A few autoencoders were constructed for finding binary code for 16 decimal numbers from 0 to 15. All the autoencoder has used a position encoding scheme. I have used 16 nodes for each number. The position encoding means that for ith number, ith node is 1 while all the other nodes are 0. For example, for the first number, 0, the first node is 1 and all the other nodes are zero. As a result, the input layer looks like following:

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

For all these autoencoders, we want the output layer to be exactly the same as the input layer. Two activation functions were tried for the following architectures:

1. A 3-layer NN (with layers for input, hidden and output). The hidden layer has 5 perceptrons.
2. A 3-layer NN. The hidden layer has 4 perceptrons.
3. A 3-layer NN. The hidden layer has 3 perceptrons.
4. A 5-layer NN. The 1st, 2nd, and 3rd hidden layers have 8, 4, and 8 perceptrons, respectively.

Since we are interested in the stable states of all hidden layers, during the training of the model I have set the iteration number to be very big so that I can get the model parameters for the converged model. For all the autoencoders I have built, I have checked that the iteration number for which whose cost function stops decreasing and forms a plateau is much smaller than the iteration number I have set. This suggests that the model I have trained has converged. All the output layers have checked after the models have converged. They all are the same as the input layer. The typical example of the output layer is as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1.0E+00 | 1.3E-19 | 4.5E-53 | 5.0E-52 | 9.8E-53 | 1.0E-15 | 1.5E-31 | 1.2E-15 | 2.0E-50 | 4.1E-18 | 3.8E-18 | 9.1E-16 | 8.8E-16 | 2.0E-19 | 1.6E-16 | 1.2E-15 |
| 4.3E-16 | 1.0E+00 | 2.7E-18 | 2.2E-18 | 1.5E-48 | 4.7E-45 | 6.3E-31 | 2.0E-38 | 4.5E-49 | 5.3E-17 | 1.2E-15 | 3.6E-15 | 6.1E-16 | 1.4E-15 | 4.9E-46 | 6.1E-16 |
| 2.4E-41 | 7.9E-16 | 1.0E+00 | 2.6E-16 | 1.2E-15 | 2.2E-41 | 9.4E-32 | 1.2E-15 | 1.5E-16 | 5.0E-16 | 5.9E-46 | 2.4E-15 | 5.0E-47 | 7.6E-16 | 3.0E-46 | 4.3E-16 |
| 3.0E-50 | 1.5E-15 | 2.5E-16 | 1.0E+00 | 1.4E-15 | 7.5E-32 | 6.7E-15 | 1.3E-49 | 1.7E-31 | 3.6E-60 | 4.5E-16 | 1.8E-26 | 4.3E-32 | 1.6E-43 | 7.9E-61 | 3.3E-15 |
| 2.2E-40 | 1.5E-44 | 7.8E-16 | 4.5E-16 | 1.0E+00 | 2.8E-15 | 6.2E-32 | 6.6E-16 | 3.4E-17 | 1.3E-45 | 2.9E-16 | 2.2E-41 | 4.3E-47 | 1.2E-15 | 1.9E-17 | 8.4E-16 |
| 4.2E-16 | 8.2E-79 | 3.6E-68 | 5.4E-38 | 1.0E-15 | 1.0E+00 | 3.7E-15 | 2.2E-15 | 2.6E-25 | 3.3E-66 | 7.1E-28 | 2.7E-32 | 7.3E-38 | 4.2E-56 | 1.1E-15 | 1.8E-21 |
| 2.2E-47 | 3.1E-65 | 5.9E-68 | 2.6E-15 | 6.6E-64 | 3.9E-15 | 1.0E+00 | 2.1E-60 | 1.4E-15 | 1.6E-65 | 5.3E-63 | 5.5E-15 | 3.2E-15 | 2.2E-112 | 7.2E-64 | 7.9E-55 |
| 1.2E-15 | 7.2E-56 | 7.2E-16 | 1.7E-57 | 3.4E-16 | 1.8E-15 | 1.4E-40 | 1.0E+00 | 2.0E-17 | 6.0E-16 | 2.1E-57 | 1.2E-18 | 5.5E-59 | 9.1E-16 | 1.0E-17 | 8.1E-16 |
| 5.0E-49 | 1.3E-64 | 1.3E-16 | 1.3E-35 | 1.8E-19 | 9.1E-22 | 6.7E-15 | 4.9E-15 | 1.0E+00 | 8.7E-16 | 7.1E-69 | 6.7E-24 | 2.0E-36 | 6.2E-50 | 2.2E-20 | 6.7E-51 |
| 7.5E-16 | 3.2E-16 | 2.2E-16 | 8.7E-48 | 9.8E-47 | 5.1E-38 | 1.0E-30 | 8.0E-16 | 7.2E-18 | 1.0E+00 | 1.6E-45 | 2.6E-15 | 1.4E-16 | 7.6E-16 | 1.4E-15 | 3.8E-39 |
| 1.1E-15 | 1.7E-17 | 1.2E-46 | 3.3E-16 | 2.5E-16 | 5.6E-16 | 2.9E-28 | 4.8E-37 | 7.3E-46 | 4.8E-47 | 1.0E+00 | 3.3E-40 | 1.2E-15 | 9.8E-17 | 9.0E-16 | 2.7E-15 |
| 2.7E-15 | 2.5E-16 | 1.5E-15 | 7.8E-31 | 2.6E-66 | 1.3E-34 | 3.6E-15 | 1.2E-19 | 3.7E-31 | 1.0E-16 | 2.9E-68 | 1.0E+00 | 1.4E-33 | 6.2E-51 | 6.0E-69 | 4.3E-16 |
| 2.9E-15 | 1.0E-15 | 1.1E-60 | 6.7E-34 | 8.0E-63 | 6.1E-30 | 6.2E-15 | 9.8E-53 | 5.6E-34 | 8.2E-16 | 7.8E-16 | 6.9E-29 | 1.0E+00 | 1.0E-43 | 1.0E-15 | 1.7E-52 |
| 8.8E-16 | 1.4E-15 | 6.1E-22 | 8.5E-53 | 3.9E-23 | 1.6E-42 | 1.1E-65 | 3.8E-15 | 5.5E-55 | 8.8E-18 | 1.4E-15 | 4.4E-43 | 4.1E-47 | 1.0E+00 | 2.0E-17 | 3.4E-16 |
| 1.1E-15 | 3.2E-48 | 4.5E-49 | 1.1E-54 | 3.4E-21 | 3.4E-15 | 2.8E-36 | 7.9E-17 | 5.9E-22 | 1.3E-15 | 3.5E-19 | 5.0E-45 | 9.7E-20 | 1.4E-15 | 1.0E+00 | 1.5E-42 |
| 1.7E-15 | 2.0E-16 | 2.9E-17 | 1.5E-15 | 2.7E-16 | 2.0E-21 | 1.1E-35 | 2.0E-15 | 1.7E-52 | 2.9E-53 | 1.1E-16 | 1.6E-16 | 4.2E-53 | 3.2E-16 | 1.1E-53 | 1.0E+00 |

Since this is a 16\*16 matrix and very big, I will not report the output layer for the rest of this writeup. However, all the autoencoder models I have trained have the same output layers which are the same as the input layers.

I have used fmincg function in octave for constructing sigmoid activation autoencoder, this is a function file provided by the Machine Learning class of Coursera as part of the homework. In terms of the ReLU activation autoencoder, I have used Keras autoencoder in python. I have referenced the following website: <https://blog.keras.io/building-autoencoders-in-keras.html>.

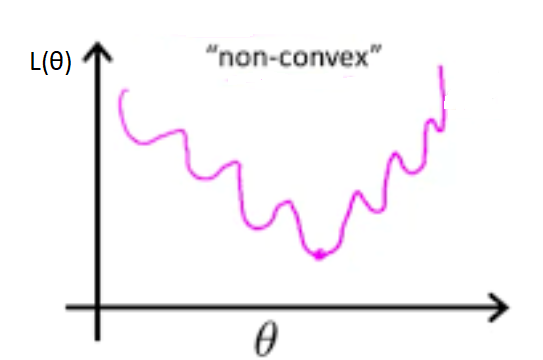
Before I dive into the details about the model constructing, I want to introduce the cost function I have used. For the two activation function, I have used the same cost function. Since we want all the output for the output nodes to be in the range [0, 1], logistic regression was applied to the output layer. That is to say

(1)

where is logistic regression. Because of the use of logistic regression, I didn’t use the following loss function for linear regreesion.

(2)

where is the true label for each sample, is the output from logistic regression. If we plot the loss function as a function of ϴ using equation 2, the result is as follows:



It is clear from the above plot that Logistic Function will cause the output to be wavy, causing many local optima. If we apply gradient decent on the above non-convex function, it is not guarantee the algorithm will converge to the global minimum. Thus another loss function is used here, which is shown in equation 3.

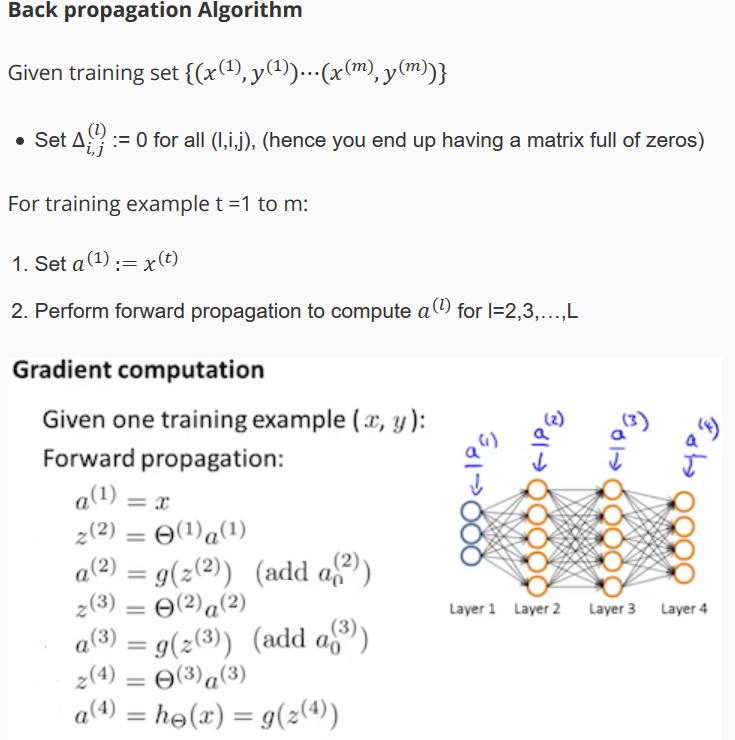
J(ϴ) = (3)

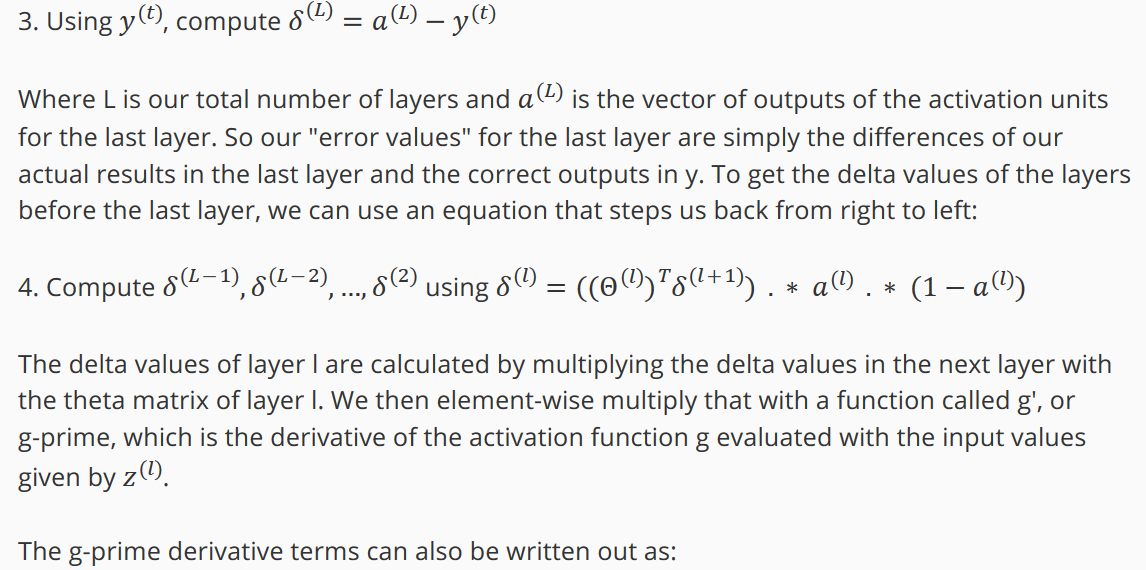
where m is the number of samples, k is the number of output classes. The loss function for ith sample which have kth label is inside the brackets, which is the loss function for logistic regression and is a convex function.

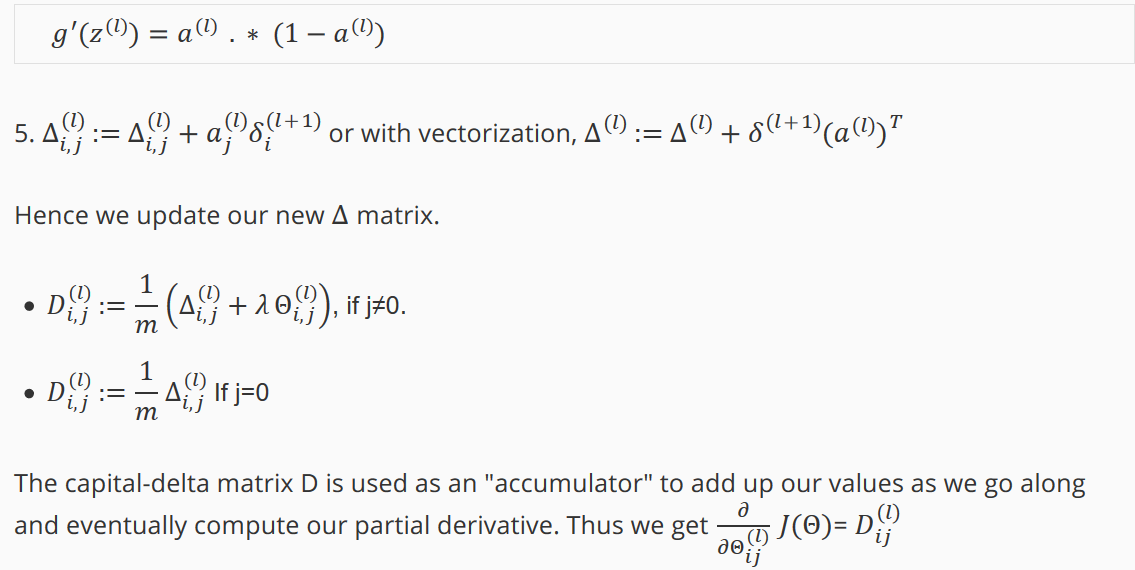
For the rest of the writeup, I will separately describe how I built the model and the results I got using these two activation function.

**Autoencoder using sigmoid activation**

According to the online course of machine learning by Stanford university in Coursera, using equation 3 as the loss function and the sigmoid function as activation function, the Back propagation Algorithm is as follows.







The weights for each layer then were updated using the following equation

where J(Θ ).

As mentioned above, fmincg function in Octave was used to train the model. This function takes in three arguments: function name(f), initial weight(X), and stop criteria(options). Then fmincg will minimize a continuous differentialble multivariate function f with the starting point given by X. The function returns when either the length of the run is up, or if no further progress can be made (ie, we are at a minimum, or so close that due to numerical problems, we cannot get any closer). By setting the length of the run very big as discussed above, I can make sure the function returns the converged model parameters. Inside the function f, I have defined the cost function using equation 3 and a vector of partial derivatives of f. So fmincg will minimize the cost function using partial derivatives of f and the backpropagation algorithm as described above. The initial weight X is assigned randomly according to randInitializeWeights function. This function takes in two arguments: L\_in, L\_out. Then X = RANDINITIALIZEWEIGHTS(L\_in, L\_out) randomly initializes the weights of a layer with L\_in incoming connections and L\_out outgoing connections. The random initialization is achieved by rand function. This function returns a matrix with random elements uniformly distributed on the interval (0, 1). By setting X = rand(L\_out, 1 + L\_in) \* 2 \* epsilon\_init - epsilon\_init, X can return a matrix with random elements uniformly distributed on the interval (-epsilon\_init, epsilon\_init). These initial weights were then save in mat format and can be uploaded as the initial weights using ReLU activation function. With the same initial random weights, we can compare the runtime and steps using these two activation functions.

fmincg function returns the weights for each layer for the converged model, the cost value of the model and the iteration number for the model to converge. The weights then can be used to retrieve the hidden layer value by sigmoid(weights \* input), where sigmoid means sigmoid function applied to (weights\*input). Before the training of the model, I have used the following function to get the current time in seconds: mktime (localtime (time ())). I have set the time to be t1. Similarly, I get the time after the convergence of the model and set it to be t2. Then t2-t1 is the running time for the model to converge.

For each autoencoder architecture, the iteration number and runtime for the model to converge, and the stable states of all hidden layers after you train the autoencoders using different initial weights are listed below.

**A 3-layer NN (with layers for input, hidden and output). The hidden layer has 5 perceptrons.**

Try 1

Iteration number: 829

runtime: 14.6 s

hidden layer

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |

Try 2

Iteration number: 1557

runtime: 21.2 s

hidden layer

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |

Try 3

Iteration number: 1918

runtime: 23.8 s

hidden layer

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |

Try 4

Iteration number: 1995

runtime: 24.1 s

hidden layer

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

Try 5

Iteration number: 993

runtime: 15.8 s

hidden layer

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |

On average the iteration number is about 1500 and the runtime is 20s.

If the sequence of the columns is important, the stable states of all hidden layers for 5 runs are different. Take the stable states of all hidden layers for integer 1 as an example, that corresponds to the second column. The states for different runs are:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1st run | 0 | 1 | 0 | 1 | 1 |
| 2nd run | 1 | 0 | 1 | 0 | 1 |
| 3rd run | 0 | 0 | 0 | 1 | 0 |
| 4th run | 0 | 0 | 1 | 1 | 1 |
| 5th run | 0 | 0 | 0 | 0 | 1 |

However, it seems that for all the 5 runs, the 16 column values are different from each other, which represent 16 different input values. In that aspect, the results are stable.

**A 3-layer NN (with layers for input, hidden and output). The hidden layer has 4 perceptrons**.

Try 1

Iteration number: 4506

runtime: 45 s

hidden layer

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |

Try 2

Iteration number: 4399

runtime: 43 s

hidden layer

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |

Try 3

Iteration number: 4379

runtime: 46 s

hidden layer

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |

Try 4

Iteration number: 3879

runtime: 40 s

hidden layer

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |

Try 5

Iteration number: 3483

runtime: 38 s

hidden layer

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |

On average, iteration number is 4000 and runtime is 42 s.

In terms of the stability of the hidden layer, it is the same as the autocoder with 1 hidden layer and 5 perceptrons for this hidden layer. That is to say, if the sequence of the columns is important, the stable states of all hidden layers for 5 runs are unstable. But for all the 5 runs, the 16 column values are different from each other, which represent 16 different input values. In that aspect, stable states of all hidden layers are stable.

**A 3-layer NN (with layers for input, hidden and output). The hidden layer has 3 perceptrons.**

Try 1

Iteration number: 16584

runtime: 147 s

hidden layer

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |

Try 2

Iteration number: 12719

runtime: 121 s

hidden layer

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |

Try 3

Iteration number: 22001

runtime: 194 s

hidden layer

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Try 4

Iteration number: 10840

runtime: 106 s

hidden layer

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |

Try 5

Iteration number: 18059

runtime: 173 s

hidden layer

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

On average, the iteration number is 15000 and run time is 150s. If the sequence of the column is important, the stable states of the hidden layer for 5 runs are not stable. Moreover, for this autoencoder architecture, for all the 5 runs among 16 column values, there are some columns are the same as others. This suggest that this autoencoder architecture can’t differentiate 16 different input values. This makes sense since we will need 4 bits to express 0-15, this suggest that only using 3 bits we can’t express 0 to 15.

**A 5-layer NN. The 1st, 2nd, and 3rd hidden layers have 8, 4, and 8 perceptrons, respectively**

try 1

Iteration number: 227761

runtime: 4014 s

hidden layer 1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |

hidden layer 2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |

hidden layer 3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |

try 2

Iteration number: 5360

runtime: 117 s

hidden layer 1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |

hidden layer 2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

hidden layer 3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |

try 3

Iteration number: 28157

runtime: 495 s

hidden layer 1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

hidden layer 2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

hidden layer 3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

try 4

Iteration number: 4934

runtime: 101 s

hidden layer 1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

hidden layer 2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |

hidden layer 3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |

try 5

Iteration number: 93105

runtime: 1588 s

hidden layer 1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

hidden layer 2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |

hidden layer 3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

On average, the iteration number is 71860 and run time is 1260s. If the sequence of the column is important, the stable states of the hidden layer for 5 runs are not stable. Moreover, for this autoencoder architecture, for all the 5 runs among 16 column values, there are some columns are the same as others for all the three hidden layer. This suggest this architecture can’t differentiate the 16 different input values. Since if we want to use the binary code to express numbers, only one step is involved. Thus the three layer architecture doesn’t work well for this problem. This homework suggests that the autoencoder architecture(like # of the hidden layer and # of nodes in each hidden layer) depends on the problem we try to solve.

**Autoencoder using ReLU activation**

Keras autoencoder in python was used to construct autoencoder using ReLU activation. We first build the input matrix like the following:

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Then with np.squeeze( np.asarray(input)), I change the input type from matrix to <class 'numpy.ndarray'> which is required in Keras. For the three hidden layer architecture, I first define the input\_size, hidden\_size1, hidden\_size2, and hidden\_size3 to be the number of nodes in the input layer, hidden layer 1, hidden layer2, and hidden layer3. I don’t need to define the output layer because for autoencoder, it is the same as the input. Setting “input\_img = Input(shape=(input\_size,))”, the model knows what input shape it should expect. Then the following four lines define the second, third, fourth, and output layer.

hidden\_1 = Dense(hidden\_size1, activation='relu')(input\_img)

hidden\_2 = Dense(hidden\_size2, activation='relu')(hidden\_1)

hidden\_3 = Dense(hidden\_size3, activation='relu')(hidden\_2)

output\_img = Dense(input\_size, activation='sigmoid')vhidden\_3)

The first argument inside the Dense function tells how many nodes in that layer, moreover, we can set activation to be relu for hidden layer 1,2 and 3. Since we want the output in the range of (0,1), the activation function for the output layer was set to be sigmoid. The string at the end of dense is the input for that layer. Keras uses tensor flow. A layer instance is callable (on a tensor), and it returns a tensor. Input tensor(s) and output tensor(s) can then be used to define a model. model.compile(optimizer='adam', loss='binary\_crossentropy') gives the loss function and optimizer. The choice of the loss function has been discussed above. mode.fit starts the training. By setting epochs to be very big number, I observe the specific epoch when loss function stops decreasing and forms a plateau to ensure the model converges. predict function can be used to retrieve the hidden layer and output layer after the model converges.

The results for four different autoencoder architectures are in the folder.

Compared with the hidden layer using sigmoid function, the hidden layer using ReLu is no longer in the range of (0,1). It is harder to examine the stability of the stable states of all hidden layers using ReLU activation function.

In theory, we would expect the number of steps to convergence and total running time is smaller for ReLU activation function than using sigmoid function because using ReLU activation function, we don’t have gradient vanishing issue. As discussed in class, back propagate the error at the output layer backward into the network to compute gradient: the weight from a hidden node hi to an output node yj.

=

where L is the loss function,

thenwill be close to 0. We use to update . With we change very little, thus it only make small progress for each iteration. Thus the iteration number and runtime will be larger when using sigmoid function. However, in this problem, since I am using two different packages in two different language. It is not easy to compare the runtime and iteration number. Thus I have run one test using Keras autoencoder. As seen in the results for autoencoder using one hidden layer, 5 perceptrons, and ReLU activation, after 32459 Epoch, ~15 minutes, the autoencoder reaches a stable train loss value of 1.0121e-07. I have changed the activation to be sigmoid and load the same initial weights as used in ReLu activated autoencoder.

After 101322 Epoch, ~25 minutes, the autoencoder haven’t reached a stable train loss value.

This agrees with what we would expect about the runtime and iteration number for this two activation functions.