COSC 522 Machine Learning Project 4 MNIST Digit Recognition Using Multi-Layer Neural Networks

Yangsong Gu

October 24, 2021

1. Introduction

1.1 Objective

The objective of this project is to have a thorough understanding of the traditional multi-layer perceptron and back propagation.

2.Task 1

Go through Nielsen's book on Neural Network and Deep Learning. Ideally, by the end of the semester, you should finish reading at least Chapters 1 through 3. For this project, you should read through Chapter 1 and to answer some of the questions, you also need to selectively read through Chapter 3. Answer the following question in the report after reading. The answer to each question should take less than a third of a page, assuming single space and a font size of 12 Times.

2.1 Batch vs. online processing.

2.1.1 Batch processing

- the parameters e.g., biases and weights, are updated based on a batch of training samples. For instance, the weights and bias of all mini batches (i.e., whole training set) are added up together and averaged before activating sigmoid function.
- when batch size is 1, the batch learning becomes online learning
- the batch learning is not fit for huge dataset training since it has to deal with huge data samples each time.

2.1.2 online processing

- update parameters like weights and biases immediately after processing each training sample.

- It could speed up the convergence or slow down the convergence. As one of online learning approach, wta cost much more time than kmeans in image compression (Project 3). This issue is prone to happen when algorithm is feed with bad data (i.e., outliers).
- it needs a learning rate to update parameters. If the learning rate is too high, the algorithm will be difficult to converge. Conversely, the algorithm can converge while it is time-consuming if learning rate is too small.
- compared to batch learning, it is more time effective.

2.2 Gradient descent vs. stochastic gradient descent

The goal of gradient descent is to compute the descent ∇C of training data. The differences are:

2.2.1 Gradient descent

- use the loss and derivative of all training samples.
- It becomes time-consuming when handling huge training set. It is slower than Stochastic gradient descent.
- gradient descent can obtain the optimal solution.

2.2.2 stochastic gradient descent

- use a small sample of randomly chosen training samples.
- it can speed up the convergence when dealing with a large number of training samples.
- by averaging this small portion of training set each time, it turns out that we can quickly get a good estimate of true gradient.
- due to the random selection, the solution of stochastic gradient descent may keep oscillating around the true solution.

Mini-batch processing is an approach that dealing with a small portion of training samples each time.

2.3 Perceptron vs. sigmoid neurons

2.3.1 perceptron

- a perceptron is a single layer neural network.
- can only do binary classification.
- it is similar to linear classifier generated from weighted features.
- the perceptron makes decisions by weighing up evidence whose rule can be formulated as:

$$output = \begin{cases} 0 \; if \; wx + b < 0 \\ 1 \; if \; wx + b \geq 0 \end{cases}$$

where b can be viewed as bias or threshold.

2.3.2 sigmoid neurons

- sigmoid neurons have some same qualitative behavior as perceptron, for instance, they can figure out how changing the weight and biases will change the output. In comparison, the small changes in weights and bias cause only a small change in sigmoid function output.
- the output of a sigmoid neuron is $\sigma(wx + b)$ where σ is defined by:

$$\sigma(z) = \frac{1}{1 + e^{-z}} = 1/(1 + \exp(-\sum_j w_j x_j - b))$$

- **Above sigmoid function has two advantages, one is that it is differentiable, which made the gradient descent possible. The other is that it has as output any real number between 0 and 1.**

2. 4 Feedforward vs. backpropagation

2.4.1 feedforward

- output from one layer is used as input to the next layer.
- No loops in the network, the information is always fed forward, never fed back.
- A single-layer perceptron is the simplest feedforward network.
- different from recurrent neural networks
- the input is transformed and forwarded.

2.4.2 backpropagation

- A fast algorithm for computing the gradient of the cost function with respect to parameters
- the problem is essentially “how to choose weight w to minimize the error E between expected output and the actual output.
- the basic idea behind backpropagation is gradient descent and chain rule.
- the error (i.e., difference between target and output) is back propagated

2.4.3 Why do we need to introduce “bias” when training a neural network?

- for a perceptron, th bias is a measure of how easy it is to get the perceptron to fire.
- Bias is a constant which helps the model in a way that it can fit best for the given data.

- bias can be used to delay the triggering of the activation function.

3. Task 2

3.1 Task 2.1

Figure 1 demonstrates the accuracy and loss of test set by applying Neilson's model as baseline. We found that the accuracy starts to converge after 20 epochs and the converged accuracy is around 94.7%.

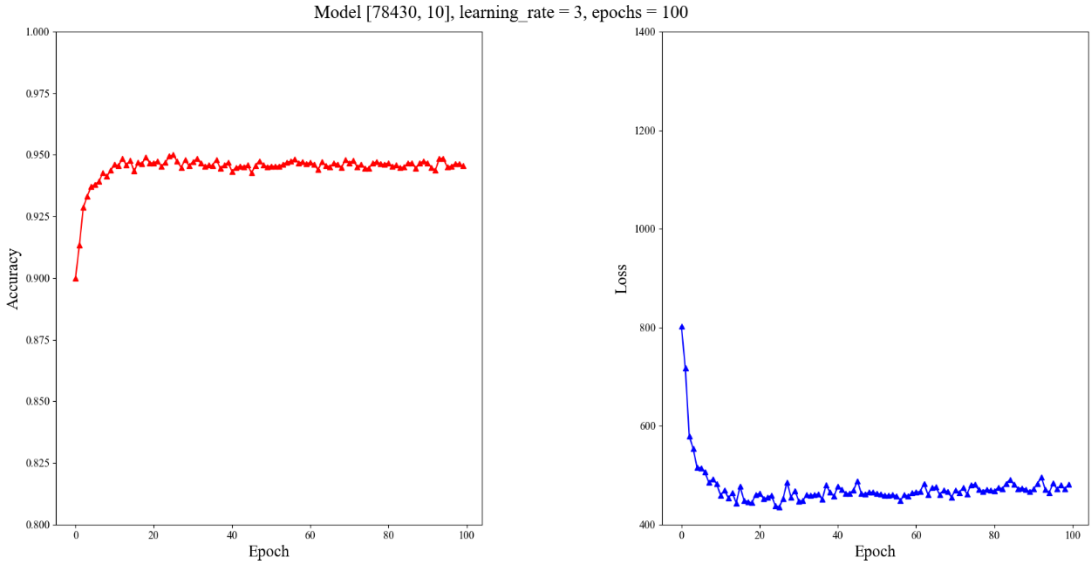

Figure 1. Neilson baseline model net = [784,30,10], learning rate = 3, mini-batch size = 10, epochs = 100

3.2 Task 2.2 Effects of hyperparameters

3.2.1 Task 2.2.1 Effects of network structure.

To untangle the influence of network structure on model accuracy, firstly, we control the learning rate = 1, mini-batch size = 15. Based on that, I created six different networks with different neurons of hidden layer and different number of hidden layers. For each network, I set 100 epochs to train the model. Here are the model specs.

Table 1 Model accuracy of different network structure.

	Network structure	Learning rate	Batch size	Optimal accuracy
NN 1	[784, 256, 128, 64, 10]	1	15	(epoch = 77, acc =98.05%)
NN 2	[784,60,60,10]	1	15	(epoch = 42, acc = 97.52%)
NN 3	[784,64,128,256,10]	1	15	(epoch = 62, acc = 95.75%)
NN 4	[784,30,30,30,10]	1	15	(epoch = 39, acc = 94.90%)

NN 5	[784,60,60,60,10]	1	15	(epoch = 28, acc = 97.12%)
NN 6	[784,30,30,10]	1	15	(epoch = 87, acc = 94.62%)

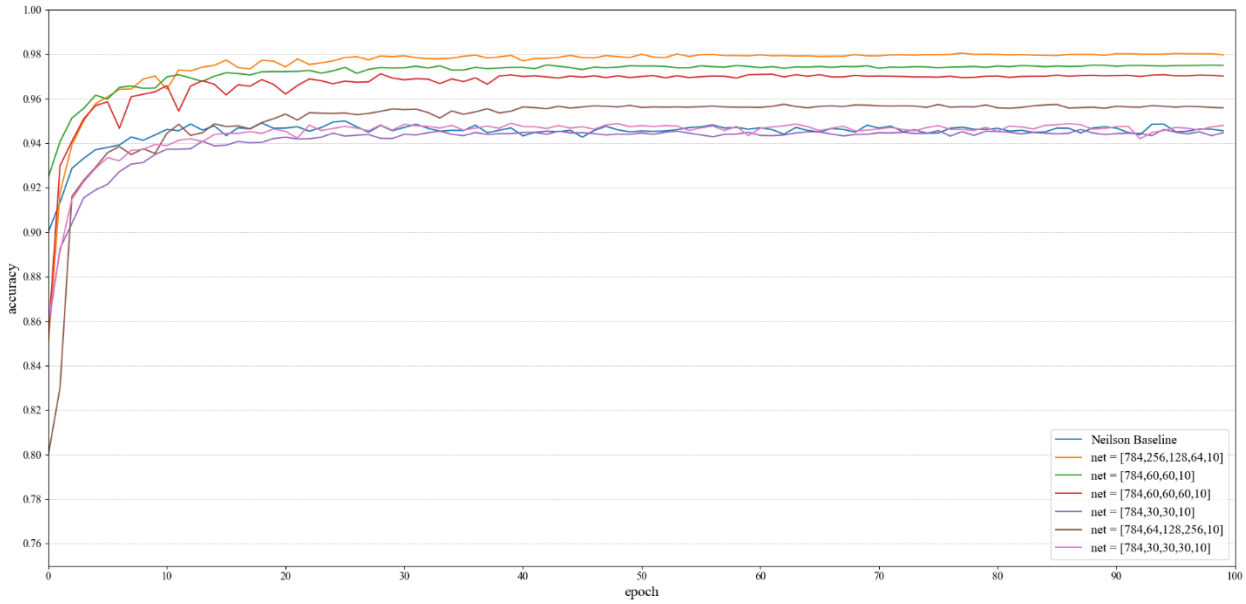

Figure 2. Epochs vs. accuracy of different network structure.

In addition to the table summary, Figure 1 visualizes the convergence curve of different network structures. The blue line shows the baseline of Nielson model (i.e., network = [784,30,10], learning rate = 3, mini-batch size = 10). From the figure 2, we can see that different network structures produced different convergence speed and point. For instance, the orange one (i.e., network = [784, 256, 128, 64, 10]) which outperforms other network structures starts to converge around 10 epochs and finally oscillates around 98%. From the component of network structures, this one is most complex since it comprises 3 hidden layers and largest number of neurons in the first hidden layer. With such complex network, the different combinations of weights and biases can be explored to large extent, leading to a high accuracy. In addition, it is worth noting that the network [784, 64, 128, 256, 10] with and increasing number of hidden neurons, does not gain much in accuracy, as the brown curve shows in Figure 3. The total number of hidden neurons are same as the best network structure, which means they creates the same complexity for computation. The result suggests that setting a higher number of neurons of the first layers could feed more information from input to next hidden layers and at the same time, setting a lower number of neurons of last layers could reduce the disturbance of making decisions.

Finally, as above mentioned, the optimal network structure is [784, 256, 128, 64], which will be implemented in the following sections.

3.2.2 Task 2.2.2 Effects of mini-batch size

Using the bet network structure identified in previous section (i.e., NN1 {784, 256, 128, 64, 10})), I set the different batch size which are 20, 25, 30, 60, 75, 100 in addition to 15. Table 2 summarizes the model accuracy of different batch size. The red row hints the optimal model whose mini-batch size is 20, which can generate the highest test accuracy (98.1%) at epoch 51.

Table 2 Model accuracy of different mini-batch size based on optimal network structure

	Network structure	Learning rate	Mini-batch size	Optimal accuracy
NN1	[784, 256, 128, 64, 10]	1	15	(epoch = 77, acc =98.05%)
NN2	[784, 256, 128, 64, 10]	1	20	(epoch = 51, acc =98.10%)
NN3	[784, 256, 128, 64, 10]	1	25	(epoch = 93, acc =97.07%)
NN4	[784, 256, 128, 64, 10]	1	30	(epoch = 99, acc =94.28%)
NN5	[784, 256, 128, 64, 10]	1	60	(epoch = 95, acc =97.88%)
NN6	[784, 256, 128, 64, 10]	1	75	(epoch = 87, acc =97.83%)
NN7	[784, 256, 128, 64, 10]	1	100	(epoch = 89, acc =97.71%)

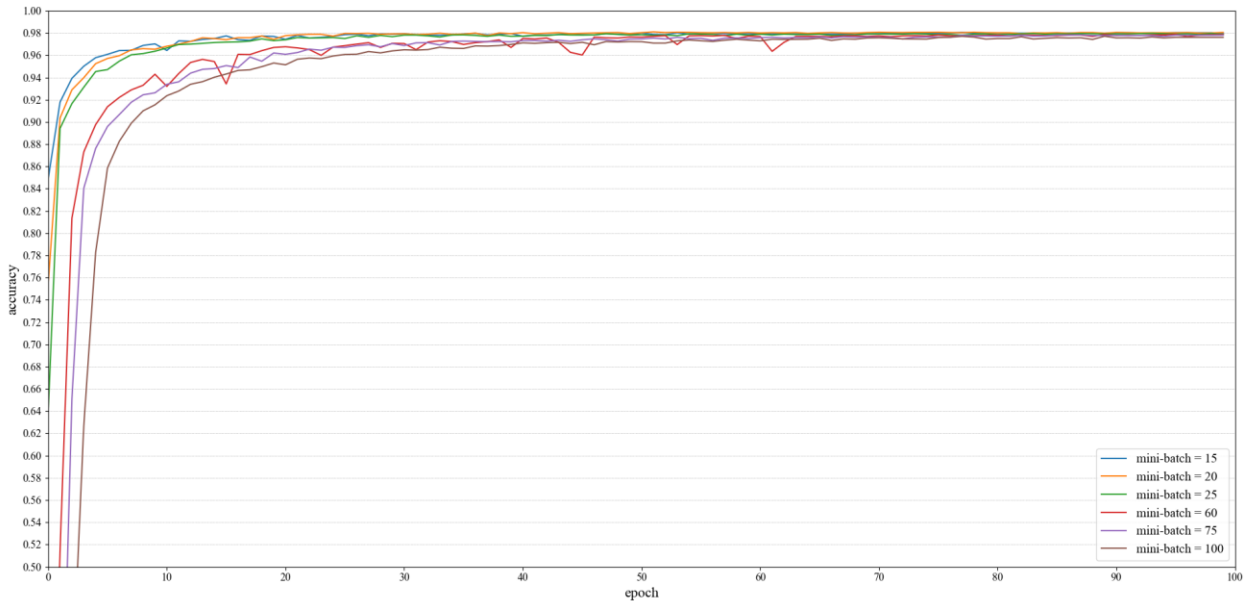

Figure 3.1 Epochs vs. accuracy of different mini-batch size

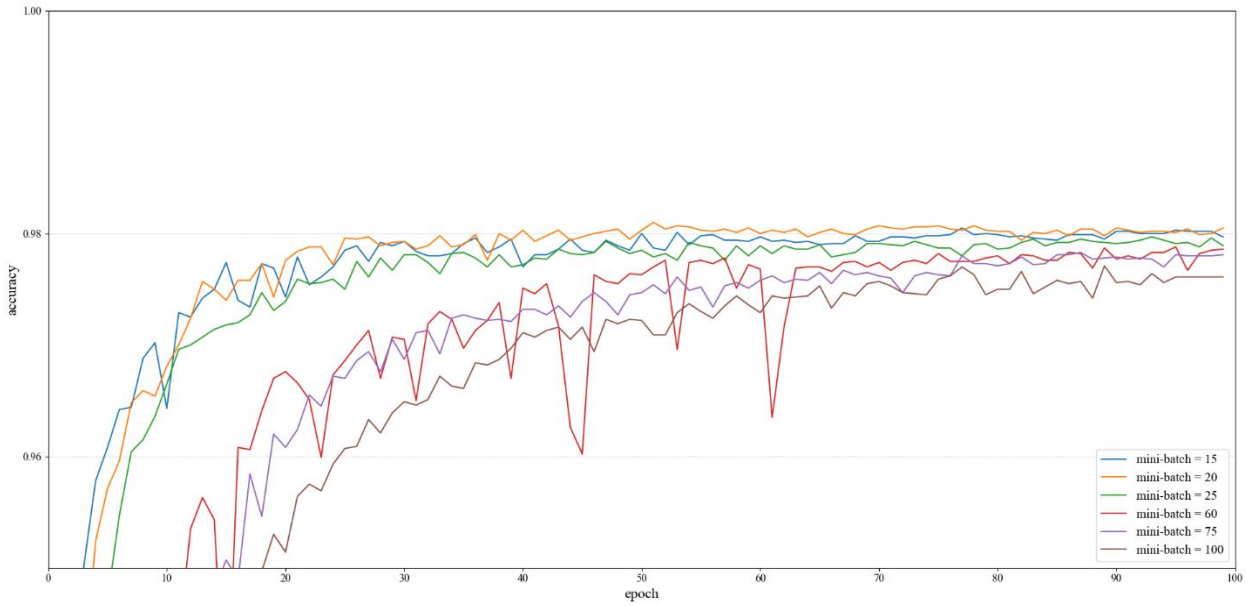

Figure 3.2 Epochs vs. accuracy of different mini-batch size (zoom in at tail.)

Figure 3.1 (original) and 3.2 (zoom in) visualize the convergence curve of different mini-batch size based on identified optimal network structure. Strategically, I not only choose the small learning rate such as 0.001, but also some large numbers such as 100. Although those curves converge at similar accuracy, it should be noticed that 1). Small mini-batch size (e.g., 15, 20, 25) starts to converge earlier than large mini-batch size (e.g., 30, 60, 75, 100), as their curvature indicates 2). When mini-batch size is 20, the model owns the highest accuracy. 3) the curves become more and more fluctuated as the mini-batch size increases.

In the end, the optimal batch size is 20 based on the optimal network structure determined in previous section.

3.2.3 Task 2.2.3 Effects of learning rate

Table 3 summarizes the model accuracy of different learning rate based optimal model determined in previous sections. The best model is marked as red color of which the corresponding learning rate is 3. This model can obtain 98.18% accuracy on test set.

Table 3 Model accuracy of different learning rate based on optimal network structure and batch size.

	Network structure	Learning rate	Mini-batch size	Optimal accuracy
NN1	[784, 256, 128, 64, 10]	1	20	(epoch = 51, acc =98.10%)
NN2	[784, 256, 128, 64, 10]	0.1	20	(epoch = 49, acc = 96.1%)

NN3	[784, 256, 128, 64, 10]	0.01	20	(epoch = 49, acc = 21.05%)
NN3	[784, 256, 128, 64, 10]	0.001	20	(epoch = 2, acc = 11.35%)
NN4	[784, 256, 128, 64, 10]	3	20	(epoch = 49, acc = 98.18%)
NN5	[784, 256, 128, 64, 10]	10	20	(epoch = 39, acc = 98.05)
NN6	[784, 256, 128, 64, 10]	100	20	(epoch = 0, acc = 9.8%)

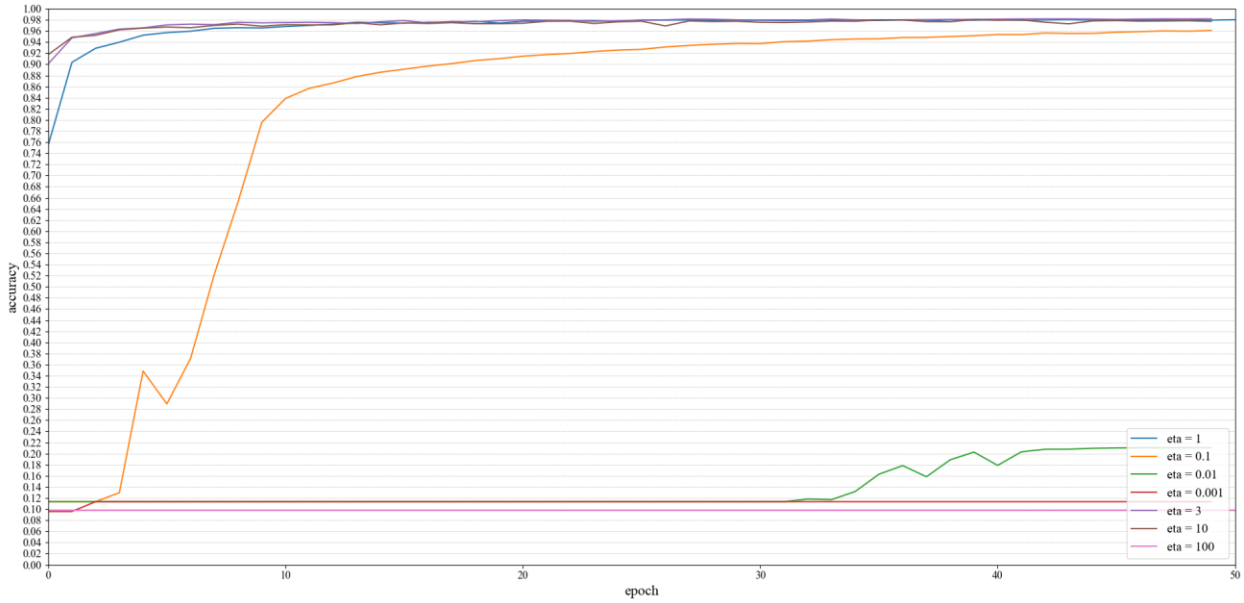

Figure 4.1 Epochs vs. accuracy of different learning rate

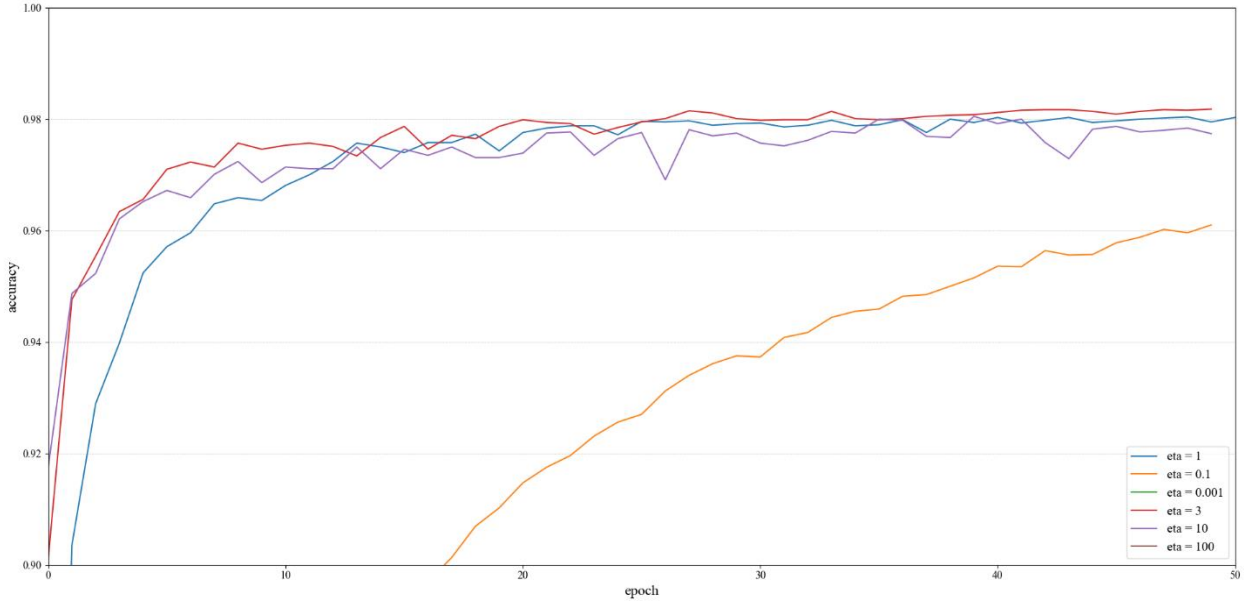

Figure 4.2 Epochs vs. accuracy of different learning rate (zoom in at tail)

To better understand the convergence path, Figures 4.1 and 4.2 were plotted to track the impact of different learning rate on convergence. Similar to mini-batch size setting, I take three small values (i.e., 0.1,0.001, 0.01) and 2 medium number (i.e., 3, 10) and an extreme large learning rate 100. Figure 4.1 reveals that model with 0.01, 0.001, and 100 produce the relatively low accuracy throughout all epochs. In addition, model with learning rates 3 and 10 have a faster convergence speed than learning rate 1, as figure 4.2 shows.

In the end, the optimal model is determined as network = [784,256,128,64,10], mini-batch size = 20, and learning rate is 3. This model will be further used in applying momentum based gradient descent algorithm.

3.3 Task 2.3 XOR classification

Use the network library to implement the XOR gate. You have experienced in HW4 that single-layer perceptron cannot realize XOR since the decision boundary is not linear in the 2-d space. Going through the practices you had in Task 2.2 and show again three figures on the effects of different hyperparameters. Fix mini-batch size at 1.

To implement the neural network on XOR gate, the training set is prepared as Table 4 shows. At the same time, we set the test set same as training set. Before injecting the training and test set into neural networks, I only vectorized the training labels to ensure the correct input of evaluation function.

Table 4. Training set of XOR gate

Training input	Training label
[0,0]	0
[1,0]	1
[0,1]	1
[1,1]	0

3.3.1 Task 2.3.1 Effect of network structure

Inspired from previous test, I set the number of neurons of hidden layer in a descending order such as [16,8], [32,16,8], [64, 32, 16, 8, 4] and so on. Not surprisingly, the convergence speed increases as the number of neurons of first hidden layer and number of hidden layer increases. The orange line in Figure 5 outperforms rest models. To have a quantitative impression on how these network structures make difference, I list the number of 100% classification cases in table 5.

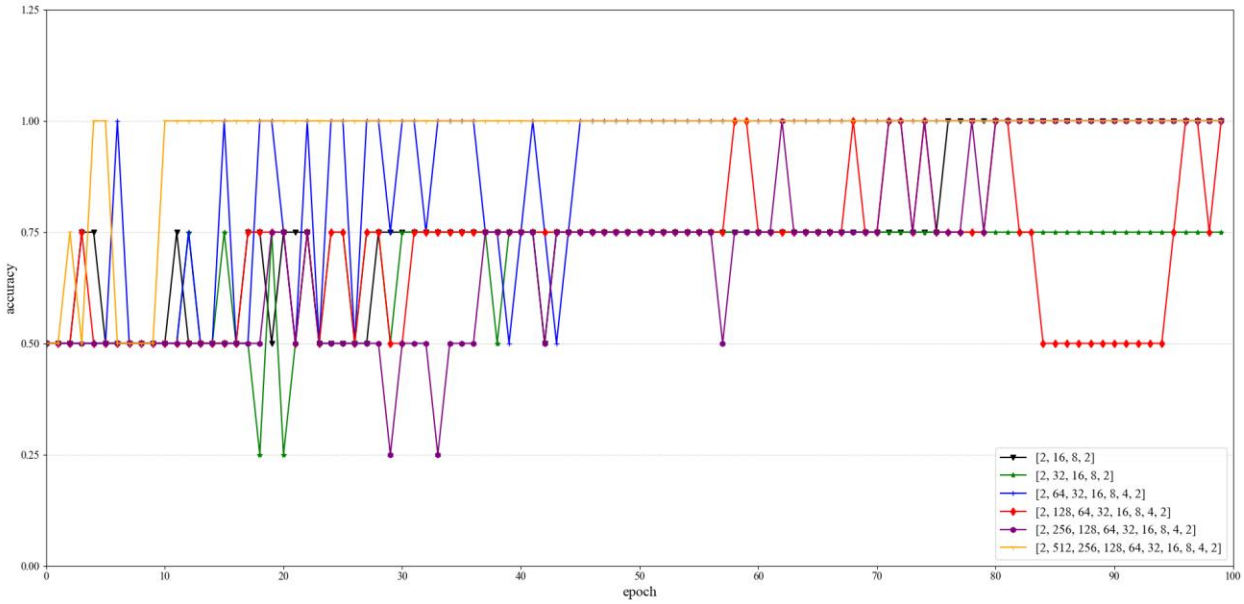

Figure 5 epochs vs. accuracy of different network structures

Table 5 network structure

Network	Structure	# of 100% classification cases
NN1	[2,16,8,2]	24/100
NN2	[2, 32, 16, 8, 2]	0/100
NN3	[2, 64, 32, 16, 8, 4, 2]	71/100
NN4	[2, 128, 64, 32, 16, 8, 4, 2]	11/100
NN5	[2, 256, 128, 64, 32, 16, 8, 4, 2]	25/100
NN6	[2, 512, 256, 128, 64, 32, 16, 8, 4, 2]	92/100

before explaining this table 5, I observed that: 1) even for the same network, the performance would change as I refresh the code chunk. That is because of the initialization of weights and biases vary as the refresh and these changed swinged the performance. 2) No matter how I

refreshed the code, the model with higher number of neurons in first hidden layer would have better performance. It can be generalized that the model built by decreasing number of neurons in hidden layers could gain the accuracy effectively.

With the help of table 5, we can further identify that model NN6 [2,512, 256, 128,64, 32,16,8,4,2] is the best model under the scenario that learning rate is 1 and mini-batch size is 1.

3.3.2 Task 2.3.2 Effect of learning rate.

Based on the optimal network structure, the model was implemented on different learning rates 0.1, 0.01, 0.001, 0.5, 1, 3,100. Table 5 and Figure 7 indicates that learning rate =1 could have much more 100% correct cases, thus the optimal model learning rate among these candidates is 1. Note that for two sides of extreme values (i.e., 0.001, 0.01 or 100), they don't have any 100% correct cases and their classification accuracy stays at 50%.

Table 5 network structure

Network	Structure	Learning rate	# of 100% classification cases
NN1	[2, 512, 256, 128, 64, 32, 16, 8, 4, 2]	0.001	0/100
NN2	[2, 512, 256, 128, 64, 32, 16, 8, 4, 2]	0.01	0/100
NN3	[2, 512, 256, 128, 64, 32, 16, 8, 4, 2]	0.1	39/100
NN4	[2, 512, 256, 128, 64, 32, 16, 8, 4, 2]	1	92/100
NN5	[2, 512, 256, 128, 64, 32, 16, 8, 4, 2]	3	75/100
NN6	[2, 512, 256, 128, 64, 32, 16, 8, 4, 2]	10	0/100
NN7	[2, 512, 256, 128, 64, 32, 16, 8, 4, 2]	100	0/100

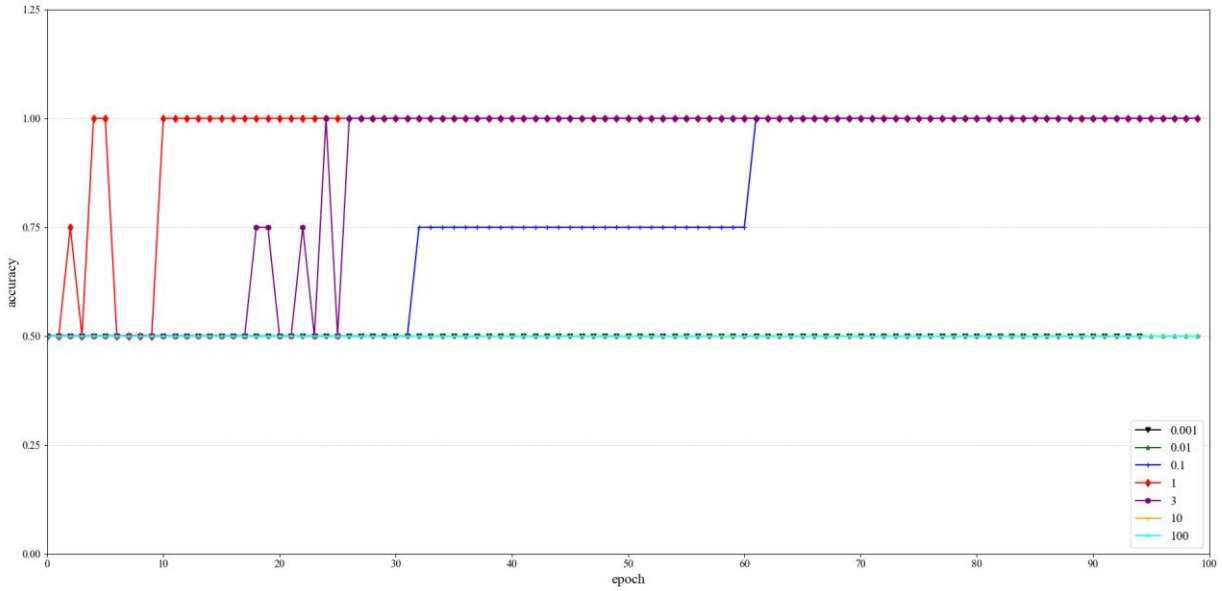

Figure 6. epochs vs. accuracy of different learning rate.

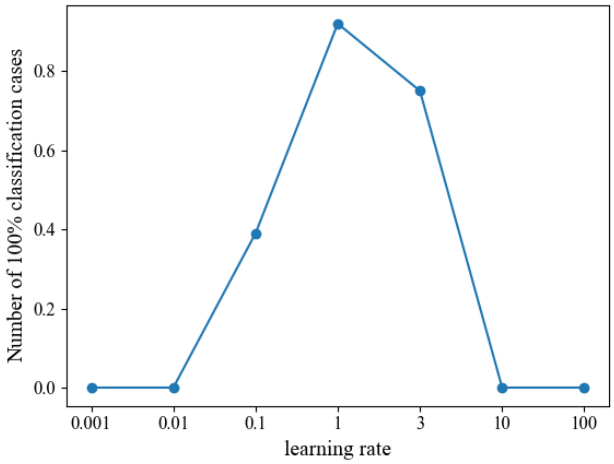

Figure 7. learning rate vs. 100% correct classification cases.

3.3.3 Task 2.3.3 Effect of random initialization vs. symmetric initialization

In Neilson's book, he initializes the weights and biases based on N (0,1) distribution which generates similar number of positive and negative numbers. To compare this symmetric weights and biases with random initialization, I added an initializer generating the weights and biases following uniform distribution. Specifically, the uniform distribution [-1/sqrt(x), 1/sqrt(x)] will produce the symmetric values of weights and biases, while the uniform distribution [-1/sqrt(x), 0] and [0, 1/sqrt(x)] will generate either all negative or positive numbers.

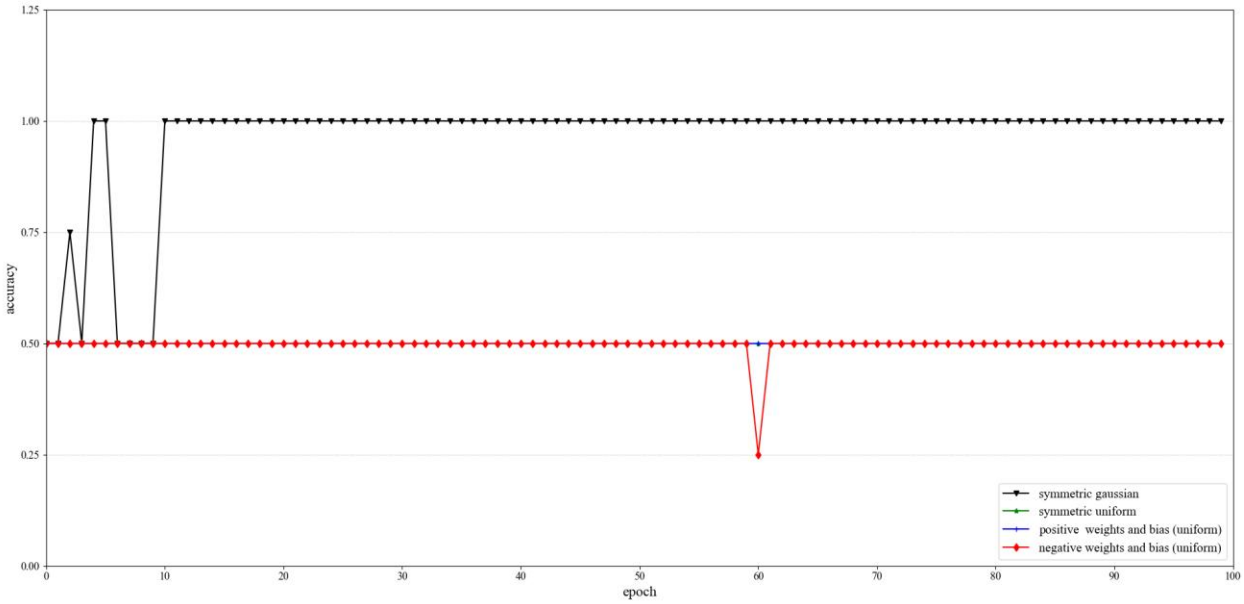

Figure 7 Random initialization vs. symmetric initialization.

Figure 7 demonstrates the classification results of using random initialization and symmetric initialization. The symmetric distribution (gaussian and uniform) still outcompetes than other distribution. In addition to that, Gaussian initialization outcompetes the uniform initialization, as we can see that the accuracy of uniform case retains 50%.

Table 6. symmetry vs. random initialization

Network	Structure	Learning rate	# of 100% classification cases	Max accuracy
Gaussian	[2, 512, 256, 128, 64, 32, 16, 8, 4, 2]	1	92/100	1
Uniform	[2, 512, 256, 128, 64, 32, 16, 8, 4, 2]	1	0/100	0.5
Uniform (positive)	[2, 512, 256, 128, 64, 32, 16, 8, 4, 2]	1	0/100	0.5
Uniform (negative)	[2, 512, 256, 128, 64, 32, 16, 8, 4, 2]	1	0/100	0.5

3.3.4 Task 2.3.4 Comments on XOR

Why it is difficult to train the smaller network?

- The smaller network is vulnerable to outliers. Specifically, the backpropagation would be influenced by these outliers. However, if training large dataset, the impact of outliers can be eliminated by other samples.
- The smaller network is vulnerable to initial weights and biases. From above tests, I found that once the initial weights and biases are not reasonable, then the model would not converge. For instance, three uniform cases retain 50% accuracy throughout the epochs.

How to improve it when using BPNN for classification

- Previous experiments already show that we should have more neurons at the first hidden layers, and fewer neurons on last hidden layers. Generally, we should have a descending order of number of neurons of hidden layers.

  The reason is that when we inject the input data, the first hidden layer which accepts the input should try to preserve most of potential information, which needs more neurons. In contrast, when we compute the output, we should have less neurons in hidden layers,

otherwise it will add the disturbance of making final decision. This finding can be verified through comparing Figure 8 bad cases and Figure 5 good cases.

Recall the procedure of implementing BPNN

- When creating the different network structure, we can start with hidden layers such as [8, 4], [16,8,4], [32,16,8,4], etc… gradually increase the front hidden layers' neurons. The number of neurons is largely depended on the number of inputs. After that, the model should be decided looking at the corresponding accuracy. Note that with the increasing layers and number of neurons, the model becomes more and more complex, so when we see the gain is not that big from model to model, we should stop as soon as possible.
- After deciding the best neural network, we try to find a optimal mini-batch size.
- Then, we can try different learning rate from small to large. The value of learning rate is also up to the scale and number of inputs.
- Last but not least, to increase the model accuracy, we can try ensemble learning, that is, we can randomly sample the training set and choose the average or the majority of vote to be the final classification result

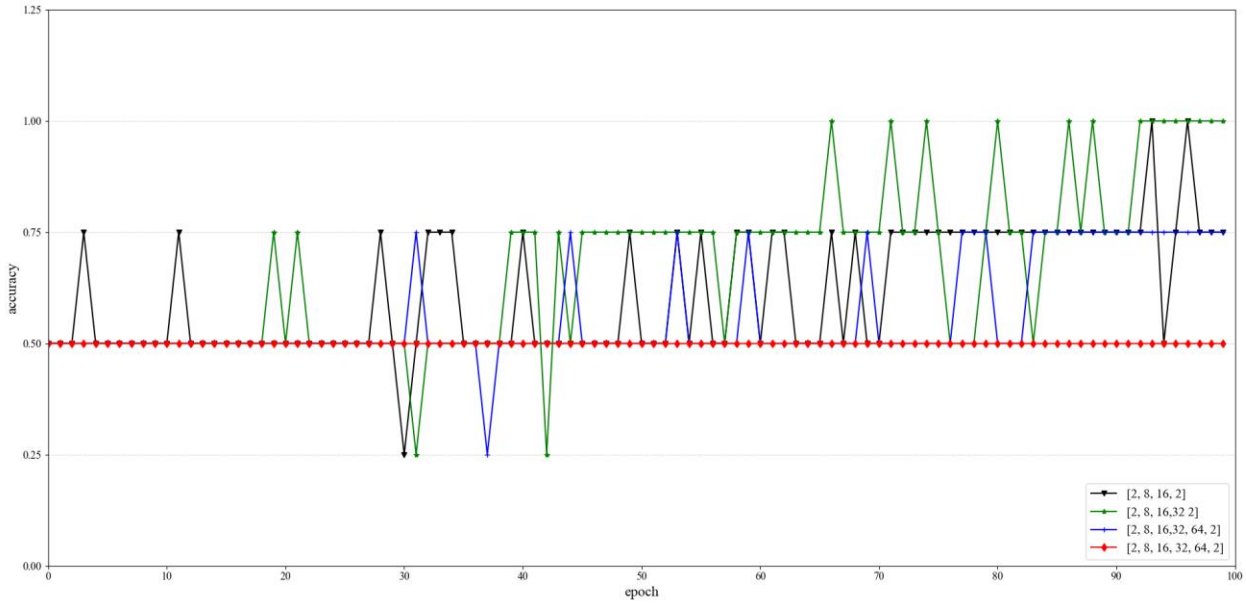

Figure 8 Bad cases, ascending neurons of hidden layers.

4. Bonus Question.

Bonus: Implement "Momentum" for faster and more stable convergence.

The momentum modified the standard SGD by introducing velocity v and momentum μ which tries to control the velocity and prevents overshooting valleys while allowing faster descent as well as giving Gradient descent a short-term memory.

$$v_t = \mu v_{t-1} \quad (1-\mu)\nabla_\theta J(\theta)$$

$$\theta = \theta - \eta v_t$$

where the momentum μ is suggested to set as 0.5.

Therefore, the core modification of code is to rewrite "*update_mini_batch*" function. The revised code is snapped in below table. In addition, we need to initialize the velocity of weights (*self.weights_vel*) and biases (*self.biases_vel*) at the beginning of creating instance.

```
def update_mini_batch_momentum(self, mini_batch,eta):
    # apply momentum to update weights and biases
    # ys g 6:20PM 10/30/2021
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
     mu = 0.9
    self.weights_vel = [mu * v - (eta/len(mini_batch)) * nw for v, nw in zip(self.weights_vel, nabla_w)]
    self.biases_vel = [mu * v - (eta/len(mini_batch)) * nw for v, nw in zip(self.biases_vel, nabla_b)]
    self.weights  = [w  + v for w, v in zip(self.weights, self.weights_vel)]
    self.biases = [b + v for b, v in zip(self.biases, self.biases_vel)]
```

```
# for momentum based gradient descent
self.weights_vel = [np.zeros((y, x)) for x, y in zip(self.sizes[:-1], self.sizes[1:])]
self.biases_vel = [np.zeros((y, 1)) for y in self.sizes[1:]]
```

Figure 9.1 and 9.2 display the convergence curve of standard gradient descent and momentum based gradient descent approaches. Their final converged results are close to each other. It is worth noting that from epoch 27 to 39, the accuracy of momentum based is higher than standard gradient descent.

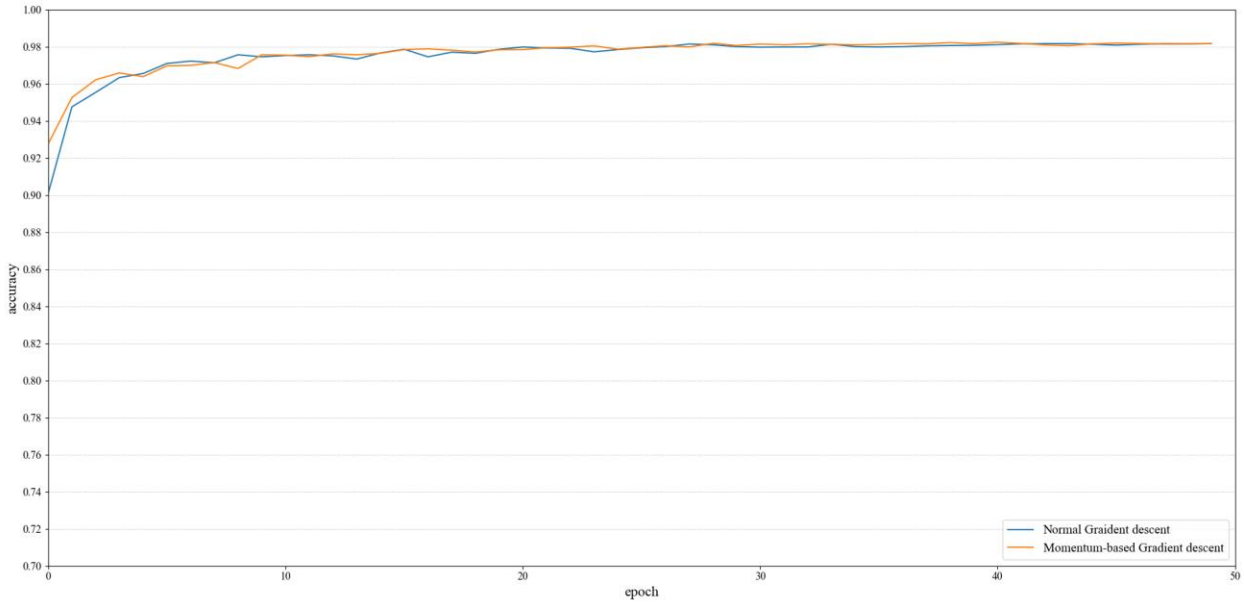

Figure 9.1 Momentum based gradient descent (digit identification)

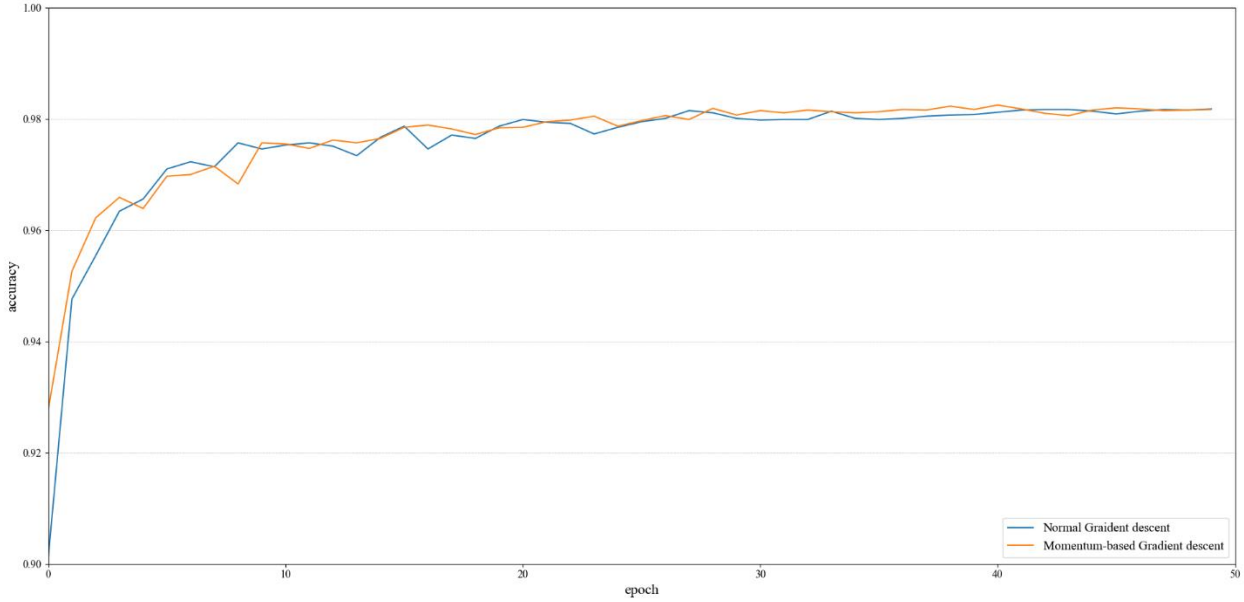

Figure 9.2 Momentum based gradient descent (digit identification, zoom in at tail)

Figure 10 demonstrates the accuracy curve of momentum based gradient descent approach compared to standard gradient descent. We can see that momentum-based approach converges faster than standard gradient descent. The former converged at 7^{th} epoch while the later converged at 11^{th} epoch.

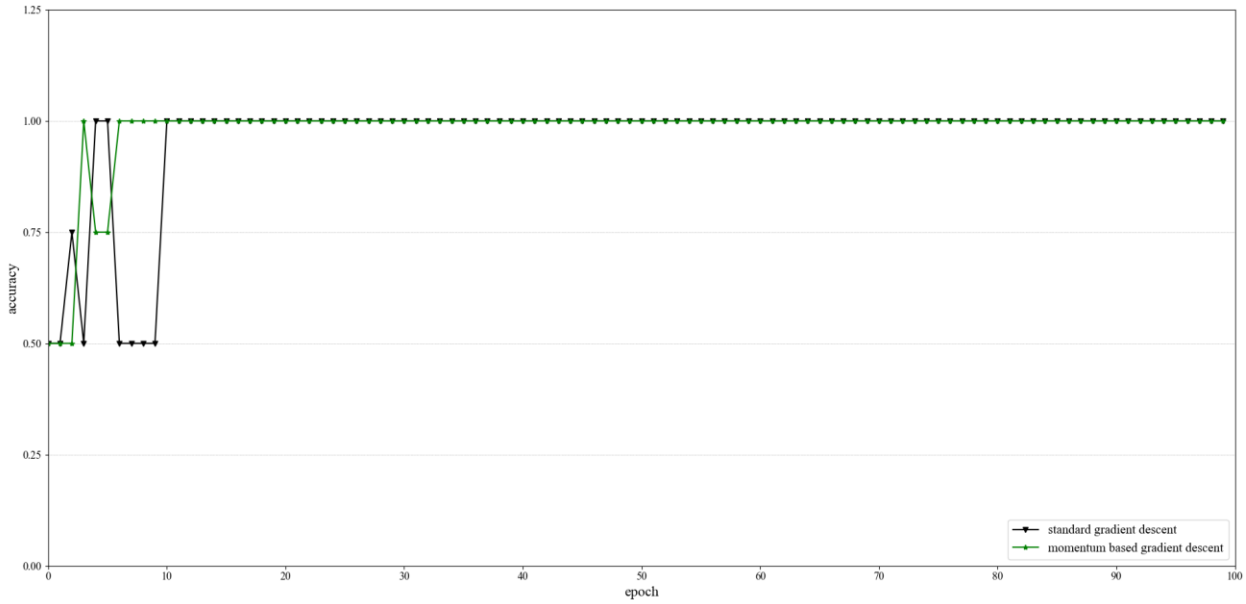

Figure 10 Momentum based gradient descent (XOR classification)