P1:

- **Part0: Data Preparation and other General information:**

  Using the position Encoding Schema described in class, the number 0 – 15 is coded to have the matrix form represented by the following table:

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

Which is used as both input and desired output matrix of the trained autoencoder. TensorFlow with Kersa is used to train the autoencoder. Since the we are using non-linear activation function here, I used cross-entropy method instead of MSE as loss function for optimization. For the ease of computation, the convergence requirement is set to the gradient at one iteration is smaller than 10^-10 rather than gradient equals to zero. I tested both condition and although setting the condition to gradient = 0 gives slightly better result, the cost of computation outweighs its benefit.

Result and Discussion for each configuration are summarized at next few pages as follow:

- **PART 1: 3 Layer NN with 5 perceptrons in hidden layer, sigmoid function & RELU as activation function:**

The first table summarize the initial weights, lost at convergence and epoch for each run. The second table summarize the coding schema generated by perceptrons from 5 runs. Since the numbering of perceptrons are arbitrary for each run (that says the sequence of perceptrons can be changed for each run), some rearrangement is made to reflect the actual result from each perceptrons in table 2.

	Initial Weights	Loss at convergence	Epoch	Run Time
Run 1	0.2	1.35 e-06	35021	335 s
Run 2	0.4	3.47 e-06	29986	289 s
Run 3	0.6	3.01 e-06	32754	327 s
Run 4	0.8	9.85 e-06	27988	280 s
Run 5	1	1.04 e-04	29804	287 s

Table 1: Weights, Loss and Epoch for each run.

	Perceptron 1	Perceptron 2	Perceptron 3	Perceptron 4	Perceptron 5
0	1	1	0.5	0	1
1	1	1	1	0	0
2	0.5	0.5	0	1	0
3	1	1	1	1	1
4	0	0	0	0	1
5	0	0	1	0	0
6	0	0	1	1	1
7	0	0	1	1	0
8	0.5	0.5	0.5	0	0
9	1	1	1	1	0
10	0	0	0.5	0.5	0
11	0.5	0.5	0	1	0.5
12	1	1	0	0.5	0.5
13	0.5	0.5	0	1	1
14	0.5	0.5	0	0	0.5
15	0.5	0.5	0	1	0

Table 2: summarized results from 5 runs

The result looks like binary coding schema for number 0-15, but with 0.5 presenting. We could see that each number receives a different "code" from 5 perceptrons. Since we are using 5 perceptrons in the middle layer and constant initial weights, we shouldn't expect a perfect binary coding schema which requires only 4 perceptrons in the middle layer or setting initial weights for one nodes to 0. Also, since we choose sigmoid function as activation, the coding schema is not necessarily binary which justify the 0.5 in the result.

Using different initial weights, we can see the loss at convergence and number of iteration to convergence is different. It is no clear that how loss at convergence and epoch related to the initial weights and it might depend on how close the suggested initial weight to the optimum

weight (which we have no information about) is. However, the loss at convergence are small enough that we can confidently state the reconstruction is successful.

Using different initial weights at each run, the result seems to be varies a little bit but still comprehendible and result in the table 2 above. One possible explanation for small variation is that since gradient is a local search, by setting up different initial weights, we could approach to different local minimum or more likely, approach to the same local minimum from different direction. Since I set the program to terminate when a small enough gradient is reached rather than require exactly 0 (reason mentioned in part 0), the result minimum may not be the same if approach from different direction.

- **PART 2: 3 Layer NN with 4 perceptrons in hidden layer, sigmoid function as activation function:**

The following 2 Tables summarize same information as from PART 1

	Initial Weights	Loss at convergence	Epoch
Run 1	0.2	1.33 e -04	35738
Run 2	0.4	1.24 e -05	39421
Run 3	0.6	2.04 e -04	19784
Run 4	0.8	3.32 e-04	25438
Run 5	1	1.23 e-04	27503

Table 1: Weights, Loss and Epoch for each run.

	Perceptron 1	Perceptron 2	Perceptron 3	Perceptron 4
0	1	0	1	1
1	1	0	1	0
2	1	0	0	1
3	1	1	1	1
4	0	0	0.5	1
5	0	0	1	0
6	0	0	0	0.5
7	0	0	1	1
8	0.5	1	0	0.5
9	1	1	1	0
10	0	0	1	0.5
11	1	1	0	1
12	1	1	1	0
13	0	0	0	1
14	0	0	0	0
15	0	1	0	1

Table 2: summarized results from 5 runs

Similar to that in part 1, the result gives a binary like coding schema for 16 numbers. There are still several 0.5 but in general, the result looks more like a binary coding which is expected because 4 perceptrons are exactly what we need for binary coding of 16 numbers. We still could not get the perfect binary schema because the choice of activation function here.

Another thing worth noticing is that the average loss at convergence for 5 runs using 4 perceptrons are larger compared to that using 5 perceptrons, which is reasonable as the model is less complex here. Also, the loss at convergence is still not too large here so again, the reconstruction is satisfactory. Similarly, little variations are observed due to different initial weight approaches the optimum from different directions.

- **PART 3: 3 Layer NN with 3 perceptrons in hidden layer, sigmoid function as activation function:**

Table 1 summarize the same information as from PART 1. The result from 5 runs are too different to be comprehendible so Table 2 is not reported for this configuration. Possible reasons are discussed later.

	Initial Weights	Loss at convergence	Epoch
Run 1	0.2	5.44 e -04	80851
Run 2	0.4	7.58 e -04	53781
Run 3	0.6	3.47 e -05	89932
Run 4	0.8	1.02 e -04	80435
Run 5	1	2.33 e -04	83210

Table 1: Weights, Loss and Epoch for each run.

The result is still a coding schema for number from 0 – 16. Although the number of iterations took for convergence increases significantly from using 4 perceptrons (note that is almost same for using 4 and 5 perceptrons), the loss at convergence is still at the acceptable level, which suggests that the original input matrix can still be successfully constructed. Similarly, the different starting initial weights approaches the minimum at different distance and from different direction (and possibly to different minimum), so there are some variations in the stats.

Although a coding schema could be found from each run, it differs from other so much that it is impossible to be summarized to a general rule. Also, the generated pattern is not binary like anymore. Since the minimum number of nodes required to generate a binary coding for 16 different number is 4 ($\log_2 16 = 4$), using 3 nodes here might result in more complex structure rather than a binary like coding in previous configurations. It is possible that the complex coding schema has more than one local minimum exists so starting from different initial point end at different local minimum and makes the result from each run different.

- **PART 4: 5 Layer NN with 8,4,8 perceptrons in hidden layer, sigmoid as activation function**

	Initial Weights	Loss at convergence	Epoch	Run Time
Run 1	0.2	9.33 e -07	973480	120 min
Run 2	0.4	2.35 e -06	1137803	167 min
Run 3	0.6	3.04 e -06	1198418	198 min
Run 4	0.8	1.37 e -06	1017544	187 min
Run 5	1	5.23 e -04	743552	145 min

Table 1: Weights, Loss and Epoch for each run.

	P1	P2	P3	P4	P5	P6	P7	P8
0	0	0.5	0	0	0.5	0	1	0
1	1	1	0.5	1	0.5	0.5	0.5	0
2	1	0	1	0	0	0.5	0	0
3	0.5	0	1	0.5	1	0.5	1	0.5
4	0	0	1	1	0.5	1	0.5	0
5	0	1	0	1	0.5	1	0.5	1
6	0	0.5	0	0	1	0	1	0.5
7	0	0.5	1	1	0	1	0	0.5
8	1	1	0	0	0.5	0.5	0.5	0.5
9	0.5	0.5	0.5	0	0.5	0	0.5	0
10	0.5	0.5	1	1	1	0.5	0.5	1
11	1	0	0.5	0	0	0.5	0	1
12	1	1	0	0.5	0	1	0.5	0
13	0.5	0.5	0.5	1	0	0	1	0
14	0	0	0.5	0	1	0.5	0	1
15	0	0.5	1	0	0	0	1	0

	P1	P2	P3	P4
0	0	1	1	0
1	1	0.5	0	1
2	1	1	1	1
3	0	0	0	1
4	1	0	0	1
5	1	1	1	0
6	1	0	0	0
7	0	0	0	0
8	0	1	0	0.5
9	1	1	0	0
10	1	0	1	1
11	0	1	1	0
12	0	0.5	1	1
13	1	0	1	0
14	0	0	1	1
15	0	1	0	0

	P1	P2	P3	P4	P5	P6	P7	P8
0	0	1	0	0	1	0	1	0.5
1	1	1	0.5	0.5	0	0.5	0	0
2	0.5	0	1	0	0	0.5	0	0
3	0	0.5	0	0	0.5	1	0.5	0
4	0	0.5	0.5	0	0	0	0	1
5	0	1	1	1	0	0	0	0
6	1	1	1	0.5	0	1	0.5	1
7	0	0.5	0	0	0	1	1	1
8	1	0.5	0	0	0	0.5	0.5	1
9	1	0.5	1	0	0.5	0	0	0.5
10	0	1	0.5	0	0	0.5	0	0
11	1	0	0	0	0	0.5	0	1
12	0	0	0	0.5	0	0.5	1	0
13	0	0.5	1	1	0.5	0	0	0
14	0.5	0	0	0	0	1	1	1
15	1	1	0	1	0.5	0.5	1	0.5

Table 2: summarized results from 5 runs

Similarly, the result is a coding schema for numbers from 0-15. The numbers are encoded by 2 layers, first with 8 perceptrons and then with 4 perceptron, then decoded using another two layers with 8 perceptrons and 16 perceptrons to reconstruct data. The 4-perceptron layers, compared to the 4-perceptron-layer in previous configuration, is more similar to a standard binary coding.

As mentioned in previous section, the running time and epoch depend on how far the initial weight is to the optimum. And approaching from different direction to the optimum may result in slightly different result.

- **PART 5: 3 Layer NN with 5 perceptrons in hidden layer & 5 Layer NN with 8,4,8 perceptrons in hidden layer, RELU as activation function**

The tables below summarized the result for 5 runs of both configuration. **Except the activation function for the hidden layer, all the other variables are kept same (using RELU at output layer will yield different result, which is discussed later).** The tables summarize the initial weights, lost at convergence and epoch for each run. The 5 runs using RELU yield very similar result with little variations to that using sigmoid function, so the result is not reported again here.

	Initial Weights	Loss at convergence	Epoch	Run Time
Run 1	0.2	1.16 e-06	26745	125 s
Run 2	0.4	2.47 e-06	24154	97 s
Run 3	0.6	3.23 e-06	28643	129 s
Run 4	0.8	7.68 e-06	24372	100 s
Run 5	1	3.55 e-05	25739	134 s

Table 1: Result for 5 runs of 3 layers NN with 5 neurons in hidden layer.

	Initial Weights	Loss at convergence	Epoch	Run Time
Run 1	0.2	3.88 e -05	176524	23 min
Run 2	0.4	8.48 e -06	181932	29 min
Run 3	0.6	3.44 e -05	193035	33 min
Run 4	0.8	1.15 e -06	182043	29 min
Run 5	1	3.55 e -05	141530	21 min

Table 2: Result for 5 runs of 5 layers NN with 4,8,4 neurons in hidden layer.

It seems that the both the number of iteration and run time for RELU is smaller compare to that using sigmoid at this configuration. While the loss at convergence are at the same level and data could be reasonably constructed using both functions. Possible reasons are discussed below:

Since we use gradient descent to find the optimum here, we update the weights at each iteration at the rate proportional to the gradient of the error function, and the gradient is computed by chain rule. Using sigmoid function, multiplying small numbers together would make the number decreases in exponential order, cause the vanishing gradient problem: after large numbers of iteration, almost no signal will flow through the neuron to its weights and recursively to its data. Additionally, if the initial weights are too far from the optimum (which we did not know yet) then most neurons would become saturated and the network will barely learn. RELU, however, does not have the vanishing gradient problem as its derivative is always 1. Also, while sigmoid neurons involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero, which also reduce the computing time reflected in shorter run time.

Since initially I set the condition of "convergence" when gradient is smaller than 10 e-10 rather than require it equals to 0, it might reduce the epochs for the runs with sigmoid as activation function. For the first simple model with only 3 layers, the original model does not require many updates to converge so the advantage might not be obvious here. For the 5 layers NN with more complex structure, the stats show that it significantly reduces the steps to convergence and the run time. One possible conclusion to be made here is that the choice of activation function could be arbitrary at simple model, ReLU will be a better choice if the model requires complex computations and there's completely no information on weights.

However, **if we use ReLU at the output layer, neither model returns satisfactory result and the average loss at convergence at output layer is around 0.5, which suggests that the input was not successful recreated**, and no stable result could be generated from 5 runs for both models. Also, the steps to convergence and run times varies a lot for each run. When the initial weight is set to 0.4, neither model converges to a result before exceeding the maximum number of steps set in configuration. This can more clearly see using the 5-layer model, but both stats are reported below:

	Initial Weights	**Loss at convergence**	**Epoch**	**Run Time**
Run 1	0.2	0.234	17890	80 s
Run 2	0.4	*No data	*No data	*No data
Run 3	0.6	0.176	24537	94 s
Run 4	0.8	0.225	23779	93 s
Run 5	1	0.586	176533	18 min

	Initial Weights	**Loss at convergence**	**Epoch**	**Run Time**
Run 1	0.2	0.537	398720	54 min
Run 2	0.4	*No data	*No data	*No data
Run 3	0.6	0.672	503928	82 min
Run 4	0.8	0.0942	257832	36 min
Run 5	1	0.502	334728	43 min

*N/A: Run 2 exceed maximum number of iteration, set to be 2000000, before convergence.

One possible explanation is that ReLU will simply map the negative weights to 0. So if we get negative weighted sum of the input, after activation function it will be set to 0. Such error is passed to back propagation then again ReLu maps all negative value to 0. That makes the error at each iteration distorted and approaching the optimum impossible. For example, in our case we tried to reconstruct a set of binary code consisting all negative number for 0, ReLU will change all negative value to 0. Although it seems that there's no error, there actually is but the error is removed by ReLU, therefore we would stay at starting point and never approach to the desired weights. Therefore, while ReLU is a better choice for hidden layers, it is not possible to generate satisfactory reconstruction using ReLU as activation function at output layer or get stable learning process.