Assignment 2
CIFAR-10 Image Classification using Fully Connected Neural Network

CS 519 - Deep Learning
Eugene Seo (OSU ID: 932981978)

February 14, 2017

Problem 1. Write a function that evaluates the trained network (5 points), as well as computes all the subgradients of W_1 and W_2 using backpropagation (5 points)

- Evaluation (5 points)

Algorithm 1 : Evaluation

```
class SigmoidCrossEntropy(object):
  def crossEntropy(self, x, y, w1, w2, l2_penalty=0.0):
    # cross entropy loss
    E = -np.sum(y * np.log(x) + (1.0 - y) * np.log(1-x)) / y.shape[0]
    # regularization
    E += 0.5 * l2_penalty * (np.linalg.norm(w1) + np.linalg.norm(w2))
    return E

  def evaluate(self, x, y, w1, w2, l2_penalty=0.0):
    prob = self.sigmoid(x) # P(y=1)
    E = self.crossEntropy(prob, y, w1, w2, l2_penalty) # objective loss
    performance.append(E)

    y_hat = 1 * (prob >= 0.5) # class prediction
    accuracy = 1 - (np.sum(y_hat ^ y) / y.shape[0]) # error rate
    performance.append(accuracy)
    return performance
```

For evaluation, cross entropy loss function is used to measure the error of prediction, and error rate is also calculated to check the ratio of correct classification as metric of accuracy.

cross entropy loss (E) =

$$-(y * \mathbf{log} z_2 + (1 - y) * \mathbf{log}(1 - z_2))$$

error rate (Accuracy) =

$$\frac{\text{number of correct classification}}{\text{number of examples}}$$

- Backpropagation (5 points)

Algorithm 2 : Backpropagation

```
class LinearTransform(object):
     def backward(self, grad_output):
        return np.dot(grad_output, self.W.T)

class ReLU(object):
     def backward(self, grad_output):
        gradient = (self.x > 0) * 1.0
        gradient[np.where(self.x==0)] = 0.5
        return gradient*grad_output

class SigmoidCrossEntropy(object):
    def backward(self, grad_output):
        return (self.prob - self.y)

class MLP(object):
  def train(self, x_batch, y_batch, learning_rate, momentum, l2_penalty):
        ...
    # backpropagation
    gradient3 = self.SCE.backward(0)
    gradient2 = self.LT2.backward(gradient3)
    gradient1 = self.ReLUf.backward(gradient2)

    # weight update
    delta_w2 = np.dot(z1.T, gradient3)
    delta_w1 = np.dot(x_batch.T, gradient1)
    self.LT2.update(delta_w2, learning_rate, momentum, l2_penalty)
    self.LT1.update(delta_w1, learning_rate, momentum, l2_penalty)
```

There are three gradient functions of each linear transform(f), Relu(g), and sigmoid cross entropy(E) functions. Loss function can be represented with feed forward functions as below.

$$
\begin{aligned}
E &= -(y * \mathbf{log}\sigma(f_2) + (1-y)\mathbf{log}(1-\sigma(f_2))) \\
f_2 &= \mathbf{W}_2^T g + c \\
g &= max(0, f_1) \\
f_1 &= \mathbf{W}_1^T \mathbf{x} + b
\end{aligned}
$$

The derivative of each functions is implemented based on its own differential formula. The derivative of the combined sigmoid entropy functions, $gradient3$, is

$$\frac{\partial E}{\partial f_2} = z_2 - y$$

The derivatives of linear transform function wrt. g, $gradient2$, is

$$\frac{\partial f_2}{\partial g} = \mathbf{W}_2$$

The derivatives of Relu functions, $gradient1$, is

$$\frac{\partial g}{\partial f_1} = \begin{cases} 1, & f_1 > 0 \\ [0,1], & f_1 = 0 \\ 0, & f_1 < 0 \end{cases}$$

To calculate the delta of each weight vectors, we compute $\frac{\partial E}{\partial \mathbf{W}_2}$ and $\frac{\partial E}{\partial \mathbf{W}_1}$ and update the weights.

$$
\begin{aligned}
\frac{\partial E}{\partial \mathbf{W}_2} &= \frac{\partial E}{\partial f_2}\frac{\partial f_2}{\partial \mathbf{W}_2} \\
&= (z_2 - y)g \\
\frac{\partial E}{\partial \mathbf{W}_1} &= \frac{\partial E}{\partial f_2}\frac{\partial f_2}{\partial g}\frac{\partial g}{\partial f_1}\frac{\partial f_1}{\partial \mathbf{W}_1} \\
&= (z_2 - y)\mathbf{W}_2^T g'\mathbf{x}
\end{aligned}
$$

Problem 2. Write a function that performs stochastic mini-batch gradient descent training (5 points). You may use the deterministic approach of permuting the sequence of the data. Use the momentum approach described in the course slides.

- Stochastic mini-batch gradient descent training (5 points)

Algorithm 3 : Stochastic mini-batch gradient descent

```
if __name__ == '__main__':
  for epoch in xrange(num_epochs):
    randList = np.arange(num_examples)
    np.random.shuffle(randList)
    batches = randList.reshape((num_batches, int(num_examples/num_batches)))

    for b in xrange(num_batches):
      x_batch = train_x[batches[b],:]
      y_batch = train_y[batches[b],:]
      total_loss = mlp.train(x_batch, y_batch, lr, momentum, l2_penalty)
```

For stochastic mini-batch gradient descent training, we need to divide whole examples into the subset of mini batches. In my implementation, I first randomly generate list of order, *randList* (instead of shuffling examples), then divide the list with the defined number of batches. Then, each example of batches is executed according to the randomly generated order from the shuffled list.

- Momentum (5 points)

Algorithm 4 : Momentum

```
class LinearTransform(object):
    def update(self, delta, learning_rate=1.0, momentum=0.0, l2_penalty=0.0):
        regulization = l2_penalty * self.W
        delta = delta + regulization
        self.velocity = momentum * self.velocity - learning_rate * delta
        self.W += self.velocity
```

Whenever updating weights for every batches, I apply the momentum factor to control the weight changes along with the learning rate.

Problem 3-6. 3) Train the network on all the training examples, tune your parameters (number of hidden units, learning rate, mini-batch size, momentum) until you reach a good performance on the testing set. What accuracy can you achieve? (20 points based on the report). 4) Training Monitoring: For each epoch in training, your function should evaluate the training objective, testing objective, training misclassification error rate, testing misclassification error rate (5 points). 5) Tuning Parameters: please create three figures with following requirements. Save them into jpg format:
i) test accuracy with different number of batch size: batch-test accuracy.png
ii) test accuracy with different learning rate: lr-test accuracy.png
iii) test accuracy with different number of hidden units: hidden units-test accuracy.png
6) Discussion about the performance of your neural network.

I first tuned learning rate, which is the most important to get to the local minimum, and then tuned mini-batch size, hidden units, momentum, and l2 penalty respectively in order to train the model. Each section, I put the range of test parameter in [...], and the rest predefined values of other parameters. I used 100 epoches for all experiments.

- **Tuning learning rate**

  learning_rate = [1e-06, 5e-06, 1e-05, 5e-05, 1e-04]

  num_batches = 1000
  hidden_units = 10
  momentum = 0.8
  l2_penalty = 0.001

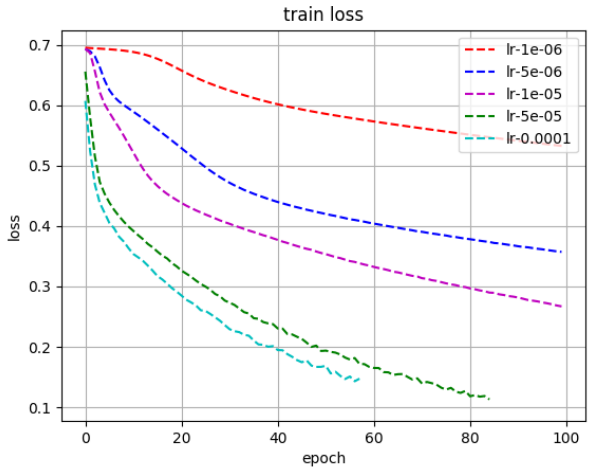

Figure 1: Train Loss

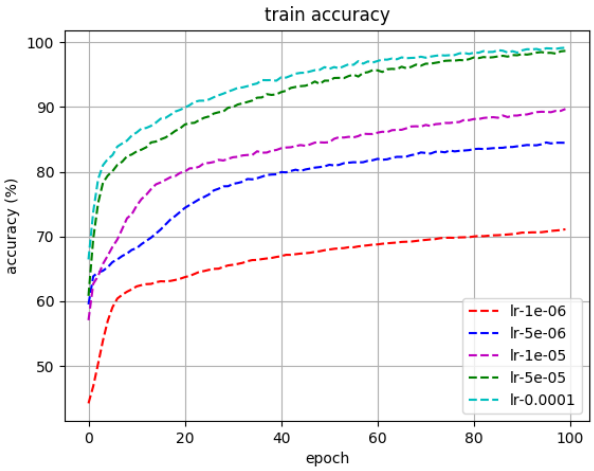

Figure 2: Train Accuracy

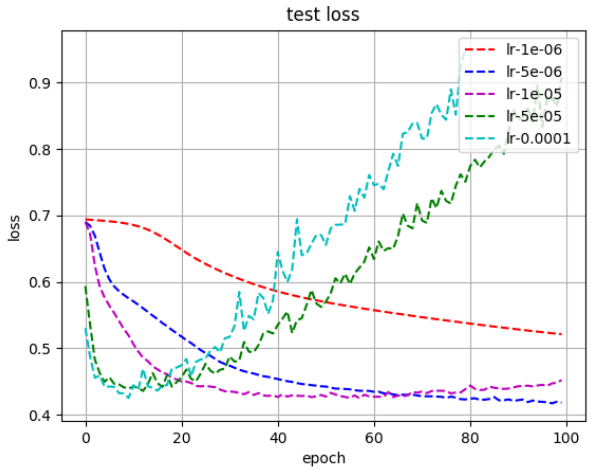

Figure 3: Test Loss

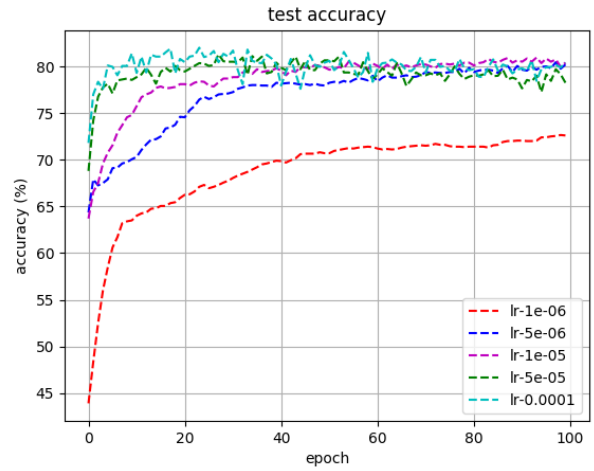

Figure 4: Test Accuracy

Analysis: Any other learning rates which are higher than 0.0001 are excluded in this experiment after observing their fluctuation without convergence. So, I found that learning rate below 0.0001 can make our model get to the local minimum, and tested which value is the most effective to obtain high accuracy. In the graph of train loss (Fig. 1), we see that as the learning rate is getting smaller, it converges very slowly. We also see that the learning rates, 0.0001 and 5e-05, are guarantee to converge on training data (Fig. 1), but both generate unstable test loss and accuracy (Fig. 3-4). Therefore, I choose 1e-05 as the learning rate in my model because it let the model to converge in a stable way and generate high test accuracy.

- **Tuning mini-batch size**

  num_batches = [5, 10, 50, 100, 500]

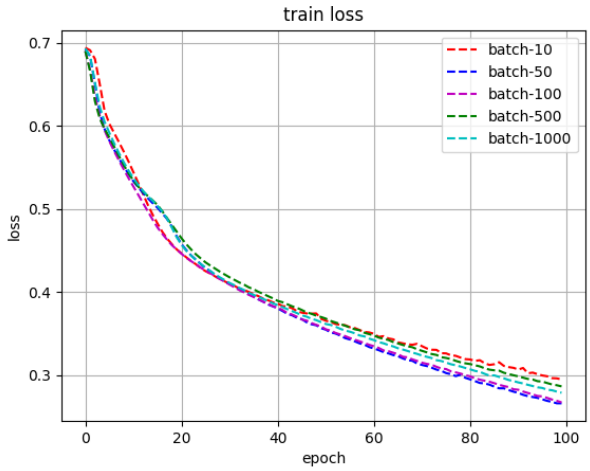

Figure 5: Train Loss

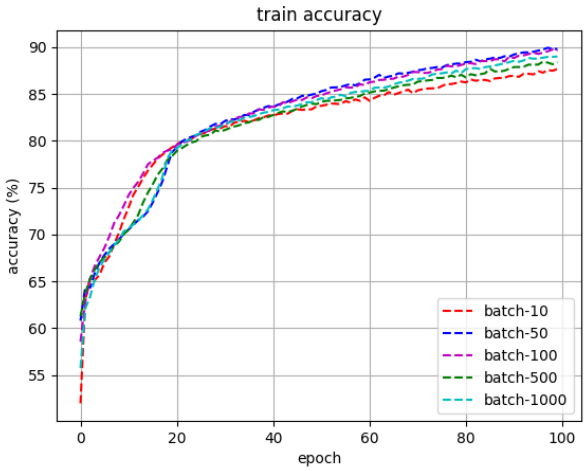

Figure 6: Train Accuracy

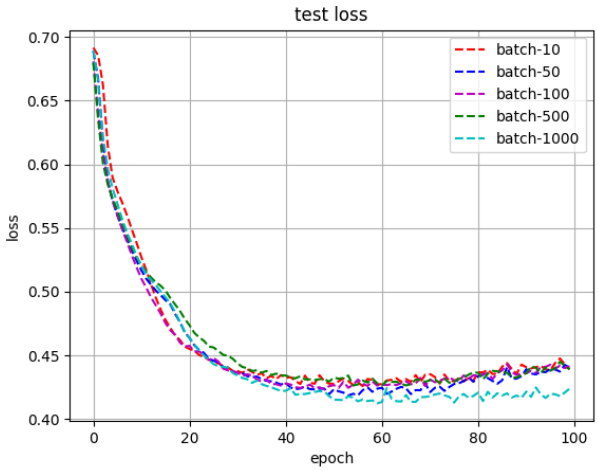

Figure 7: Test Loss

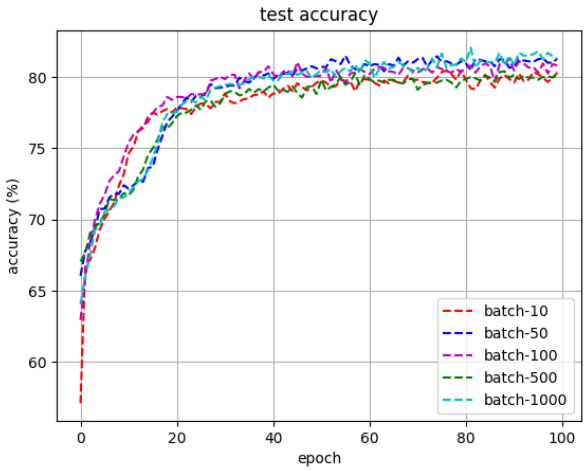

Figure 8: Test Accuracy

learning_rate = 1e-05
hidden_units = 10
momentum = 0.8
l2_penalty = 0.001

Analysis: The size of mini batches, surprisingly, does not significantly affect the loss and accuracy for both training and testing (Fig. 5-8). Rather, it influences time performance as it is related with high dimensional computation. As shown in Table 1, extreme choices of the mini batch size such as 10 or 1000, require higher computation time. It happens because mini batch size 10 has to deal with 1000-dimensional matrix computation, mini batch size 1000 has larger iterations of learning although it only deals with 10 samples per a batch. I think this experiment shows that the strong point of stochastic minibatch approach because instead of learning whole examples at one time, we can learn a subset of them in a saved time, and we still can obtain reasonable results. Therefore, I chose mini-batch size with 50 since it shows efficient time performance without significantly deteriorating the test accuracy.

Mini Batch Size	Test Accuracy(%)	Time Cost(s)
10	80.45	477.2032
50	81.50	144.3120
100	81.25	148.8222
500	80.40	177.9812
1000	82.05	214.2329

Table 1: Test accuracy and time cost with different mini batch size

- **Tuning the number of hidden units**

hidden_units = [5, 10, 50, 100, 1000]

learning_rate = 1e-05
num_batches = 50
momentum = 0.8
l2_penalty = 0.001

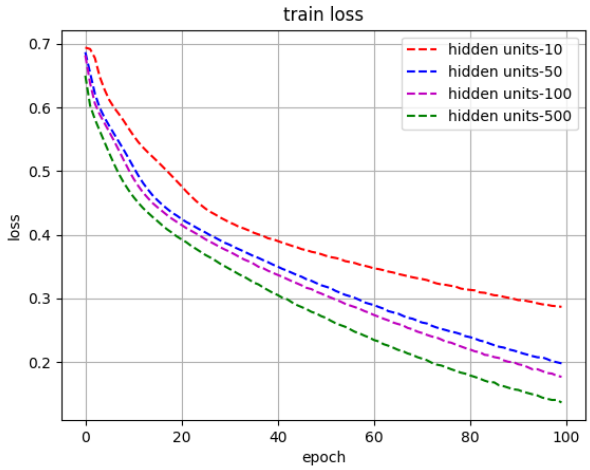

Figure 9: Train Loss

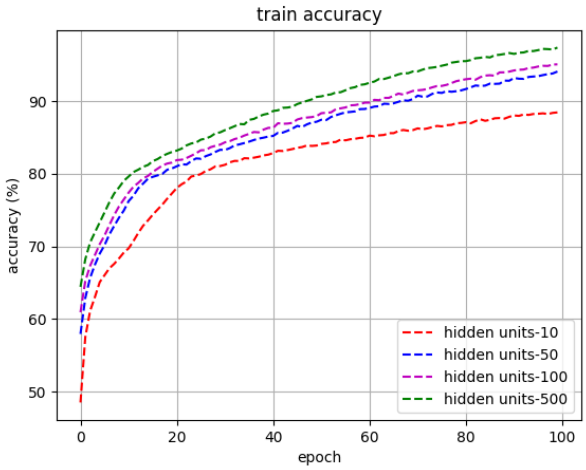

Figure 10: Train Accuracy

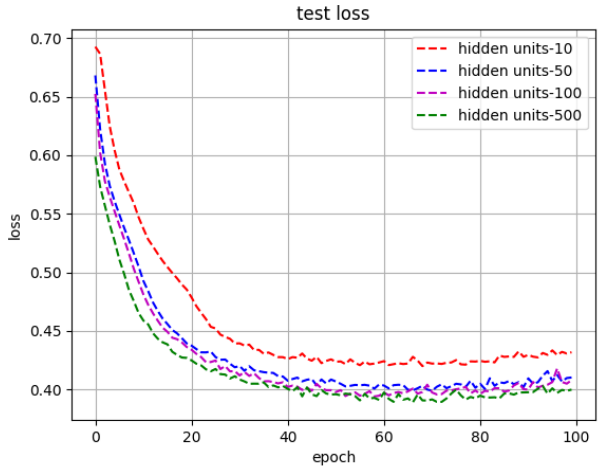

Figure 11: Test Loss

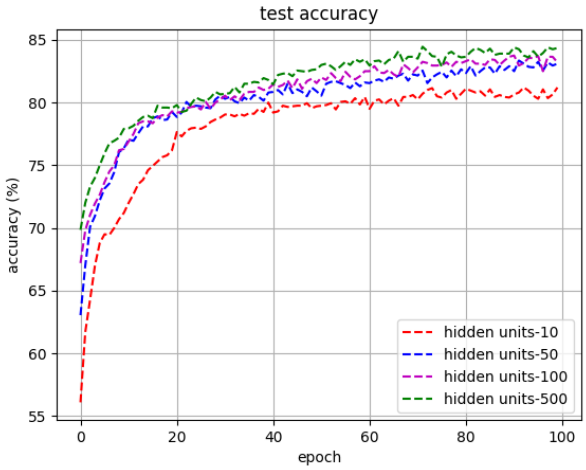

Figure 12: Test Accuracy

Analysis: The number of hidden layer units is the most influential parameter to obtain higher test accuracy. From the experiments with different number of hidden layer units, we see that test accuracy keeps increasing as the number of hidden units increase (Fig. 12). It reveals that this image classification can obtain higher accuracy with more a sophisticated neural network model. However, large number of hidden units significantly affects time performance, and from a certain point, the high number of hidden units does not improve test accuracy anymore. Therefore, we need to carefully chose the number of hidden units as considering both computing power and the mount of improvement. In my experiment, 500 would be the good choice for test accuracy if computing resource is allowed, otherwise, unit number 50 is still showing reasonable test accuracy with 500, so, I chose 50 as hidden unit number for the rest training part.

Number of Hidden Units	Accuracy(%)	Time Cost(s)
10	81.20	144.2799
50	83.35	701.02990
100	83.75	1080.9208
500	84.45	3025.8877
1000	84.30	5130.4885

Table 2: Test accuracy and time cost with different number of hidden units

- **Tuning momentum**

momentum = [0.0, 0.6, 0.7, 0.8, 0.9]

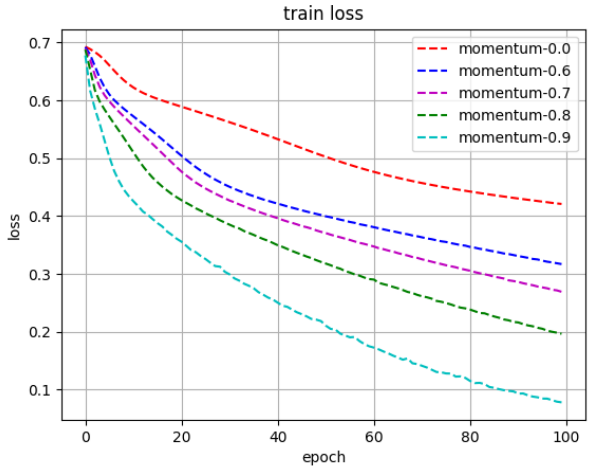

Figure 13: Train Loss

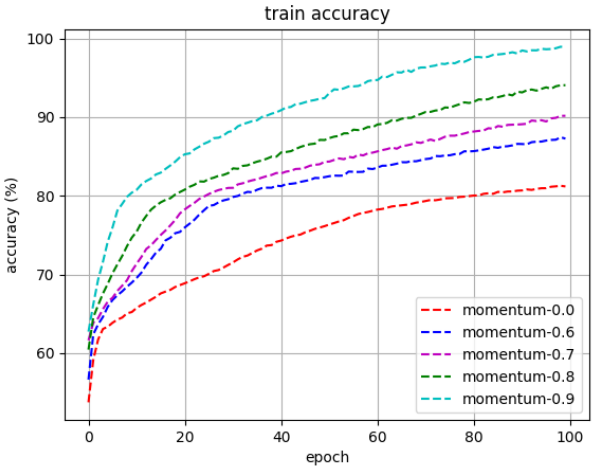

Figure 14: Train Accuracy

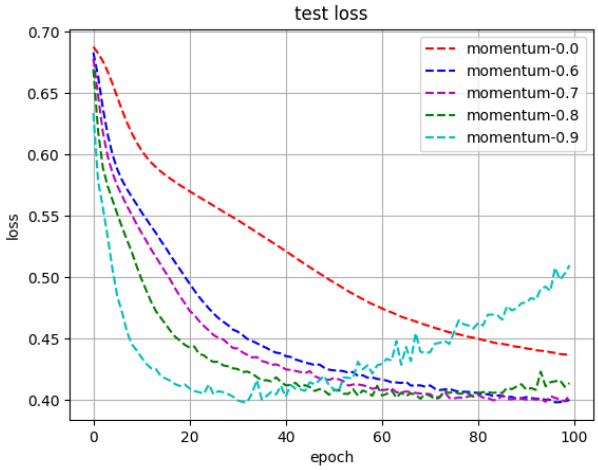

Figure 15: Test Loss

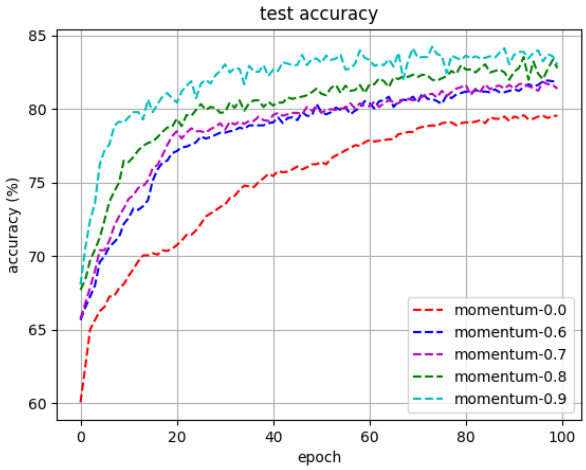

Figure 16: Test Accuracy

learning_rate = 1e-05
num_batches = 50
hidden_units = 50
l2_penalty = 0.001

Analysis: The momentum is an important factor to control the weight changes and make the model converge faster, avoiding gradient decent oscillation along with a learning rate. In my experiments, I tested momentum values from 0.6 to 0.9 and 0.0. In the result graphs (Fig. 13-16), it shows that as momentum values increase until 0.9, it expedites to converge for both training and testing. However, in the Fig. 15, the momentum value, 0.9 and 0.8, shows a little effect of overfitting, going up from a certain point. Therefore, I choose momentum with 0.7 since it shows robustness of test accuracy and faster convergence.

- **Tuning l2 penalty**

l2_penalty = [0.0, 0.001, 0.01, 1, 10]

learning_rate = 1e-05
num_batches = 50
hidden_units = 50
momentum = 0.7

Analysis: L2 penalty plays a role of preventing overfitting and increasing test accuracy. In my experiment, very high penalty like 10 is not a good choice because it deteriorates both train and test accuracy (Fig. 17-20). In fact, any other tested values of L2 penalty shows very a little improvement in test accuracy and loss (Table 3). Although the improvement is very trivial, I choose l2 penalty with 1, which shows the highest accuracy among them.

L2 penalty	Test Accuracy (%)
0.0	82.20
0.001	82.35
0.01	82.25
1	82.80
10	81.25

Table 3: Test accuracy with different L2 penalty

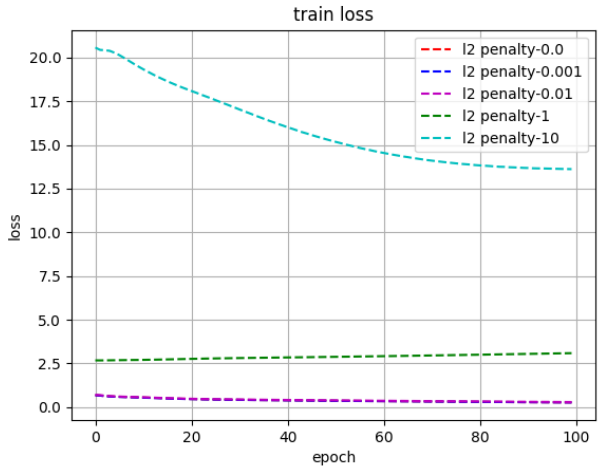

Figure 17: Train Loss

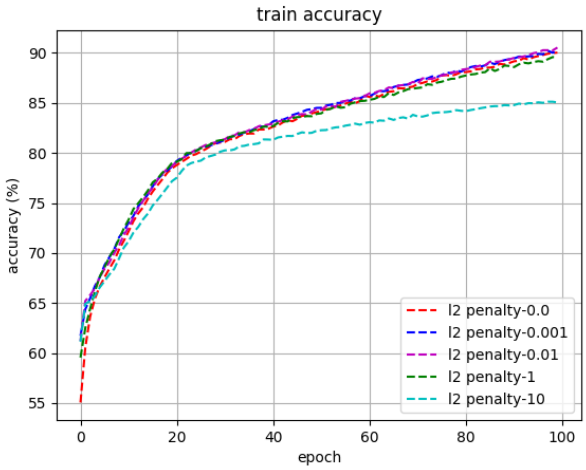

Figure 18: Train Accuracy

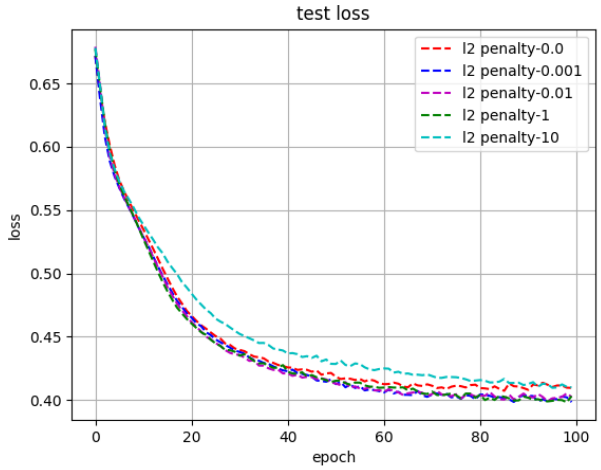

Figure 19: Test Loss

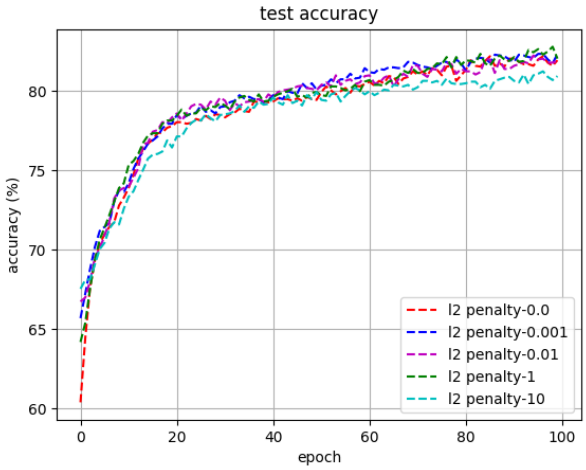

Figure 20: Test Accuracy

In conclusion, tuning parameters with appropriate values is very important to train the model in fast and efficient time as well as to obtain higher test accuracy. Tuning learning rate, momentum is important in the sense of guaranteeing convergence into the local minimum. The proper size of both mini batch and hidden unit is also important to improve time performance and test accuracy. Both hidden unit size and L2 penalty should be well chosen to increase test accuracy as preventing overfitting problem.

Finally, I chose parameter values as below to train and evaluate my model.

learning_rate = 1e-05
num_batches = 50
hidden_units = 50
momentum = 0.7
l2_penalty = 1

- **The performance of my neural network**

  *** What accuracy can you achieve? 83.15%**

  I finally trained my model with tuned parameters, and obtained 83.15% test accuracy. (although I could increase the accuracy up to 84.65% with 500 hidden units.) Fig. 21 shows the test accuracy of train and test data, and until 100 epochs, overfitting did not happen. Fig. 22 shows the objective error (loss) of train and test, and train error is higher than test error due to the regularization factor. In sum, using L2 penalty parameter (=1), I can prevent overfitting problem, improving test accuracy and loss.

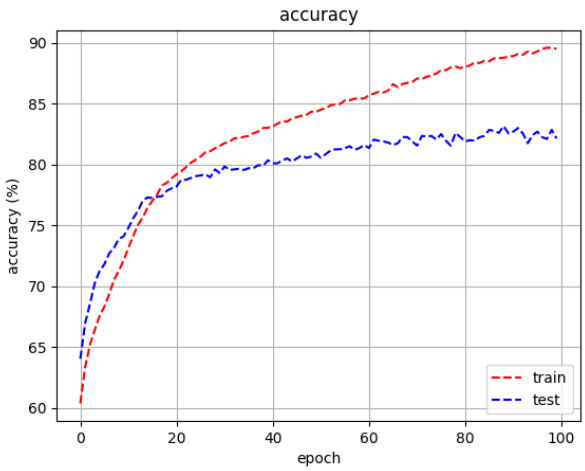

Figure 21: Train and Test Accuracy

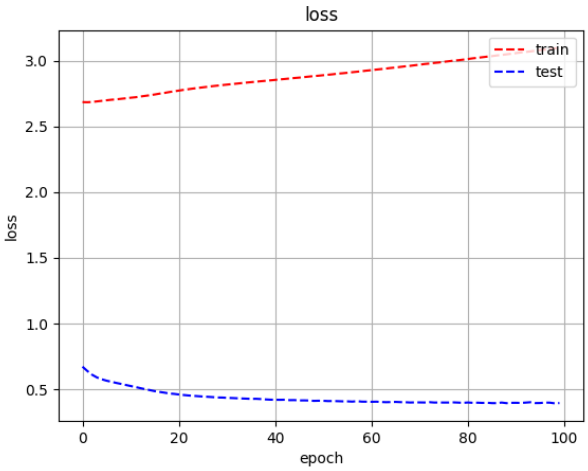

Figure 22: Train and Test Loss