when the number of epochs is less than 100, ARI increases when it increases; but ARI doesn't change much when the number of epochs is among [100:500]. This is because when the number of epochs increases, it converges.

Artificial neural networks - Exercise session 4

Deep learning: Stacked Autoencoders and Convolutional Neural Networks

0685182 Yi Zhao

1 Digit Classification with Stacked Autoencoders

An autoencoder neural network is an unsupervised learning algorithm that applies backpropagation, setting the target values to be equal to the inputs. For classification, we can discard the “decoding” layers and link the last hidden layer to the classifier.

A stacked autoencoder is a neural network consisting of multiple layers of sparse autoencoders in which the outputs of each layer are wired to the inputs of the successive layer. We use greedy layer-wise training to train it. After this phase, fine-tuning using backpropagation can be used to improve the results by tuning the parameters of all layers at the same time.

Now we analyze the performance of digit classification using stacked autoencoders.

1.1 MaxEpochs

In the script DigitClassification.m, it trains an autoencoder with two hidden layers and a final layer. The 1st hidden layer has 100 neurons and 2nd layer 50 neurons. To find how the parameter MaxEpochs influence its training results, I make the value of MaxEpochs varies from 100 to 400 with step length 100 in both two hidden layers. The epoch number of the final layer keeps the same (1000). Result is shown in Figure 1.

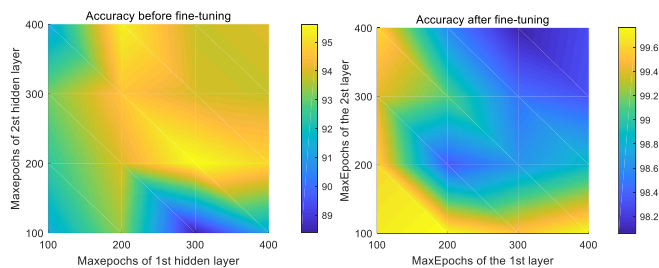


Figure 1 Accuracy for different MaxEpochs

From Figure 1, we can see:

- After finetuning, accuracy increased a lot, from around 92% to around 99%.
- If we look at the accuracy after finetuning (the second subplot), we can see increasing MaxEpochs doesn't guarantee better performance. Actually, in this simulation, the largest value (99.76%) of accuracy occurred when MaxEpochs equals to 100 in both hidden layers.

As for normal multilayer neural networks, at first, a normal neural network with one hidden layer was used. The hidden

layer has 100 neurons and the MaxEpochs is 1000. After 10 repeated simulations, the average value of accuracy is 96.66% and the maximum one is 97.88%, which is lower than that of the stacked autoencoder.

Then Classify digit with a normal neural network with two hidden layers. The 1st hidden layer has 100 neurons and 2nd layer 50 neurons. The MaxEpochs is 1000 in each layer. After 10 repeated simulations, the average value of accuracy is 96.77% and the maximum one is 97.52%, which is also lower than that of the stacked autoencoder. We can also see that more hidden layers don't guarantee better accuracy.

1.2 Number of Neurons

To find how the parameter Neuron Number influence its training results, I make its value varies from 100 to 400 with step length 100 in 1st hidden layers and from 25 to 100 with step length 25 in 2nd hidden layers. The neuron number in the final layer is 10. The MaxEpochs is 400 in the 1st hidden layer, 100 in the 2nd hidden layer and 1000 in the final layer. Result is shown in Figure 2.

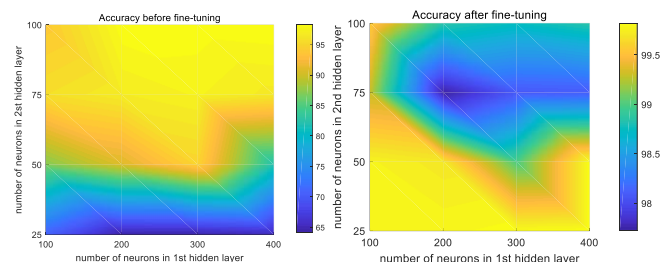


Figure 2 Accuracy for different neuron numbers

From Figure 2, we can see within a certain scope, larger value of neuron number in 1st hidden layer and smaller value of neuron number in 2nd hidden layer would result in higher accuracy. Therefore, to obtain a better result with different parameters (wrt to the default ones), we can make the neural number in 1st hidden layer to be 400 and the neural number in 2nd hidden layer 25. The accuracy is 99.82%, which is the largest value occurred.

1.3 A Stacked Autoencoder with More Hidden Layers

Here are some examples when the stacked autoencoder has more hidden layers.

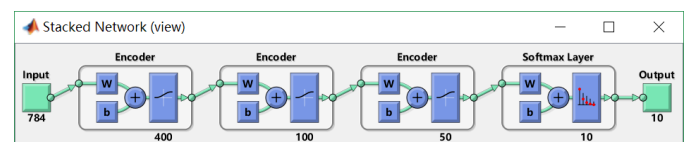


Figure 3: A stacked autoencoder with three hidden layers

For the stacked autoencoder in Figure 3, the MaxEpochs values are 400,100,100,1000 separately. The accuracy before finetuning is 47.62% and after finetuning is 99.52%.

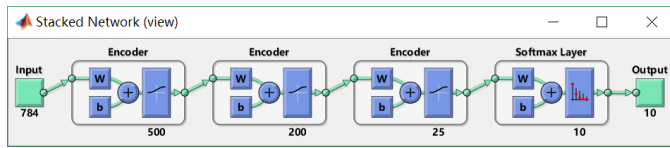


Figure 4: A stacked autoencoder with three hidden layers

Similarly, for the stacked autoencoder in Figure 4, the MaxEpochs values are same with that in Figure 3. The accuracy before finetuning is 62.22% and after finetuning is 99.80%.

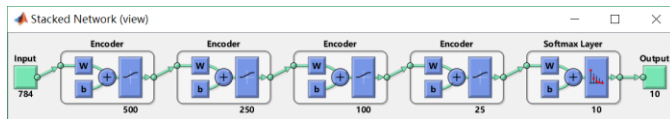


Figure 5: A stacked autoencoder with four hidden layers

For the stacked autoencoder in Figure 5, the MaxEpochs values are 400, 100, 100, 100, 1000 separately. The accuracy before finetuning is 35.74% and after finetuning is 99.46%. From these three examples, we may conclude that: it doesn't guarantee increasement in accuracy when increasing layers.

2 Answers to the Questions in Section 2.2

- Look at the first convolutional layer (layer 2) and at the dimension of the weights. what do these weights represent?

The dimension of the weights is $11 \times 11 \times 3 \times 96$. It means there are 96 $11 \times 11 \times 3$ convolutions. Convolutional layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume.

- Inspect layers 1 to 5. A ReLU and a Cross Channel Normalization layer do not affect the dimension of the input, what is the dimension of the input at the start of layer 6 and why?

The input are $227 \times 227 \times 3$ images. $(227-11)/4+1=55$. So after 'conv1', the output has dimension $55 \times 55 \times 3 \times 96$. Then after 'pool1', $(55-3)/2+1=27$, the output has dimension $27 \times 27 \times 3 \times 96$, which is the input at the start of layer 6.

- What is the final dimension of the problem (i.e. the number of neurons used for the final classification task)? How does this compare with the initial dimension?

After running the demo, the output size is 1000, which is much smaller than the input size ($227 \times 227 \times 3 = 154587$).

3 Digit Classification with CNN

Based on the script CNNDigits.m investigate some CNN architectures. The datastore contains 1000 images for each

of the digits 0-9. Hence, there are 10 classes.

3.1 Weight Dimension

When other parameters keep the same but the weight dimension among $[3 \times 3, 5 \times 5, 7 \times 7, 9 \times 9]$, time and accuracy is plotted in Figure 6.

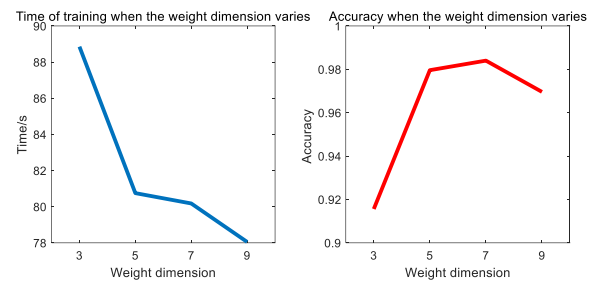


Figure 6: Time and accuracy when weight dimension varies

From Figure 6, we can see when the weight dimension increases, training time decreases, but the trend of accuracy depends. In this case, when it is 7×7 , the classification accuracy is the best.

3.2 Different number of convolutional layers

For the three networks in Figure 7, training accuracy is plotted in Figure 8 and training accuracy in Figure 9.

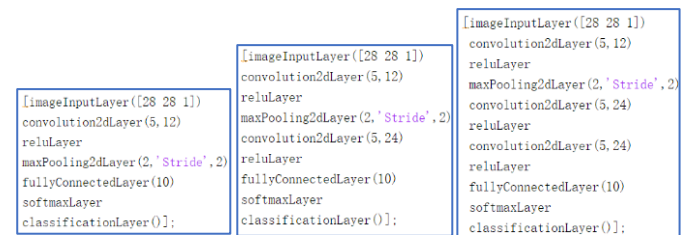


Figure 7: Different training networks

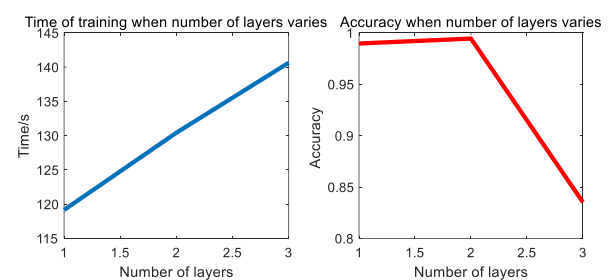


Figure 8: Time and accuracy for convolutional layers with different amounts

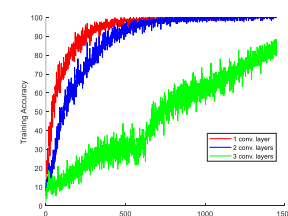


Figure 9: Training accuracy for convolutional layers with different amounts

We may conclude that: it takes more time for training deeper CNNs. And in this case, accuracy is highest when there are two convolutional layers. Maybe the third network needs more iterations and would achieve better. During training, networks with less convolutional layers converge first.

Artificial Neural Networks – Final Project

Nonlinear Regression and Classification with MLPs

0685182 Yi Zhao

1 Regression

In this section, the objective is to approximate a nonlinear function using a feedforward artificial neural network. Given a set of 13600 uniformly sampled datapoints. X1 and X2 contain the input variables and T1 to T5 are the 5 independent nonlinear functions evaluated at the corresponding point from (X1, X2). My student number is r0685182, therefore d1 to d5 are assigned as 8, 8, 6, 5, 2. The target function would be:

$$T_{new} = (8 * T1 + 8 * T2 + 6 * T3 + 5 * T4 + 2 * T5) / 29.$$

1.1 Define the Datasets

Now the dataset consists of X1, X2 and T_{new} . To draw 3 (independent) samples of 1000 points each, I have used the command `randsample` whose results are sampled uniformly at random, without replacement. Use them as the training set, validation set, and test set, respectively. Plot the surface of the training set with the help of the command `scatteredInterpolant`, as is shown in Figure 1.

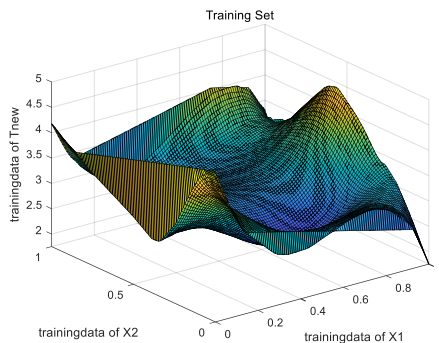


Figure 1: Surface of the training set

1.2 Build and Train the Neural Network

Neural networks are good at fitting functions. In fact, there is proof that a simple neural network can fit any practical function. Now let's use the training and validation sets to build and train the feedforward Neural Network with 2 inputs (X1, X2) and 1 output (T_{new}).

1.2.1 Learning algorithm and transfer function

In Exercise 1, we have analyzed the performance of different learning algorithms. The results show that generally `trainbr` has best performance. Therefore, I use `trainbr` to solve this problem.

From Figure 1, we can see the output (T_{new}) has values larger than 1. The default network for function fitting (or regression) problems, `fitnet`, is a feedforward network with the default tan-sigmoid transfer function in the hidden layer and linear transfer function in the output layer. Therefore, I

chose the `fitnet` default transfer functions `tansig/purelin`.

1.2.2 Number of hidden layers and number of neurons on each hidden layer.

To find a suitable model, I would analyze two situations: when the number of hidden layers is 1 and that is 2. When there is only 1 hidden layer, suppose the number of neurons varies among [10:10:100]. For all the simulations, the max number of epochs is 1000 (default). For every value of neuron number, perform 10 simulations and then calculate the corresponding average training and validation MSE. The result is plotted in Figure 2.

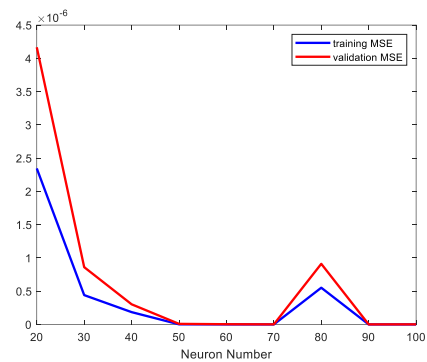


Figure 2: Training and validation MSE when neuron number varies for the feedforward network with 1 hidden layer

According to Figure 2, we can see the validation MSE decreases first and then increases when neural number increases. Besides, when neuron number is 70, it achieves the lowest validation error, which is 1.1555e-09.

When there are 2 hidden layers, suppose the neuron number of 1st layer varies among [10:10:100] and that of 2nd one varies among [5:5:25]. For all the simulations, the max number of epochs is 100. For every combination of neuron numbers, perform 5 simulations and then calculate the corresponding average validation mse. The result is plotted in Figure 3.

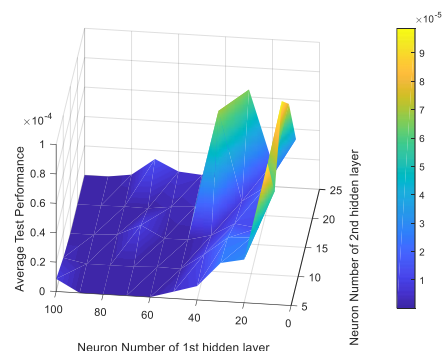


Figure 3: Validation MSE when neuron numbers vary for the feedforward network with 2 hidden layers

From Figure 3, we can see generally for the feedforward network with 2 hidden layers, when the neuron number of 1st layer increases, the validation error would decrease first and then increase; when the neuron number of 2nd layer increases, the validation error would decrease first and then increase. The best performance occurred when the neuron numbers are 90 for the 1st layer and 5 for the 2nd layer. The lowest validation error is 2.6324e-09, which is not better than the case when there is only 1 hidden layer.

1.2.3 Model Chosen

According to Section 2.2.2, I would select the ANN model with 1 hidden layer and the neuron number is 70, as is shown in figure 4.

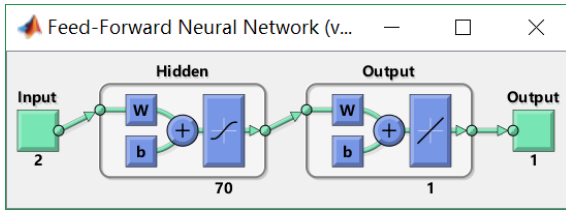


Figure 4: The chosen model

In addition, the learning algorithm is trainbr, the transfer function functions are tansig/purelin and the number of max epochs is 1000.

1.3 Test Performance Assessment

Now let's evaluate the performance of the selected model on the test set. The surface of the test set and the approximation given by the network are drawn in Figure 5.

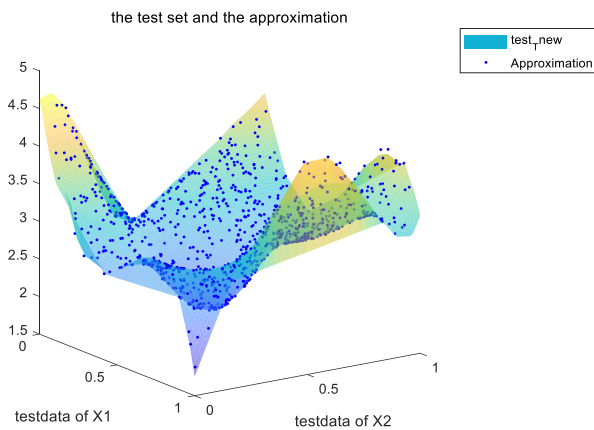


Figure 5: The approximation and the surface of the test set

Plot the error curve, see Figure 6.

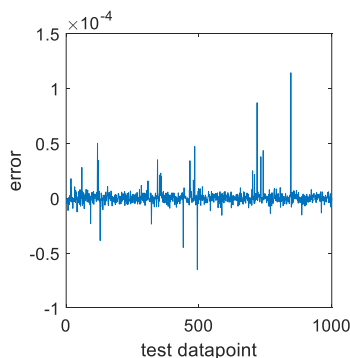


Figure 6: Error Curve

Compute the Mean Squared Error on the test set. The MSE is 5.8194e-11 in this simulation. The training MSE in this case is 1.2942e-11. Therefore, we may conclude that generally training MSE is smaller than test MSE. Anyway, if training MSE is much smaller than test MSE, there might be an overfitting and if training MSE is larger than test MSE, there might be an underfitting.

To improve the performance of the network, we can apply Bayesian inference. Additionally, here I sampled 3000 data points randomly and applied training. We can improve network performance by using more data.

2 Classification

In this section, the objective is to build feedforward neural networks for classification and explore the benefits of using dimensionality reduction techniques. Since the last digit of my student number is 2, I would solve the binary classification problem of class 5 (C+) vs. class 7(C-), white wine.

2.1 The Specific Dataset

The specific dataset I obtained is a 13×2337 matrix. The first 11 rows are the 11 features and the last 2 rows are the target. There are 2337 examples.

2.2 ANN on the original dataset

Now let's create and train feedforward neural networks to solve the classification task.

2.2.1 The network architecture and transfer functions

Since the numbers of datapoints and features are not large, the network architecture would be with 1 hidden layer. The input is in 11 dimensions and the output in 2 dimensions. In addition, since tansig is the default transfer function for both hidden and output layers in patternnet, I would select it as the transfer function.

2.2.2 The learning parameters

After the network architecture and transfer functions are determined, the main learning parameters needed to discuss are learning algorithms and the number of neurons. In this section, I would compare two learning algorithms: trainscg (the default learning algorithm) and trainlm (which has showed great performance in our past experiments). At the same time, the number of neurons would vary among [10:2:40]. For all the simulations, the max number of epochs is 1000. For every combination, train the network with the training data set, perform 5 simulations and then calculate the average correct classification ratio (CCR) for the validation set. The CCR is defined as:

$$CCR = \frac{\text{Number of Correctly classified data} \times 100}{\text{Total number of data}}$$

Since their 2337 datapoints, suppose we use the first 1500 datapoints as the training set and the last 337 datapoints as

the test set. Other datapoints are used to form the validation set. The simulation result (CCR for the validation set) is shown in Figure 7.

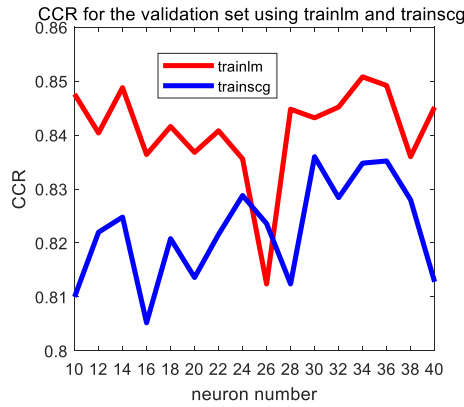


Figure 7: CCR for the validation set

From figure 7, we can see the best performance occurred when the number of neurons is 34 and the training algorithm is trainlm. The corresponding validation CCR is 0.8508. The network would be in the form as Figure 8 shows.

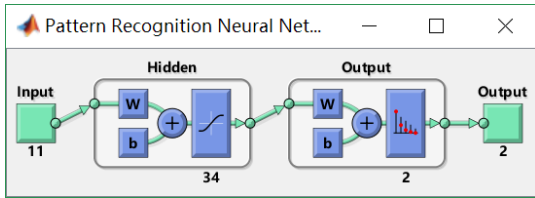


Figure 8: The network when best validation CCR was occurred

2.3 PCA on the wine dataset

In this section, let's perform PCA on the wine dataset. At first, we compute the eigenvectors and eigenvalues of the covariance matrix of the training set. The training set is the first 1500 datapoints. The eigenvalue spectrum is shown in Figure 9.

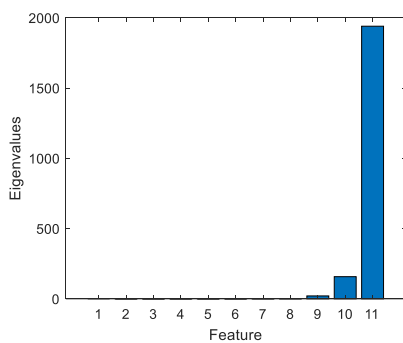


Figure 9: Eigenvalues

From Figure 9, we may get the conclusion that we can use 3 principal components to perform dimensionality reduction via PCA.

Now let's reconstruct all the training, validation and test data using the selected 3 principal components. Suppose we chose the first 1500 datapoints to form the training dataset, the last 337 datapoints to form the test dataset and others to form the validation set. In the end, the size of the datasets

would be:

Table 1: Size of reconstructed datasets

dataset	size
finanl_PCA_traindata	1500×3
finanl_PCA_valdata	500×3
finanl_PCA_testdata	337×3

2.4 ANN on the dimension-reduced dataset

Now let's use the reconstructed dataset design a new feedforward network to classify the data. Again, train this new network to obtain the highest CCR on the new validation data.

Similarly, I would compare trainlm and trainscg and the number of neurons would vary among [10:2:40]. The max number of epochs is 1000.

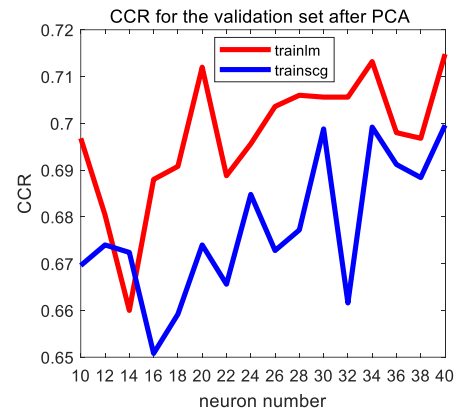


Figure 10: CCR for the validation set after PCA

In this case, the best performance also occurred when the number of neurons is 34 and the training algorithm is trainlm. The corresponding validation CCR is 0.7132.

2.5 Comparison

Comparing both networks: the one trained using the original data and the one trained using the lower dimensional data, we may get the conclusions as following:

- After PCA, the validation CCR would be smaller;
- No matter before and after PCA, the best performance is occurred at the same network;
- In practice, we can adopt PCA first, and training with the dimension reduced data to speed up, and finally when the network with best performance is obtained, we then use it to deal with new raw data.

Artificial neural networks – Final Project

Character Recognition with Hopfield Networks

0685182 Yi Zhao

1. Problem description and data Preparation

In this report, let's investigate the retrieval-capabilities of the Hopfield network, applied to the letters of the alphabet. The collection of characters is created by pre-pending the lowercase characters of my first and last name to the set of all capital characters, that is: y, i, z, h, a, o, A, B, C, D, ... etc. The first 12 characters are displayed in Figure 1.

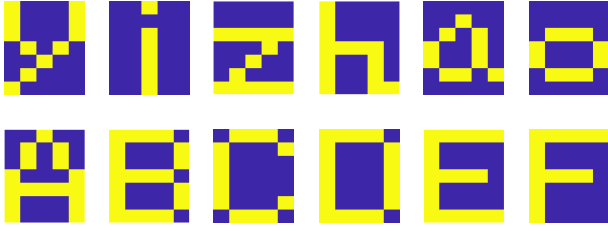


Figure 1 First 12 characters of the dataset

Besides, I would transform the pixels from $\{0,1\}$ into $\{-1, +1\}$. Since $\xi_j \in \{-1, +1\}$ guarantees $\xi_j^2 = 1$, when the interconnection weight is $w_{ij} = \frac{1}{N} \xi_i \xi_j$, then the output $\text{sign}(\sum_j w_{ij} \xi_j)$ would equal to ξ_j , that is, the pattern ξ is stored as an equilibrium point.

2. A Hopfield neural network

In this section, a Hopfield neural network was build and tested to see whether it can retrieve the first 5 of these characters (y, i, z, h, a).

At first, noise is added to the input by inverting three randomly chosen pixels for each character. An example is shown in Figure 2. The first line are original characters and the second line are distorted ones.

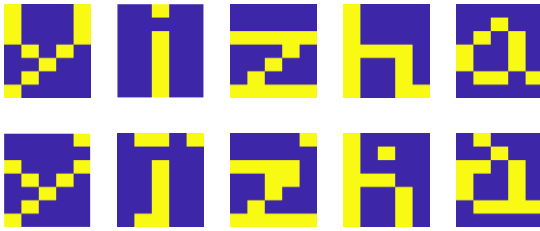


Figure 2: Original and distorted characters

Then, using a Hopfield network to recall these distorted patterns. After 1 iteration, the reconstructed characters are shown in Figure 3. Rounding the pixels of the final state to +1 and -1, characters are recalled.



Figure 3: reconstructed character after 1 iteration

In addition, after 6 iterations, the reconstructed characters can be the same as the original ones even without rounding the pixels of the final state to +1 and -1.

Besides, this setup (3 distorted pixels, 5 attractor states) is rather easy for the network. Increasing the number of iterations excessively did not change the results or lead to any spurious states.

3. Determine the critical loading capacity

In this section, we determine the critical loading capacity of the network. Characters are also distorted by inverting three randomly chosen pixels and try to retrieve them. The number of patterns P varies among $[4:2:32]$. And the number of iteration times I varies among $[1:1:20]$. The pixels of the final state would be rounded to +1 and -1. Then, we calculate the error for different values of P . The error is measured according to the total number of wrong pixels over all the retrieved characters, as is shown in Figure 4.

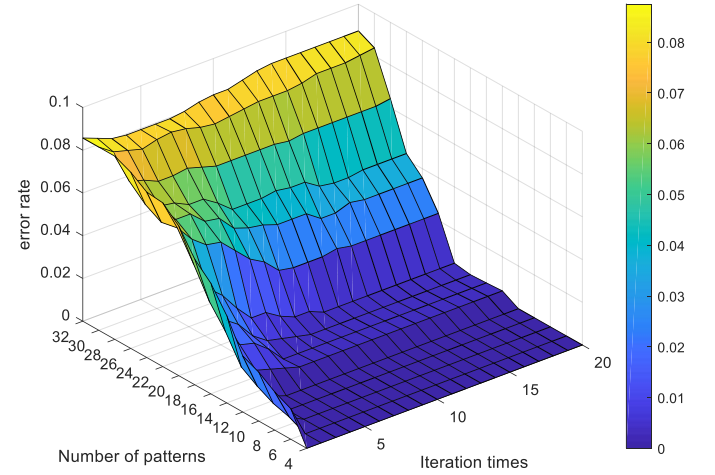


Figure 4: Error rate when P and I varies

From Figure 4, we can see when the number of patterns increases, the error rate would increase, too. According to the values I got, the critical loading capacity is 12, which is larger than the theoretical one, it might be due to the noise it small (only 3 pixels inversed).

Theoretically, if we store P patterns in a Hopfield network with N nodes (now N is 35), then the probability of error

is: $Pro_e = \frac{1}{\sqrt{2\pi\sigma}} \int_1^\infty \exp\left(-\frac{x^2}{2\sigma^2}\right) dx$, where $\sigma = \sqrt{P/N}$. A

long and sophisticated analysis of the stochastic Hopfield network shows that if $P/N > 0.138$, small errors can pile up in updating and the memory becomes useless. The storage capacity is $P_{max} \approx N \times 0.138 \approx 4.88 \approx 5$.

Therefore, the capacity we got by simulation is similar with the theoretical one.

The presence of spurious patterns was not recorded in the first experiment which was trained with only 5 characters and simulated with patterns of which only 3 pixels were inverted. Repeating iterations during simulation would eventually lead to converge the distorted patterns to one of the retrieval states. However, if the setting exceeds the network's capacity it will converge to spurious patterns, which are different from the training patterns but are also at local minimum. For example, when the number of patterns is 24, and the number of iterations is 40, there might be some spurious patterns shown in Figure 5:



Figure 5: Examples of spurious patterns

These spurious patterns are mostly awkward mixtures of other stored patterns, encountered after simulation of the Hopfield network.

4. Other options

To store 25 characters with three random inversions and to retrieve them perfectly, my first idea is to construct the letters with more pixels: $25/0.138 \approx 182$. However, it seems to be unpractical. Since the digit recognition is a classification task, we might be able to deal with it with multilayer networks, stacked autoencoders, or CNNs. In this section, I would use multiple layer networks and stacked autoencoders.

No matter what type of network we choose, they all require large amount of training examples. Therefore, the first step is to create enough examples. At the beginning, we have 25 characters and their corresponding targets. So, I made 50 copies of the original dataset. Than for each example, randomly choose 3 pixels and do inversion. And now we have 750 noised datapoints. Using the first 500 datapoints as training set and the last 250 datapoints as the test set. I chose a stacked autoencoder with 2 hidden layers. Let the neuron number of the 1st layer varies among [20:2:50] and that of the 2nd layer varies among [10:1:20]. The test performance is measured by error rate, that is:

$$error\ rate = \frac{number\ of\ error\ pixels}{number\ of\ all\ pixels}.$$

The number of max epochs in each layer (hidden layer and output layer) is 1000. Error rate is calculated after fine tuning.

The test error rate is shown in Figure 6.

From Figure 6, we can see among the value range we chose,

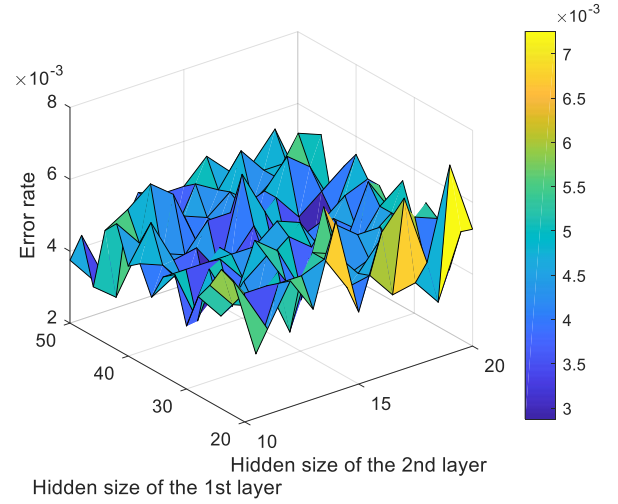


Figure 6: Test error rate when hidden sizes vary
the test error rate is very small. The smallest one occurred with value of 0.0029 when the neuron number of 1st layer is 46 and the neuron number of the 2nd layer is 11, as is shown in Figure 7.

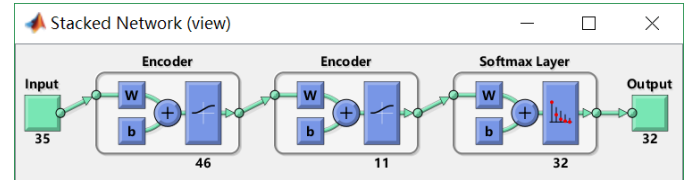


Figure 7: the stacked autoencoder with best performance
Therefore, we can use stacked autoencoder to achieve our goal of storing and classifying 25 characters.