# Deep Learning CS69000 Homework 3

Tunazzina Islam, islam32@purdue.edu

February 2020

## Q0

1. Student interaction with other students / individuals:

   (c) No, I did not discuss the homework with anyone.

2. On using online resources:

   (b) I have used online resources to help me answer this question, but I came up with my own answers.

   Here is a list of the websites I have used in this homework:

   - pytorch forum discussion
   - https://gluon.mxnet.io/chapter06_optimization/
   - https://towardsdatascience.com/understanding-the-scaling-of-l%C2%B2-regularization-in-the-context-of-neural-networks-e3d25f8b50db
   - https://towardsdatascience.com/coding-neural-network-regularization-43d26655982d
   - https://ruder.io/optimizing-gradient-descent/index.html#momentum
   - https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/

## Q1 (2.5 pts)

1. Yes, we can use Robbins-Monro stochastic approximation algorithm. When the gradient estimator is unbiased, this algorithm can directly approximates the zero of a function defined as an expectation with respect to a distribution depending on a parameter. Using an unbiased gradient estimate, we can typically expect substantial improvement over optimization driven by finite differences.

   While the gradients had a bias, we have cheap estimator which appears to be correlated with the expensive validation error.

2. A well-known issue with the Robbins–Monro procedure, however, is numerical stability and sensitivity to specification of hyperparameters, especially the learning rate. If the condition (a) is not satisfied that means we have large learning rates which can make the iterates diverge

numerically. If condition (b) is not satisfied, we have small learning rates which can make the Robbins–Monro iterates converge very slowly.

3. Flatter region is better model because it means very large puddle. Flatness is a measure of how sensitive network performance is to perturbations in parameters. Flat minima have training accuracy that remains nearly constant under small parameter perturbations. The stability of flat minima to parameter perturbations can be seen as a wide margin condition. When we add random perturbations to network parameters, it causes the class boundaries to wiggle around in space. If the minimizer is flat, then training data lies a safe distance from the class boundary, and perturbing the class boundaries does not change the classification of nearby data points.

4. Early Stopping is used to avoid overtraining because if we test hypothesis too many time eventually it looks good but it's not a good hypothesis. we need to stop early. When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but validation set error begins to rise again. This means we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error. Every time the error on the validation set improves, we store a copy of the model parameters. When the training algorithm terminates, we return these parameters, rather than the latest parameters. The algorithm terminates when no parameters have improved over the best recorded validation error for some pre-specified number of iterations.

   Early stopping requires a validation set, which means some training data is not fed to the model. The error on the validation set is used as a proxy for the generalization error in determining when overfitting has begun. So early stopping should never use the training data.

5. Vectorization of image patches leave the output of the filter rotation-sensitive.

   The changes of max pooling affect the downsampling of the outputs of the convolution. Max pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. Ideally, we would like patches to be permutation-invariant. Max pooling is learnable permutation-invariant.

   Maximum pooling calculates the maximum, or largest, value in each patch of each feature map. The results are down sampled or pooled feature maps that highlight the most present feature in the patch.

6. Min pooling is a pooling operation that calculates the minimum, or the smallest, value in each patch of each feature map. Min pooling selects the darker pixels from the image. It is useful when the background of the image is light and we are interested in only the darker pixels of the image. Otherhand, ReLU activation layer applies $ReLU(x) = max(0, x)$ to all of the values in input volume which is opposite of what min pooling does and we will not be able to pick the darker pixels from the image. This is why min-pooling is never applied to CNNs with ReLU activations.

## Task 1a: Minibatch Stochastic Gradient Descent (SGD)

### Task 1a: 3. (a))

To implement the training part using minibatch SGD, we modified code inside "hw3_minibatch.py" file if the argument $minibatch\_size > 0$.

We used 'MiniBatcher' class implemented in **my_neural_networks/minibatcher.py** file.

### Task 1a: 4)

Table 1: **Losses**

| **Learning Rate** | $1e^{-3}$ | $1e^{-4}$ | $1e^{-5}$ |
|---|---|---|---|
| **Batch Size** | | | |
| 100 | 0.5108 | 1.7169 | 5.4342 |
| 500 | 0.5862 | 1.5882 | 6.0633 |
| 3000 | 0.5155 | 1.5194 | 5.5005 |
| 5000 | 0.4996 | 1.6411 | 5.7330 |

Table 2: **Training Accuracies**

| **Learning Rate** | $1e^{-3}$ | $1e^{-4}$ | $1e^{-5}$ |
|---|---|---|---|
| **Batch Size** | | | |
| 100 | 85% | 61.3% | 17.5% |
| 500 | 86.3% | 60% | 13.6% |
| 3000 | 85.6% | 58.6% | 15.9% |
| 5000 | 86.2% | 61.3% | 17.7% |

We can notice that regardless of batch sizes for learning rate = $1e^{-3}$ we achieved higher accuracy both in training and testing data (Table. 2 and Table. 3) and lesser losses (Table. 1). For learning

Table 3: **Testing Accuracies**

| **Learning Rate** | $1e^{-3}$ | $1e^{-4}$ | $1e^{-5}$ |
|---|---|---|---|
| **Batch Size** | | | |
| 100 | 85.6% | 62.5% | 17.1% |
| 500 | 86.4% | 60.7% | 13.1% |
| 3000 | 85.9% | 59.7% | 16.1% |
| 5000 | 86.3% | 63% | 18.3% |

rate = $1e^{-5}$ we achieved very lower accuracy both in training and testing data (Table. 2 and Table. 3) and higher losses (Table. 1). Regarding minibatch size, our observation is "bigger batch size bigger learning rate". Because bigger batch size means more confidence in the direction of "descent" of the loss surface.

### Task 1a: 6) Comparison of the results of using regular GD and minibatch SGD

Using SGD with minibatch size= 300, we achieved 60.4% training accuracy and 62% test accuracy with loss = 1.59 (Fig. 5 and Fig. 6) during the last epoch among 100 epochs and elapse time was 0.99 seconds. Using GD, we achieved 64.8% training accuracy and 66.2% test accuracy with loss = 1.46 (Fig. 3 and Fig. 4) during the last epoch among 100 epochs and elapse time was 2.99 seconds.

While in GD, we have to run through all the samples of training set to do a single update for a parameter in a particular iteration, in SGD, on the other hand, we use subset of training sample (Minibatch) from the training set to do the update for a parameter in a particular iteration. If the number of training samples are large, then using gradient descent may take too long because in every iteration when we are updating the values of the parameters, we are running through the complete training set. On the other hand, using SGD will be faster because we use minibatch and it starts improving itself right away from the first sample.

SGD often converges much faster compared to GD but the loss function is not as well minimized as in the case of GD. Often in most cases, the close approximation that we get in SGD for the parameter values are enough because they reach the optimal values and keep oscillating there.

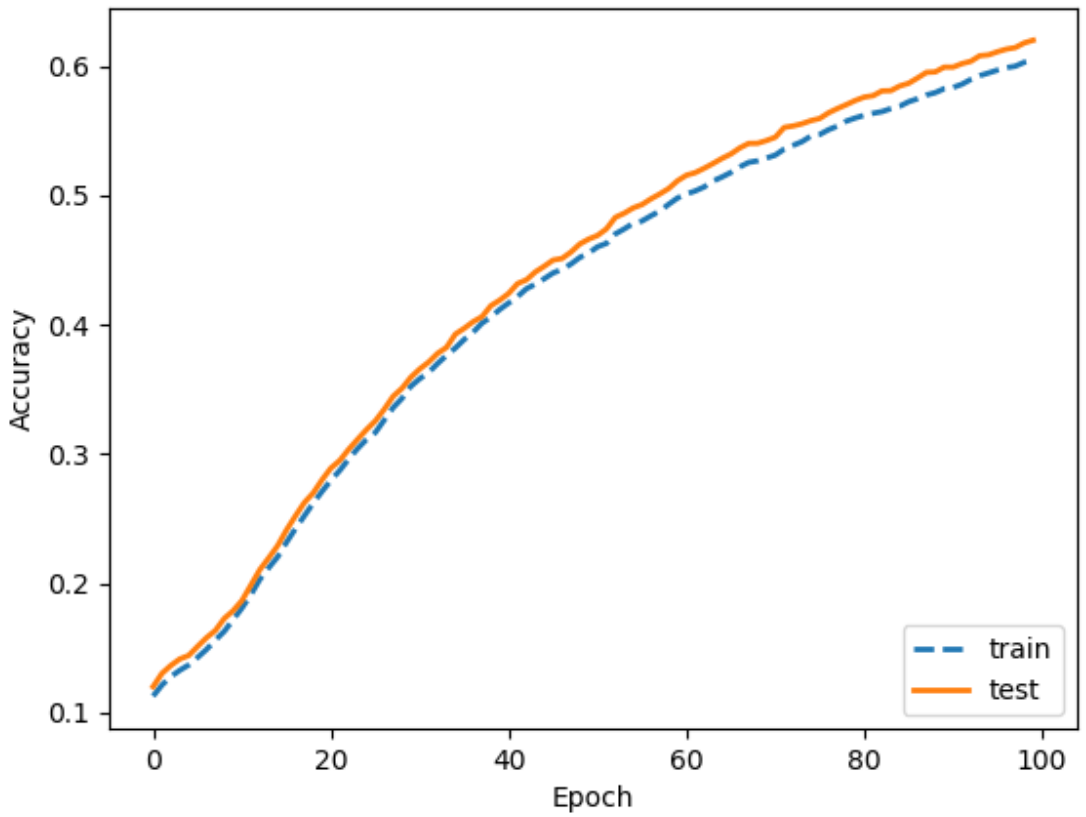


Figure 1: Accuracy SGD with minibatch size = 300.

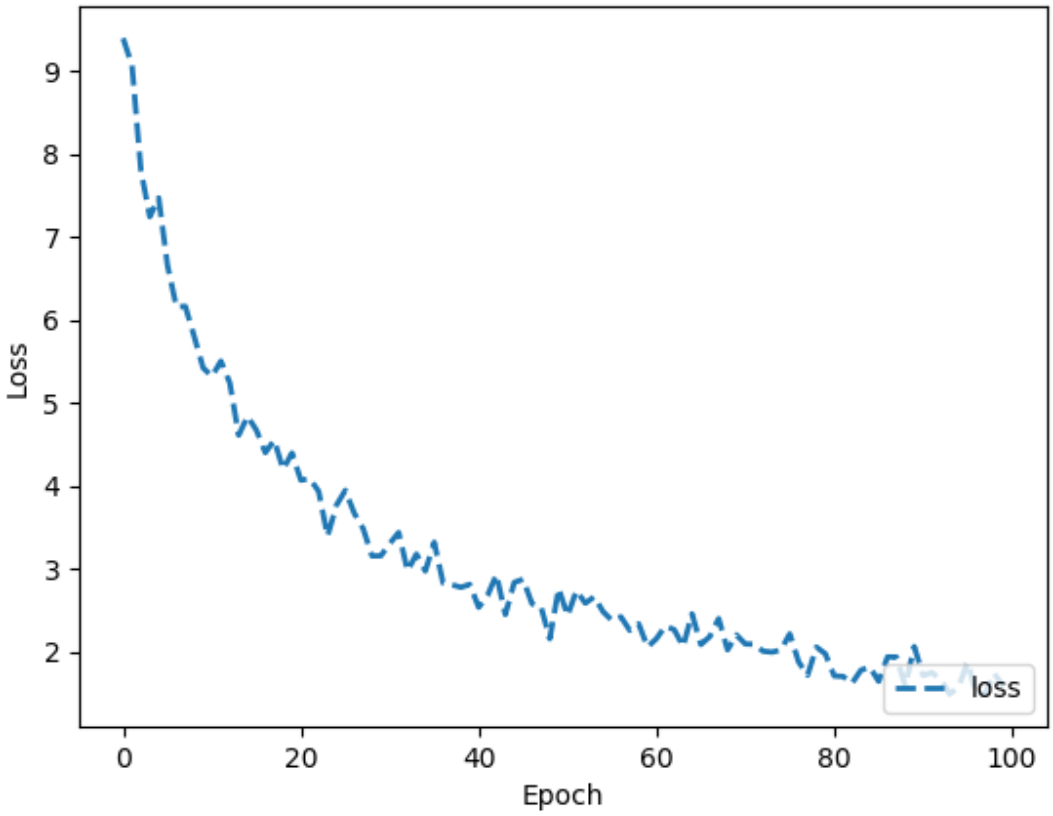


Figure 2: Loss SGD with minibatch size = 300.

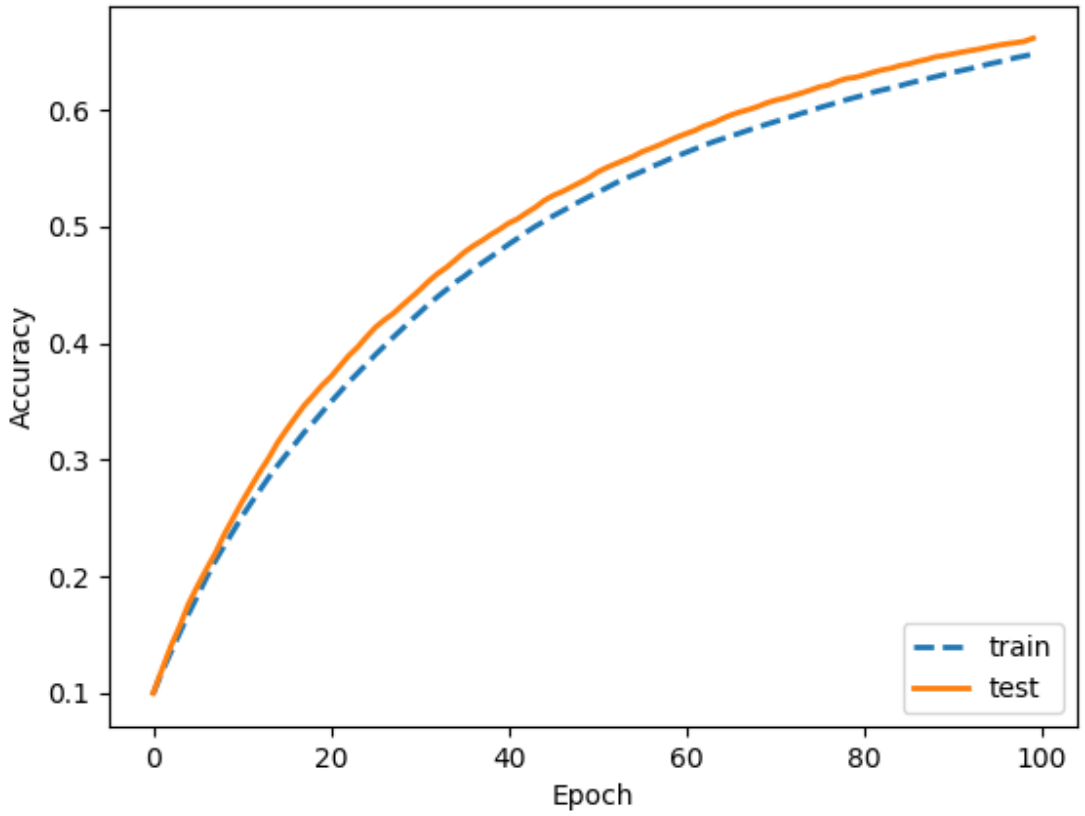


Figure 3: Accuracy GD

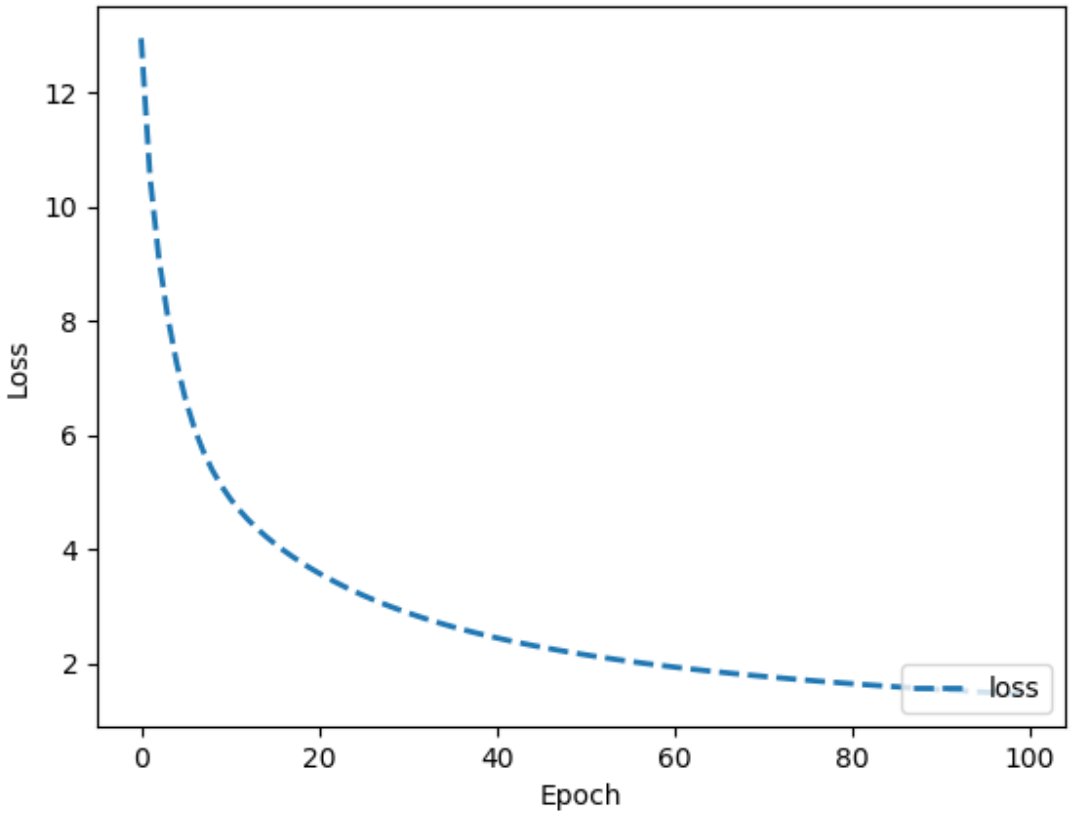


Figure 4: Loss GD

## Task 1b: Adaptive Learning Rate Algorithms

**Task 1b: 3) Plot "Loss vs. Epochs" and "Accuracies vs. Epochs" for each optimizer**

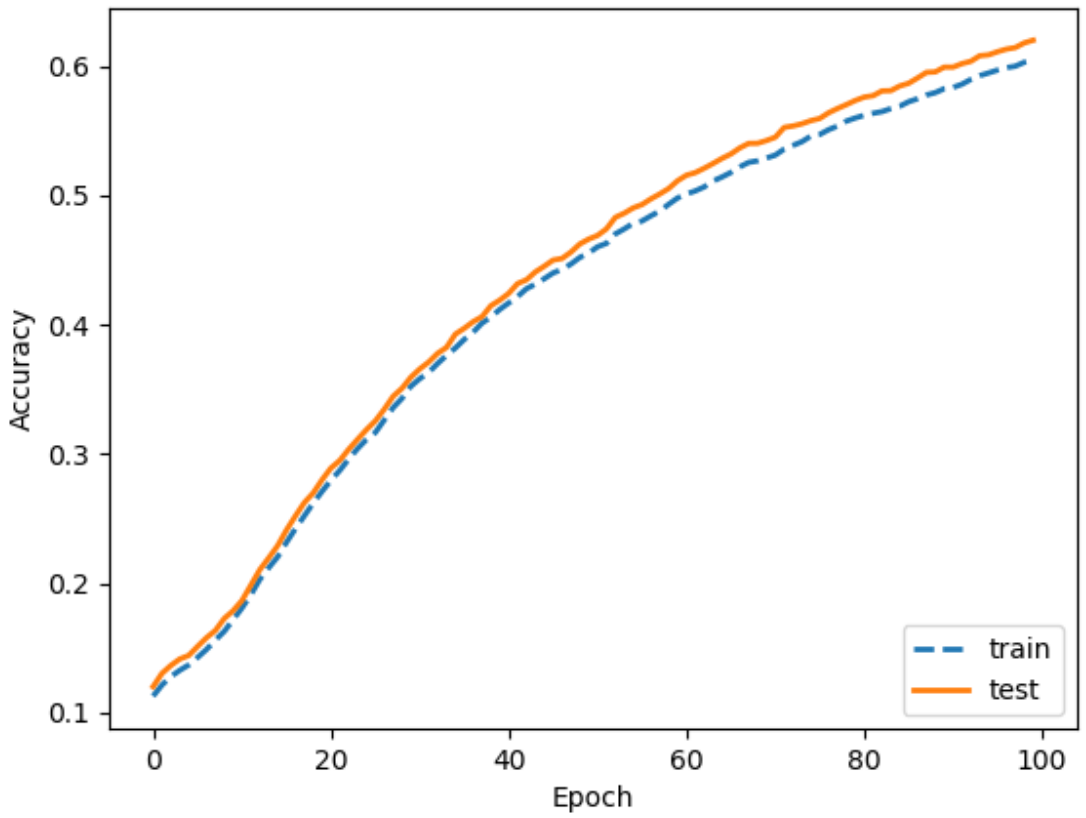


Figure 5: Accuracy SGD with minibatch size = 300.

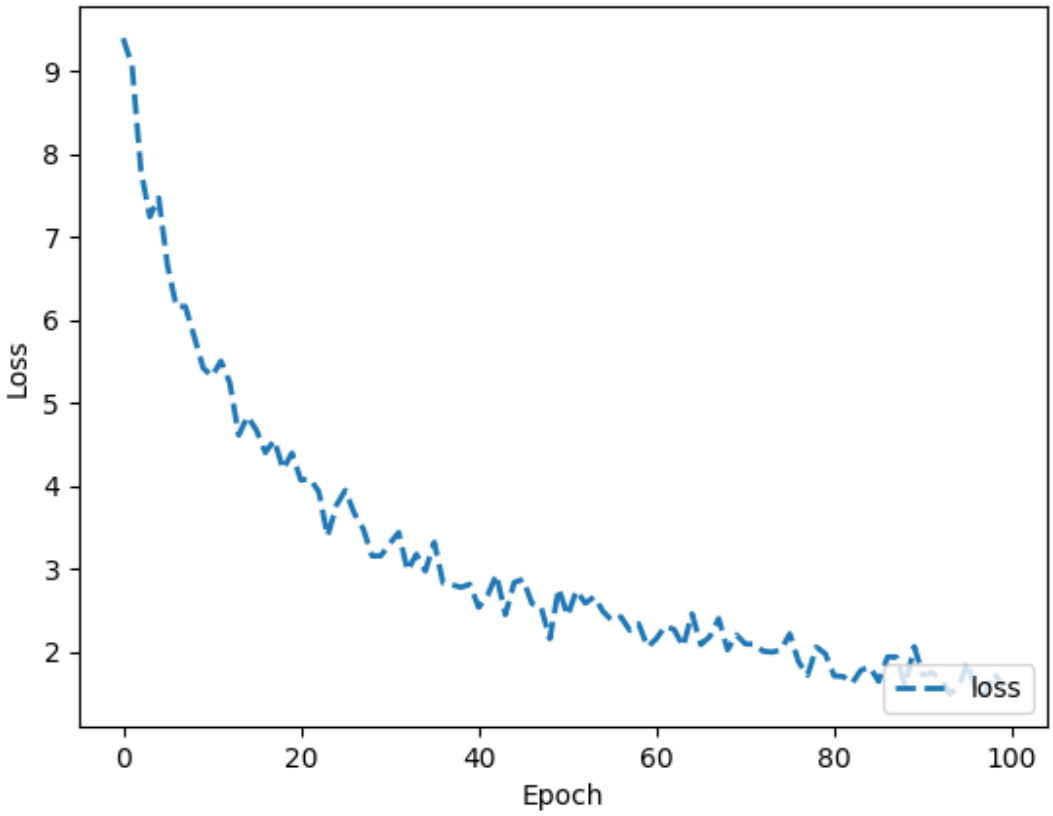


Figure 6: Loss SGD with minibatch size = 300.

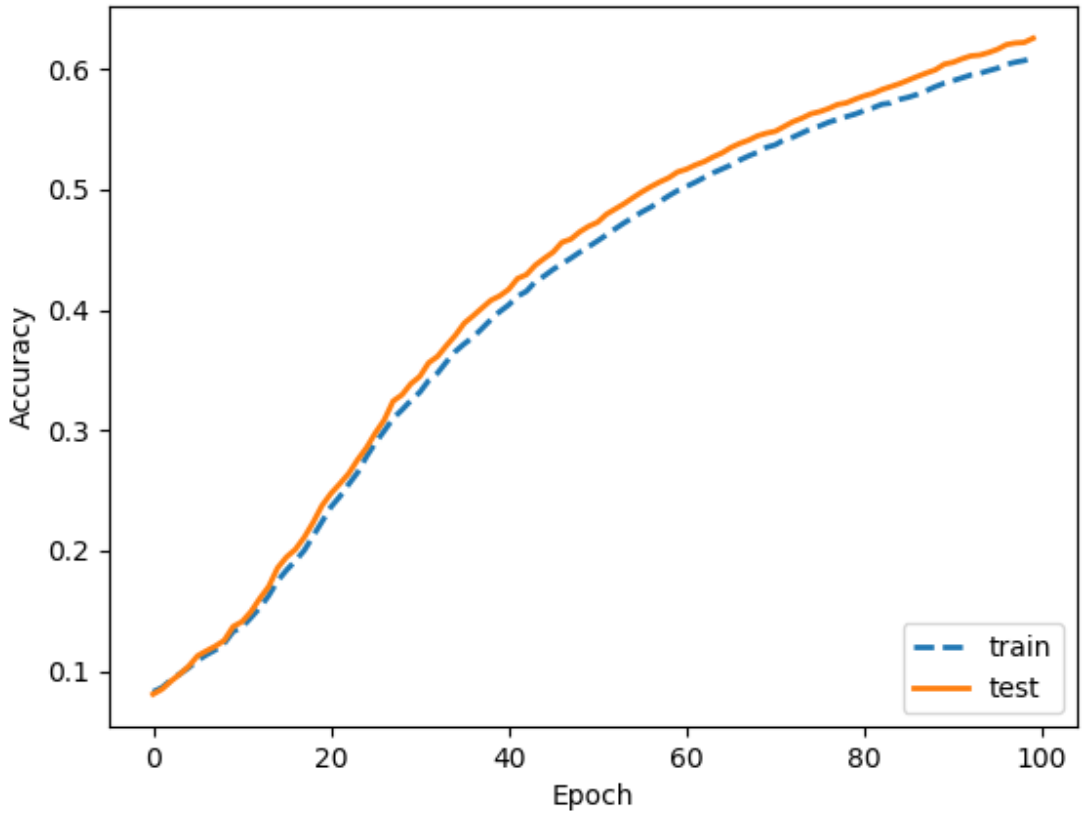


Figure 7: Accuracy Momentum with minibatch size = 300.

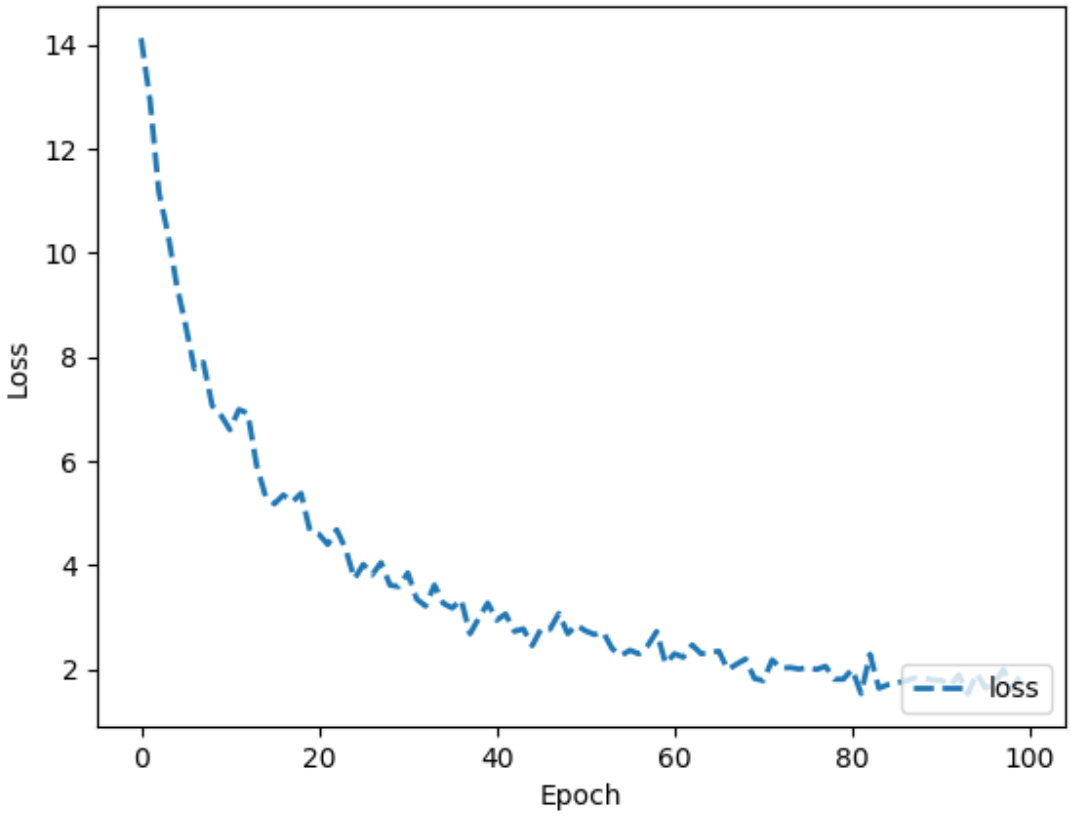


Figure 8: Loss Momentum with minibatch size = 300.

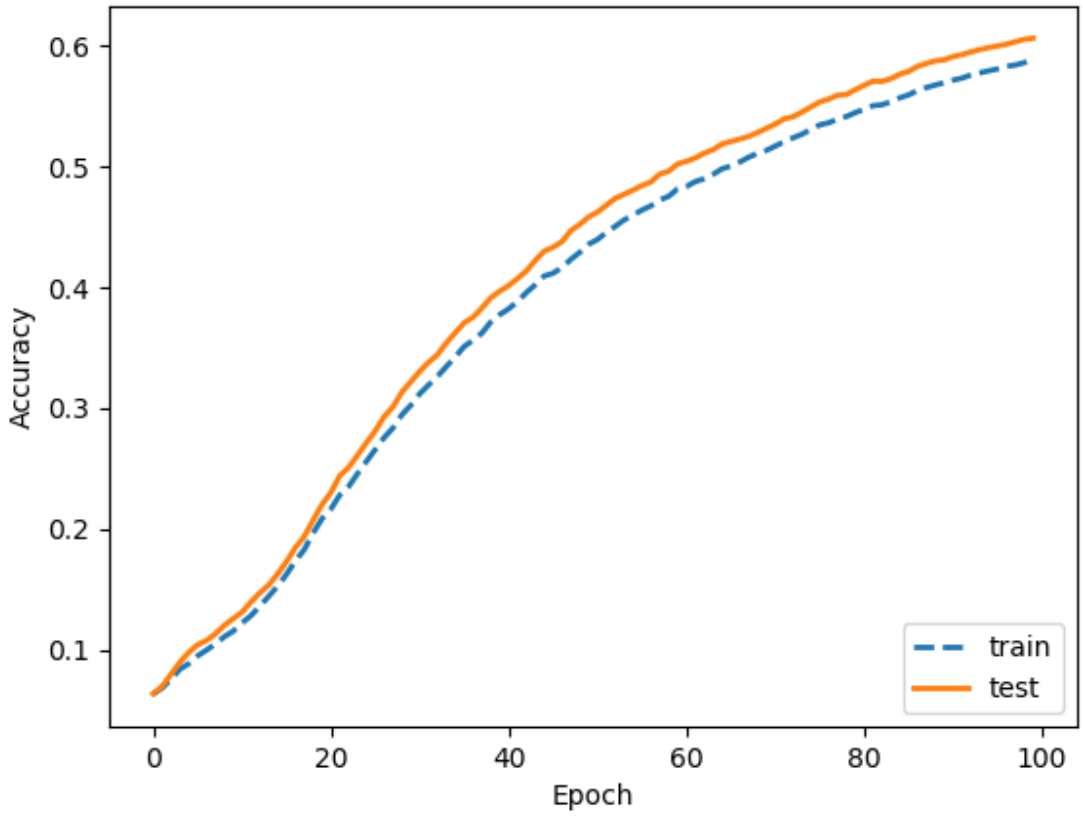


Figure 9: Accuracy Nesterov with minibatch size = 300.

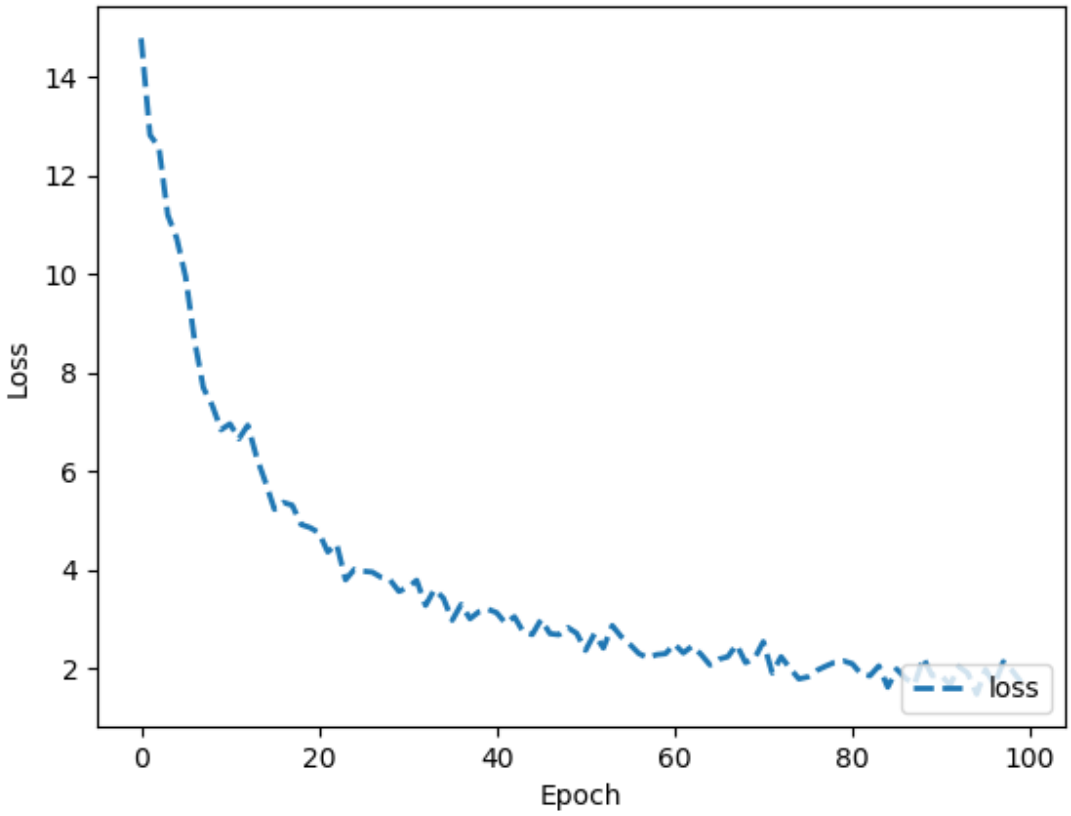


Figure 10: Loss Nesterov with minibatch size = 300.

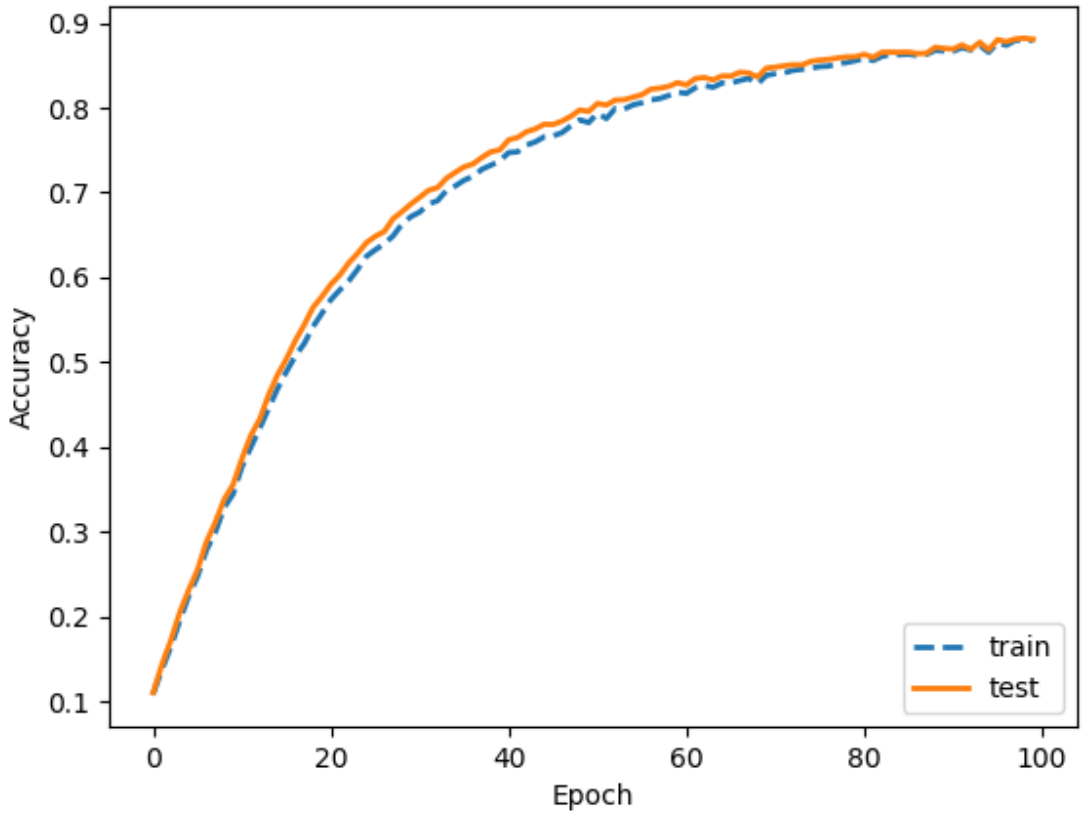


Figure 11: Accuracy Adam with minibatch size = 300.

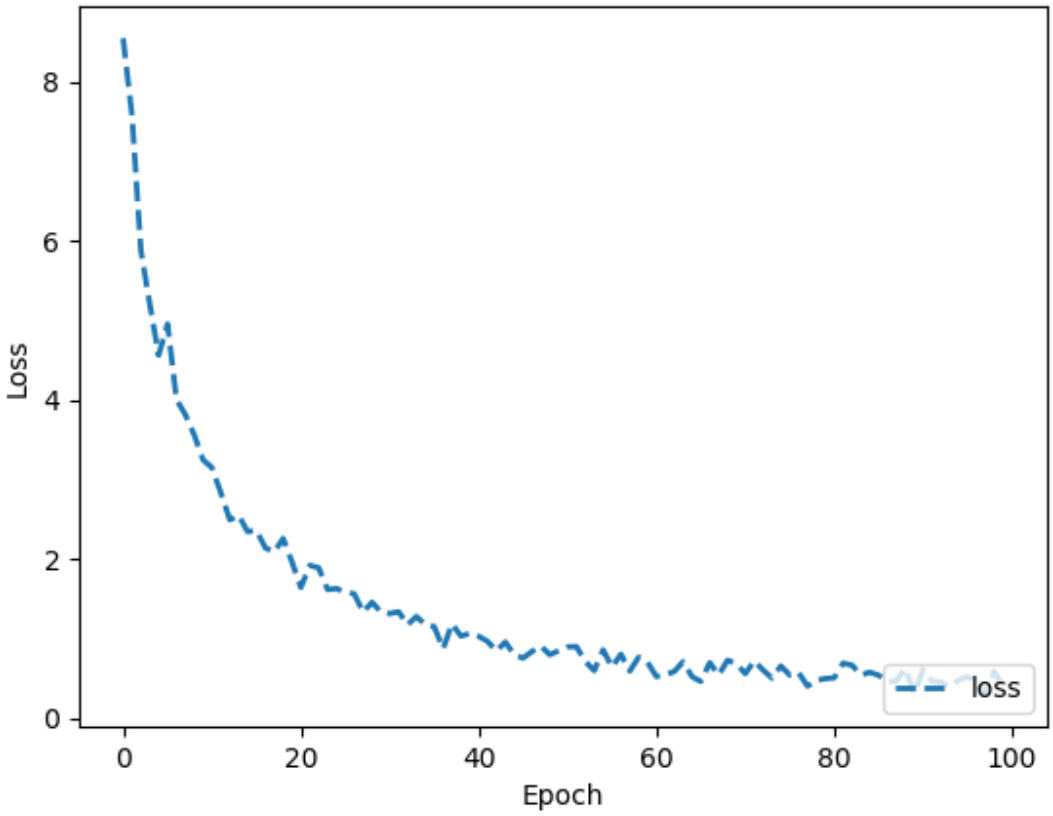


Figure 12: Loss Adam with minibatch size = 300.

### Task 1b: 4) Observation and Algorithm Comparisons

Using Adam optimizer with minibatch size= 300 and default learning rate ($1e^{-4}$), we achieved the best performance among 3 other optimizers (SGD, Momentum, Nesterov). We achieved 87.95% training accuracy and 88.08% test accuracy (Fig.11) with loss = 0.41 (Fig.12) during the last epoch among 100 epochs and elapse time was 0.84 seconds.

Momentum optimizer performs slightly better than SGD during accuracy for training (SGD: 60.4%, Momentum: 60.9%) and testing (SGD: 62%, Momentum: 62.6%) data. Nesterov optimizer provides less accuracy in training (58.8%) and testing (60.6%) than SGD and Momentum.

SGD has dificulties navigating valleys, areas where the energy surface is much more steeply in one direction than in another direction. In valleys, SGD may oscillate across the slopes of the valley while making slow progress towards the bottom of the valley.Momentum helps accelerate SGD in the most "common" direction, which is the bottom of the valley, dampening oscillations as the movement of going up and down the wall of the valley will tend to cancel out. The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

In Nesterov, we give the momentum term some kind of prescience (an approximation of the next position of the parameters). We can effectively look ahead by calculating the gradient not w.r.t. to the current parameters but w.r.t. the approximate future position of the parameters.

Adaptive Moment Estimation (Adam) computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients, Adam also keeps an exponentially decaying average of past gradients, similar to momentum.

## Task 2a: L2-Regularization

**Task 2a: 4) Plot "Loss vs. Epochs" and "Accuracies vs. Epochs" for each lambda value**

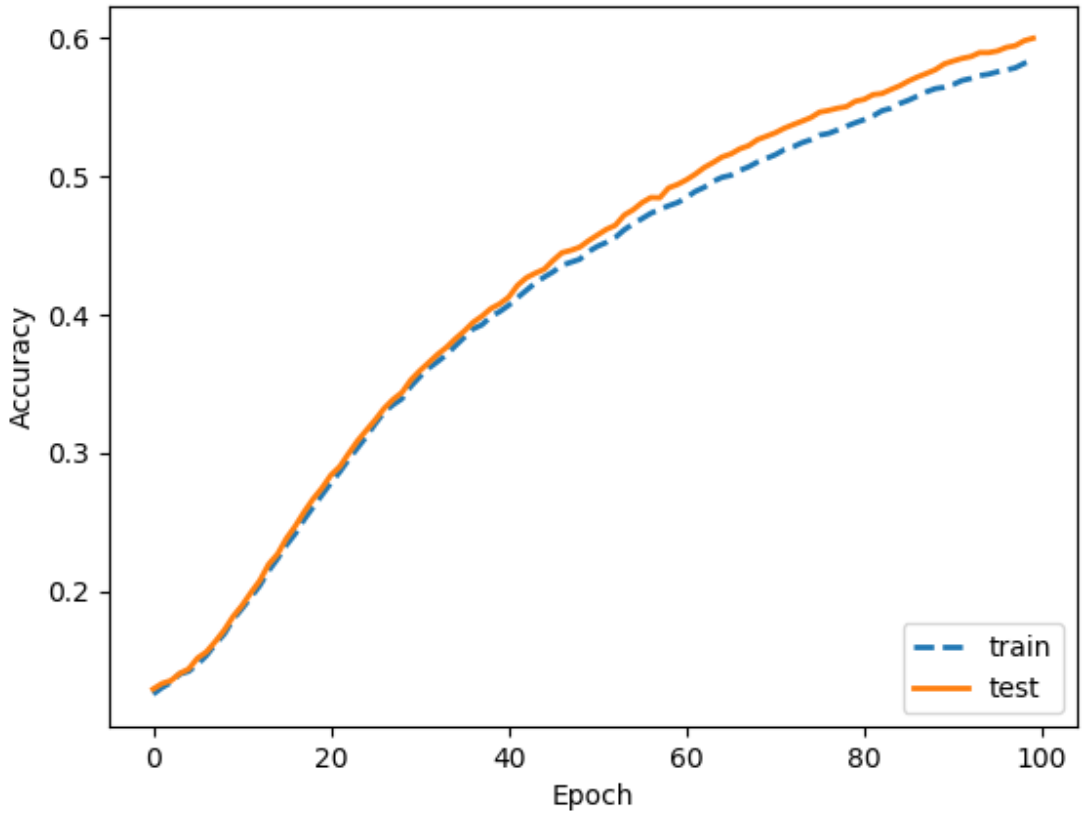


Figure 13: Accuracy SGD with minibatch size = 300, lambda = 1.

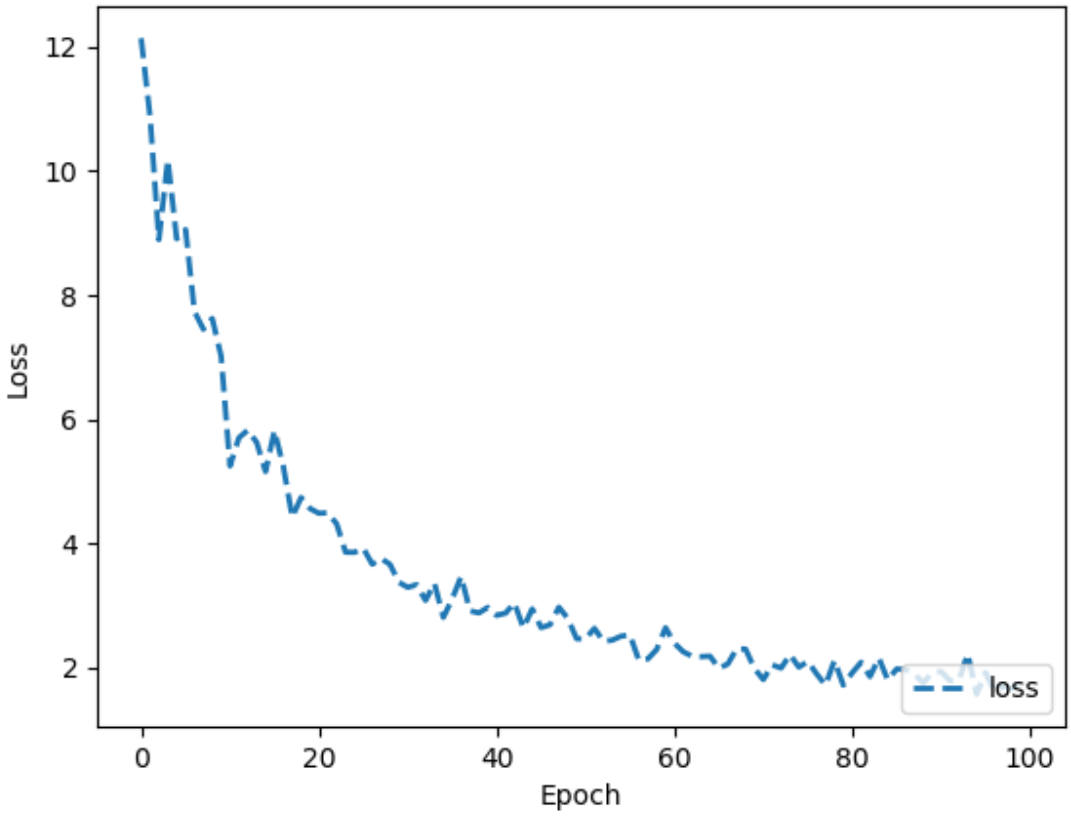


Figure 14: Loss SGD with minibatch size = 300, lambda = 1.

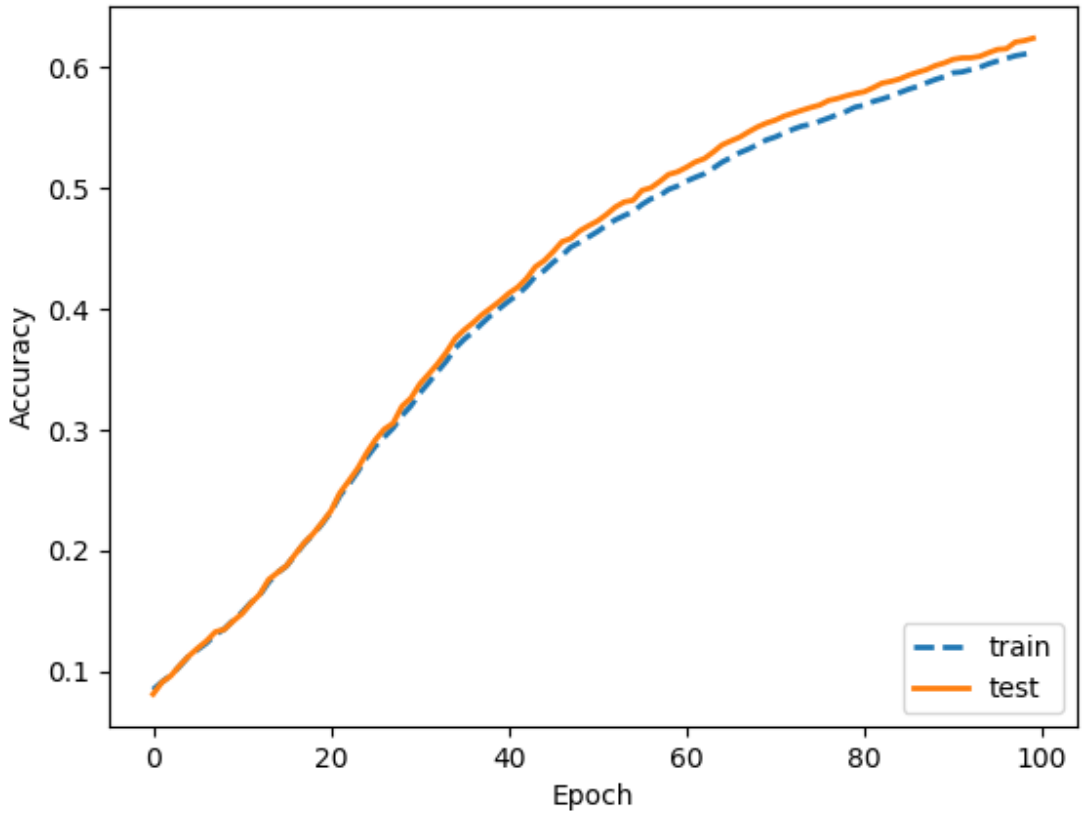


Figure 15: Accuracy SGD with minibatch size = 300, lambda = 0.1.

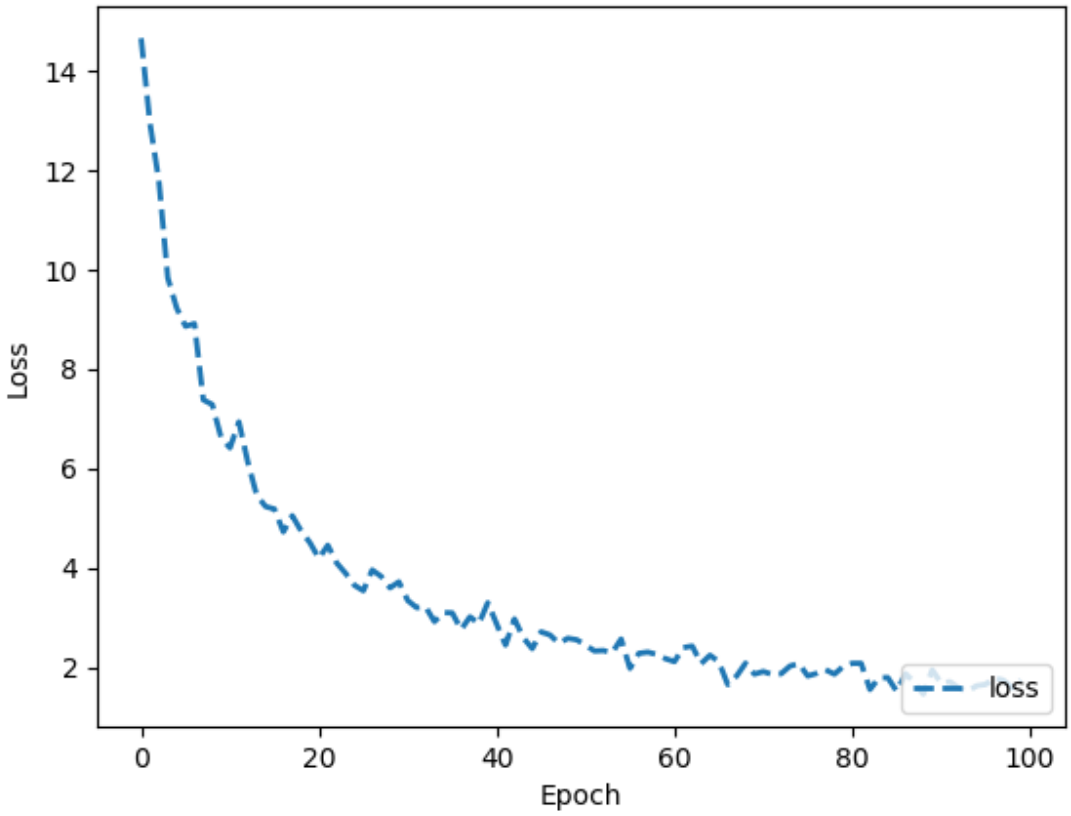


Figure 16: Loss SGD with minibatch size = 300, lambda = 0.1.

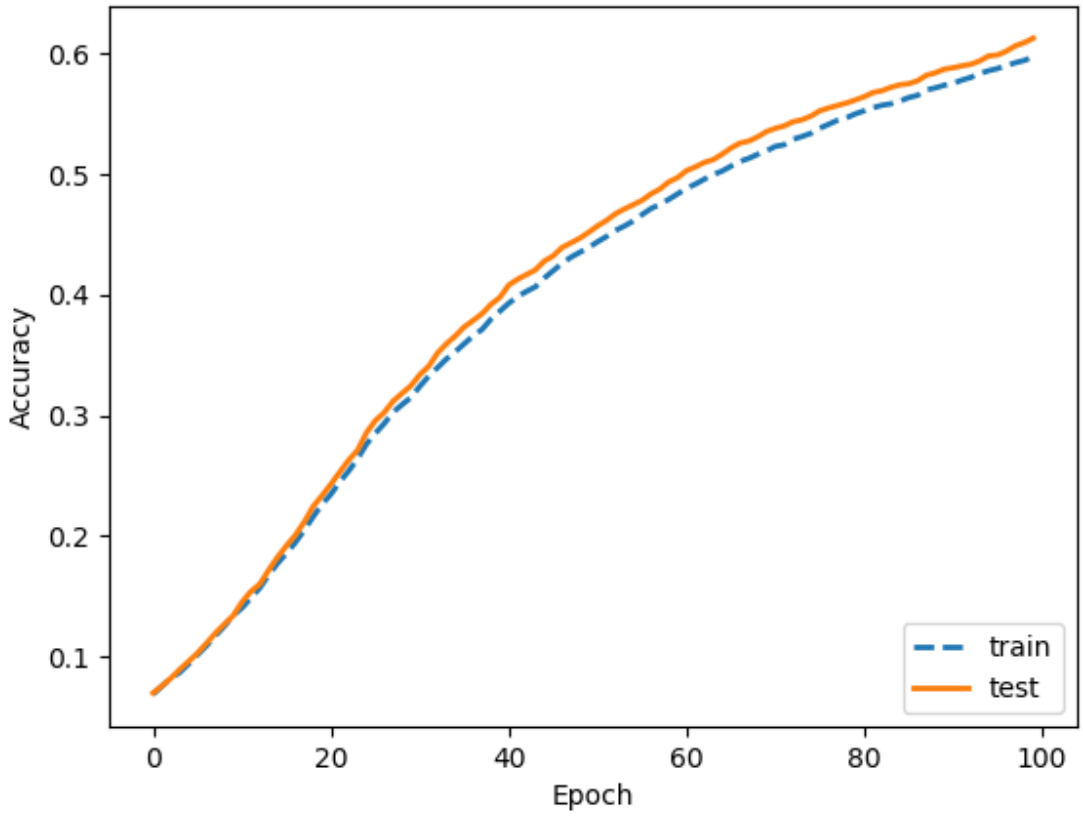


Figure 17: Accuracy SGD with minibatch size = 300, lambda = 0.01.

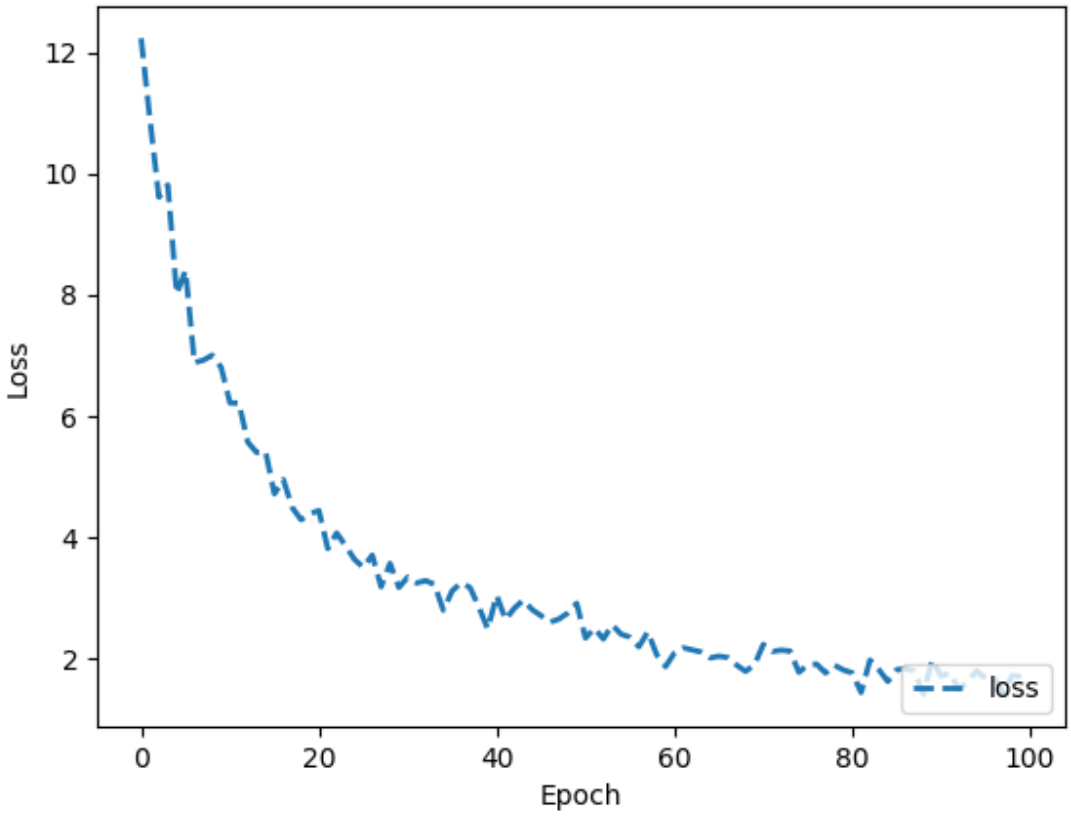


Figure 18: Loss SGD with minibatch size = 300, lambda = 0.01.

### Task 2a: 5) Observations and Comparisons

Regularization provides an approach to reduce the overfitting of model on the training data and improve the performance of the model on new data. In our case, we achieved better performance with L2 regularization for lambda = 0.1.

When lambda = 1, then train accuracy = 58.3%, test accuracy = 60%, loss = 1.8

When lambda = 0.1, then train accuracy = 61.2%, test accuracy = 62.4% , loss = 1.7

When lambda = 0.01, then train accuracy = 60% , test accuracy = 61.3% , loss = 1.7

## Task 3: Implement a CNN

### Task 3: 1) Neural Network Architecture

We use two 2D convolutional layer with max pooling layer and reLU activations, 1 fully connected layer with ReLU activation, another fully connected layer, output layer with log softmax activation and learning_rate = 0.05. The number of output features in each dimension was calculated using the following formula:

$$n_{out} = \lfloor \frac{n_{in} + 2p - k}{s} \rfloor + 1$$

$n_{in}$ : number of input features

$n_{out}$ : number of output features

$k$ : convolution kernel size

$p$ : convolution padding size

$s$ : convolution stride size

First convolution layer has in_channels = 1, out_channels = 10 as we have image size = 28 * 28, kernel size = 5 * 5 and stride = 1 (given code in lecture 5). Using the above formula number of output features of first 2D convolutional layer is 24. Then we used 2D max pooling layer with 2 * 2 kernel size and stride = 2. The output of first 2D convolutional layer is the input of 2D max pooling layer so we have $n_{in}$ = 24 * 24. So output features of first 2D max pooling layer is 12. Output of $2^{nd}$ convolutional layer is 8 and $2^{nd}$ max pooling layer is 4. We did not use zero padding in this case so, $p = 0$. So the size of X is torch.Size([32, 20, 4, 4]) after applying 2 convolution and 2 max pooling layers. So the input dimension of fully connected layer is $20 * 4 * 4 = 320$ and output dimension is 100 which is the input dimension of $2^{nd}$ fully connected layer. The output dimension of fully connected output layer is 10 as we have 10 classes and we passed it to log softmax activation function.

### Task 3: 2) CNN with k * k image patches

We added argument $-c$ in command line to run - -CNN. $-c$ 1 for running CNN and default value is 0. We ran for 10 epochs.

For kernel = 3* 3, stride = 3, we added padding = 1 to keep the input and output of the

convolutional layers of the same dimensions. We changed self.fc1 = nn.Linear(20, 100) because after changing kernel size, stride, and padding we have X of size torch.Size([32, 20, 1, 1]).

Command to run CNN:

**python hw3_minibatch.py -v data -c 1**

Our observation after running the command:

Validation set: Average loss: 0.3113, Accuracy: 8969/10000 (90%)

Validation set: Average loss: 0.2242, Accuracy: 9289/10000 (93%)

Validation set: Average loss: 0.1973, Accuracy: 9330/10000 (93%)

Validation set: Average loss: 0.2168, Accuracy: 9280/10000 (93%)

Validation set: Average loss: 0.1617, Accuracy: 9451/10000 (95%)

Validation set: Average loss: 0.1800, Accuracy: 9429/10000 (94%)

Validation set: Average loss: 0.1983, Accuracy: 9381/10000 (94%)

Validation set: Average loss: 0.1547, Accuracy: 9497/10000 (95%)

Validation set: Average loss: 0.1505, Accuracy: 9513/10000 (95%)

Validation set: Average loss: 0.1520, Accuracy: 9522/10000 (95%)

For kernel = 14 * 14, stride = 1, we changed the padding = 3 (As we have one channel, we were having issue because kernel size can't be greater than actual input size).

Validation set: Average loss: 0.0637, Accuracy: 9798/10000 (98%)

Validation set: Average loss: 0.0540, Accuracy: 9821/10000 (98%)

Validation set: Average loss: 0.0421, Accuracy: 9875/10000 (99%)

Validation set: Average loss: 0.0537, Accuracy: 9821/10000 (98%)

Validation set: Average loss: 0.0451, Accuracy: 9861/10000 (99%)

Validation set: Average loss: 0.0459, Accuracy: 9865/10000 (99%)

Validation set: Average loss: 0.0444, Accuracy: 9864/10000 (99%)

Validation set: Average loss: 0.0389, Accuracy: 9878/10000 (99%)

Validation set: Average loss: 0.0447, Accuracy: 9875/10000 (99%)

Validation set: Average loss: 0.0415, Accuracy: 9885/10000 (99%)

The reduction in the size of the input to the feature map is referred to as border effects. It is caused by the interaction of the filter with the border of the image. This is often not a problem for large images and small filters but can be a problem with small images. We used zero padding to get rid of border effects.

Smaller strides lead to large overlaps which means the output volume is high. Larger strides lead to lesser overlaps which means lower output volume.

**Task 3: 3.a) Plot with two curves: the training accuracy and validation accuracy, with the x-axis as the number of epoch for original task**

We added argument $-s$ in command line to run - -shuffleCNN. $-s$ 1 for running shuffle CNN and default value is 0. We ran for 100 epochs.

Command to run shuffleCNN:

**python hw3_minibatch.py -v data -s 1**

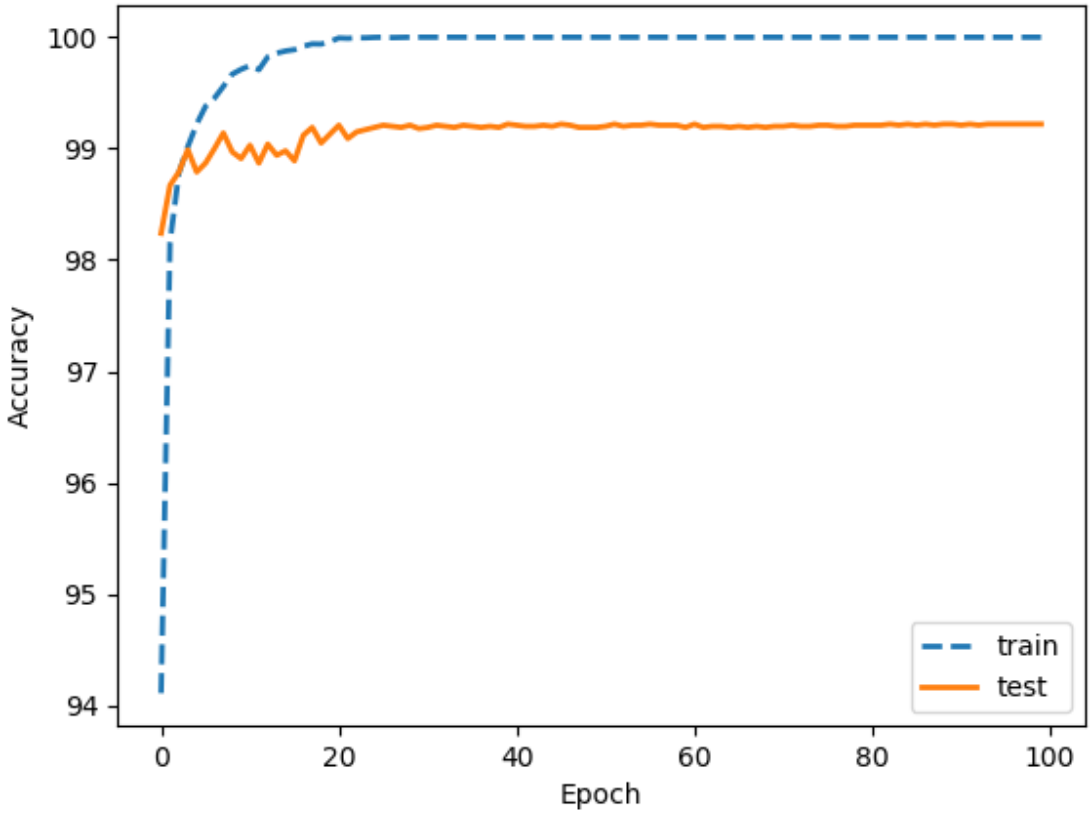


Figure 19: Accuracy vs epoch (original task)

**Task 3: 3.b) Plot with two curves: the training accuracy and validation accuracy, with the x-axis as the number of epoch for the task of randomly shuffled target labels in training data**

We added argument $-s$ in command line to run - -shuffleCNN. $-s$ 1 for running shuffle CNN and default value is 0. We ran for 100 epochs.

Command to run shuffleCNN:

**python hw3_minibatch.py -v data -s 1**

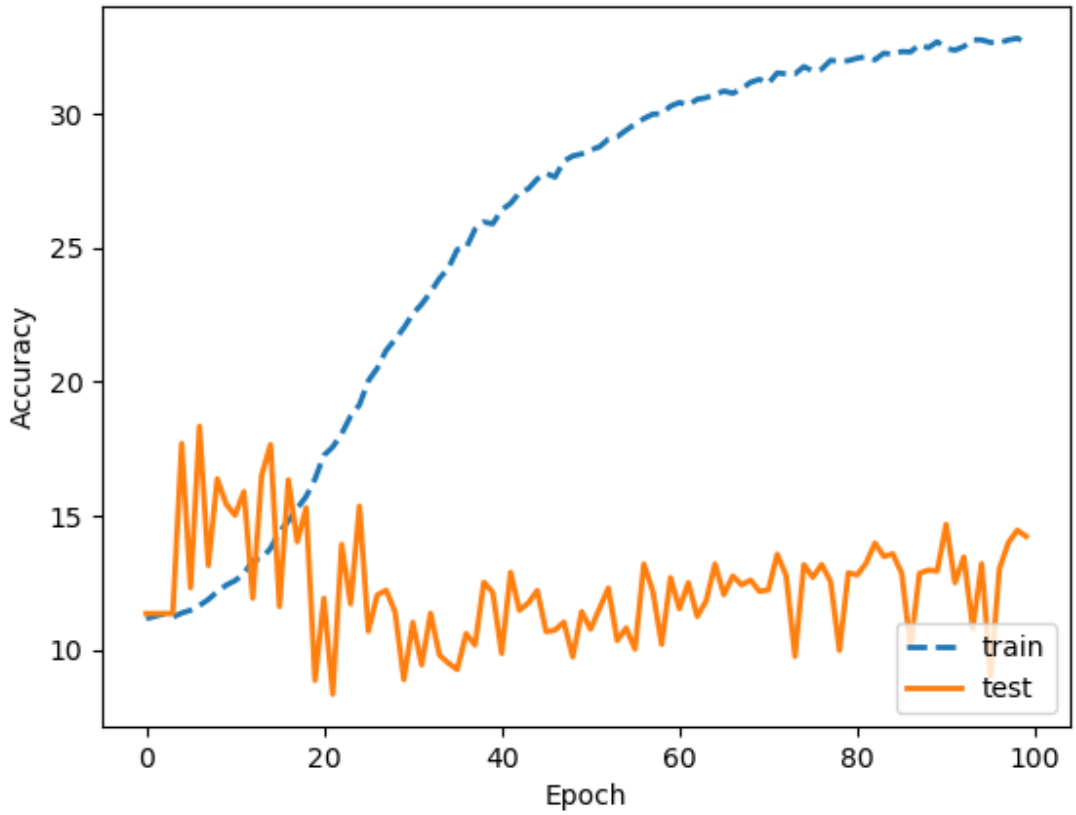


Figure 20: Accuracy vs epoch (shuffled target task)

As we randomly shuffled target labels of training data so that the image (hand-written digit) and its label are unrelated, neural network started learn wrongly annotated training data and overfit the data. For the earlier epochs until 20, validation accuracy was little higher than the train accuracy. After 20 epochs, validation accuracy remained almost similar range. The train accuracy increased gradually as number of epochs is increasing. As we know network could learn better over the time and if we continue train the network for larger epoch (let's say 1000 or so on without any early stop strategy), the train accuracy might increase but validation accuracy might remains either steady or decrease.

No, the inductive bias of a CNN can not naturally "understand images" in this "Shuffle CNN Task".

But I think in general, CNN can take advantage of pixel locality to more easily learn local pixel structure which is an inductive bias. Local pixel structural information will be extracted as a filter based on the concept of local convolutions. In case of Image Invariance, CNNs should be trained to be translation and rotation-invariant. Cropping, Rotation, Translation, Illumination are data augmentation hacks that can help a neural network become more rotation / translation invariant. At each mini-batch gradient descent, randomly rotate, translate, and change the illumination of the mini-batch images. Then CNN can classify all hand-written '0' s as the same digit.

**Task 3: 3.c) Flatness Plot**

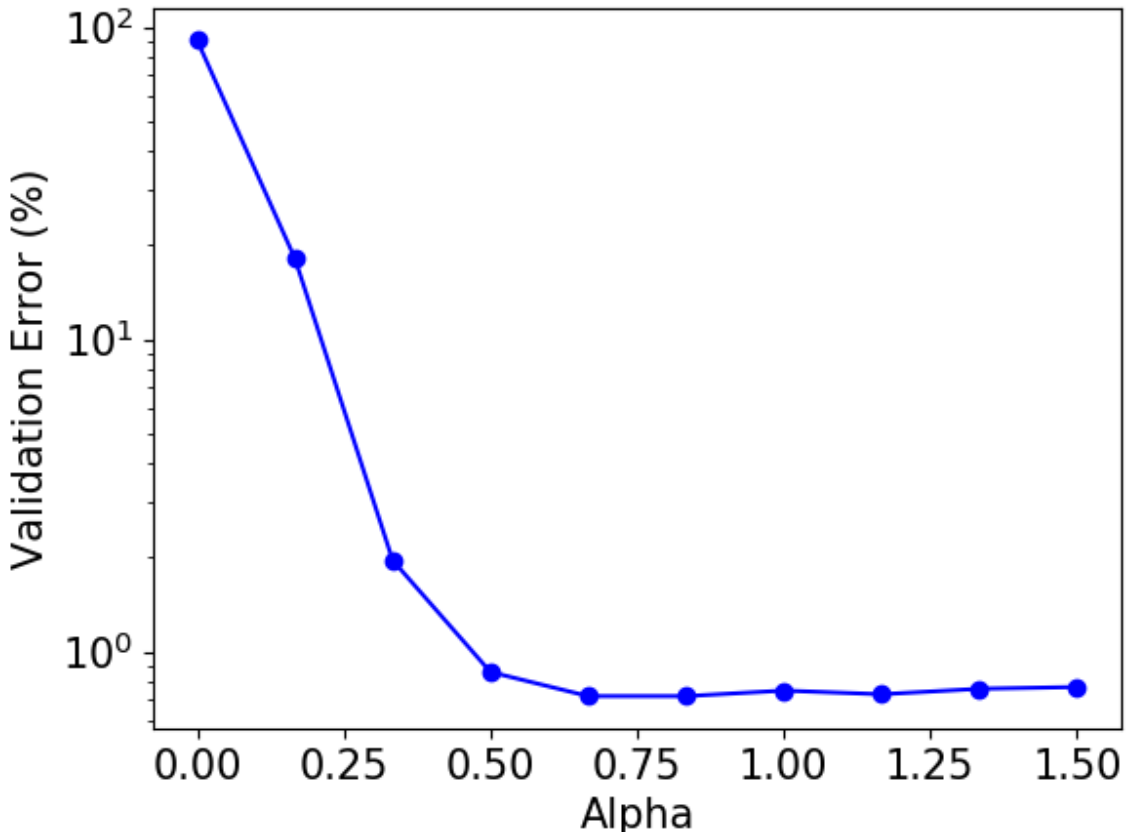


Figure 21: Validation Error (%) vs Alpha (Flatness Plot Original Task)

Let, $\theta_0$ be the initial (random) parameters of this network and let $\theta_{final}$ be the final parameters after training with SGD until we obtain very good accuracy. Let, $L(\theta)$ be the loss over the validation data for parameter $\theta$. Let, $\alpha$ is the interpolation parameter between $\theta_0$ and $\theta_{final}$. In Fig. 21 and Fig. 22 we plot the loss function $L(\theta(\alpha))$ as a function of $\alpha$. In Fig 21, we can notice that at the beginning loss is higher because of initial random solution. With increasing value of $\alpha$ loss started to decrease and reach the flat region at the bottom of the valley. Validation loss would be better if we stop earlier. Flatter region is better model as flatter region means very large puddle.

For random shuffle task (Fig. 22), we did not find flat region. We found sharp minima which have class boundaries that pass close to training data, putting those nearby points at risk of misclassification when the boundaries are perturbed. As a result, small movement away from the optimal parameters causes a large increase in the loss function.

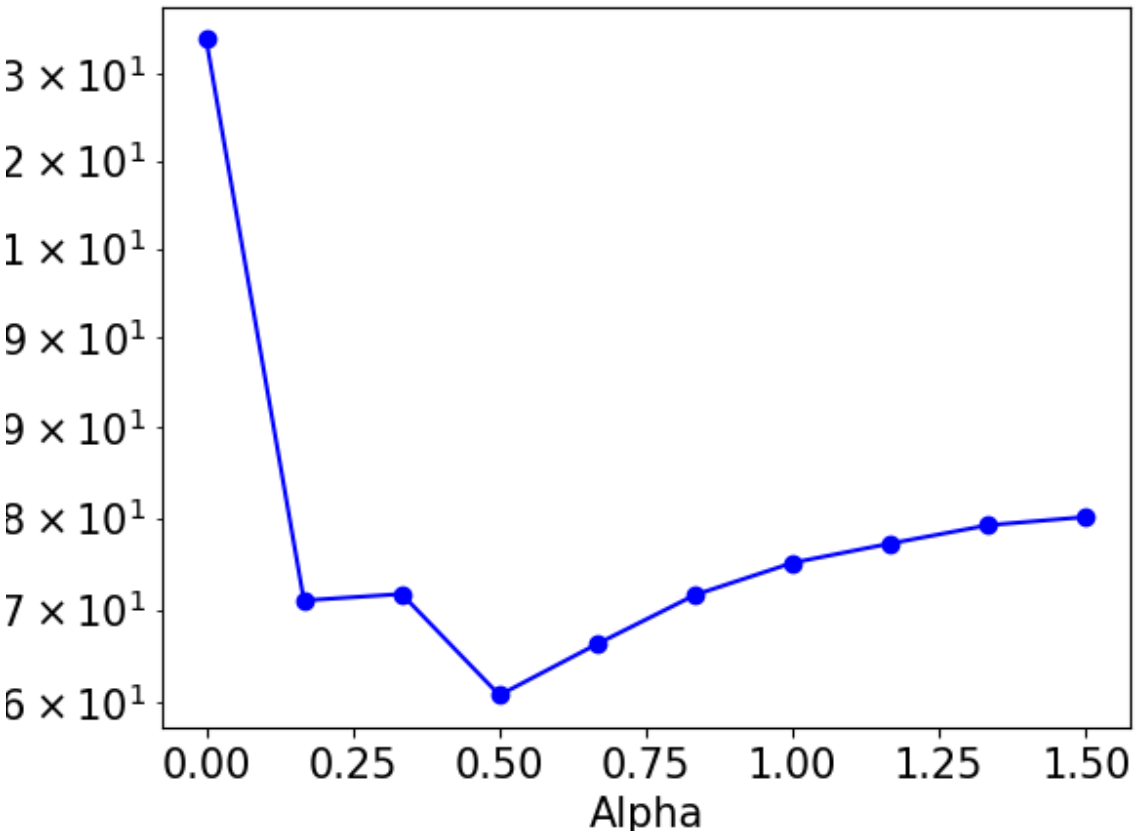


Figure 22: Validation Error (%) vs Alpha (Flatness Plot Shuffled Task)