CSCI567 HW3

Ritu Pravakar and Charlene Yuen

October 2022

Theory-based Questions

Problem 1: Multi-class Perceptron (16pts)

1.1 (8pts) To optimize this loss function, we need to first derive its gradient. Specifically, for each $i \in [n]$ and $c \in [C]$, write down the partial derivative $\frac{\partial F_i}{\partial \boldsymbol{w}_c}$ (provide your reasoning). For simplicity, you can assume that for any i, $\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}_i, \ldots, \boldsymbol{w}_C^{\mathrm{T}}\boldsymbol{x}_i$ are always C distinct values (so that there is no tie when taking the max over them, and consequently no non-differentiable points needed to be considered).

Since $F_i(\boldsymbol{w}_1, \ldots, \boldsymbol{w}_C) = \max\left\{0, \max_{y \neq y_i}\left(\boldsymbol{w}_y^{\mathrm{T}}\boldsymbol{x}_i - \boldsymbol{w}_{y_i}^{\mathrm{T}}\boldsymbol{x}_i\right)\right\}$:

If $y = y_i$:
$F_i(w_1, ..., w_C) = 0$
$\frac{\partial F_i}{\partial w_c} = 0$

Else:
For $\hat{y} = \max_{y \neq y_i}\left(\boldsymbol{w}_y^{\mathrm{T}}\boldsymbol{x}_i - \boldsymbol{w}_{y_i}^{\mathrm{T}}\boldsymbol{x}_i\right)$: $F_i(w_1, ..., w_C) = \boldsymbol{w}_{\hat{y}}^{\mathrm{T}}\boldsymbol{x}_i - \boldsymbol{w}_{y_i}^{\mathrm{T}}\boldsymbol{x}_i$
If $c = \hat{y}$:
$\frac{\partial F_i}{\partial w_c} = x_i$
If $c = y_i$:
$\frac{\partial F_i}{\partial w_c} = -x_i$
Else:
$\frac{\partial F_i}{\partial w_c} = 0$

$$\frac{\partial F_i}{\partial w_c} = \begin{cases} x_i, & \text{if } c = \hat{y} \text{ and } y \neq y_i \\ -x_i, & \text{if } c = y_i \text{ and } y \neq y_i \\ 0, & \text{otherwise} \end{cases}$$

1.2 (4pts)

Algorithm 1: Multiclass Perceptron

Input: A training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$
Initialization: $\boldsymbol{w}_1 = \cdots = \boldsymbol{w}_C = \mathbf{0}$
Repeat:
  Randomly pick $(\mathbf{x_i}, y_i)$ from $i \in [n]$
  Predict y = $\mathrm{argmax}_{k \in [C]}\, w_k^T x_i$
  if $y \neq y_i$:
    $w_y \leftarrow w_y - x_i$
    $w_{y_i} \leftarrow w_{y_i} + x_i$
  endif

1.3 (4pts)

Algorithm 2: Multiclass Perceptron with kernel function $k(\cdot, \cdot)$

Input: A training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$
Initialize: $\alpha_{c,n} = 0$ for all $c \in [C]$ and $i \in [n]$
Repeat:
    Randomly pick $(\mathbf{x_i}, y_i)$ from $i \in [n]$
    Predict y = $\text{argmax}_{k \in [C]} \sum_{j=1}^{n} \alpha_{k,j} k(x_i, x_j)$
    if $y \neq y_i$:
      For all $j \in 1, ..., n$:
        $\alpha_{y,j} \leftarrow \alpha_{y,j} - 1$
        $\alpha_{y_i,j} \leftarrow \alpha_{y_i,j} + 1$
      endfor
    endif

Problem 2: Backpropagation for CNN (18pts)

2.1 (4pts) Write down $\frac{\partial \ell}{\partial v_1}$ and $\frac{\partial \ell}{\partial v_2}$ (show the intermediate steps that use chain rule). You can use the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ to simplify your notation.

$$\frac{\partial l}{\partial \hat{y}} = \frac{-ye^{-y\hat{y}}}{1+e^{-y\hat{y}}}, \frac{\partial \hat{y}}{\partial v_1} = o_1, \frac{\partial \hat{y}}{\partial v_2} = o_2$$
$$\frac{\partial l}{\partial v_1} = \frac{\partial l}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial v_1} = \frac{-ye^{-y\hat{y}}}{1+e^{-y\hat{y}}}\frac{\partial \hat{y}}{\partial v_1} = \frac{-ye^{-y\hat{y}}o_1}{1+e^{-y\hat{y}}} = -ye^{-y\hat{y}}o_1\sigma(-y\hat{y})$$
$$\frac{\partial l}{\partial v_2} = \frac{\partial l}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial v_2} = \frac{-ye^{-y\hat{y}}}{1+e^{-y\hat{y}}}\frac{\partial \hat{y}}{\partial v_2} = \frac{-ye^{-y\hat{y}}o_2}{1+e^{-y\hat{y}}} = -ye^{-y\hat{y}}o_2\sigma(-y\hat{y})$$

2.2 (6pts) Write down $\frac{\partial \ell}{\partial w_1}$ and $\frac{\partial \ell}{\partial w_2}$ (show the intermediate steps that use chain rule). The derivative of the ReLU function is $H(a) = \mathbb{I}[a > 0]$, which you can use directly in your answer.

$$\frac{\partial l}{\partial \hat{y}} = \frac{-ye^{-y\hat{y}}}{1+e^{-y\hat{y}}}, \frac{\partial \hat{y}}{\partial o_1} = v_1, \frac{\partial \hat{y}}{\partial o_2} = v_2, \frac{\partial o_1}{\partial a_1} = \mathbb{I}[a_1 > 0], \frac{\partial o_2}{\partial a_2} = \mathbb{I}[a_2 > 0], \frac{\partial a_1}{\partial w_1} = x_1, \frac{\partial a_2}{\partial w_1} = x_2$$
$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial o_1}\frac{\partial o_1}{\partial a_1}\frac{\partial a_1}{\partial w_1} + \frac{\partial l}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial o_2}\frac{\partial o_2}{\partial a_2}\frac{\partial a_2}{\partial w_1} = \frac{-ye^{-y\hat{y}}}{1+e^{-y\hat{y}}}\frac{\partial \hat{y}}{\partial w_1} = \frac{-ye^{-y\hat{y}}}{1+e^{-y\hat{y}}}(v_1\mathbb{I}[a_1 > 0]x_1 + v_2\mathbb{I}[a_2 > 0]x_2)$$

$$\frac{\partial a_1}{\partial w_2} = x_2, \frac{\partial a_2}{\partial w_2} = x_3$$
$$\frac{\partial l}{\partial w_2} = \frac{\partial l}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial o_1}\frac{\partial o_1}{\partial a_1}\frac{\partial a_1}{\partial w_2} + \frac{\partial l}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial o_2}\frac{\partial o_2}{\partial a_2}\frac{\partial a_2}{\partial w_2} = \frac{-ye^{-y\hat{y}}}{1+e^{-y\hat{y}}}\frac{\partial \hat{y}}{\partial w_2} = \frac{-ye^{-y\hat{y}}}{1+e^{-y\hat{y}}}(v_1\mathbb{I}[a_1 > 0]x_2 + v_2\mathbb{I}[a_2 > 0]x_3)$$

2.3 (8pts) Using the derivations above, fill in the missing details of the repeat-loop of the Backpropagation algorithm below that is used to train this mini CNN.

Algorithm 3: Backpropagation for the above mini CNN

Input: A training set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$, learning rate η

Initialize: set w_1, w_2, v_1, v_2 randomly

Repeat:

randomly pick an example $(\boldsymbol{x}_i, y_i)$

Forward propagation:

$$\begin{aligned}
\boldsymbol{x}_i &= \begin{bmatrix} x_{i1} & x_{i2} & x_{i3} \end{bmatrix} \\
a_1 &= x_{i1}w_1 + x_{i2}w_2, a_2 = x_{i2}w_1 + x_{i3}w_2 \\
o_0 &= x_i, o_1 = \max\{0, a_1\}, o_2 = \max\{0, a_2\} \\
\hat{y} &= o_1 v_1 + o_2 v_2
\end{aligned}$$

Backward propagation:

$$\begin{aligned}
\frac{\partial l_i}{\partial a_L} &= \frac{-ye^{-y\hat{y}}}{1+e^{-y\hat{y}}} \\
\frac{\partial l_i}{\partial a_2} &= \frac{-ye^{-y\hat{y}}}{1+e^{-y\hat{y}}}(v_1 \mathbb{I}[a_1 > 0] + v_2 \mathbb{I}[a_2 > 0])) \\
v_1 &\leftarrow v_1 - \eta \frac{\partial l_i}{\partial a_L} o_3^T = v_1 - \eta \frac{\partial l}{\partial v_1} = v_1 - \eta(-ye^{-y\hat{y}} o_1 \sigma(-y\hat{y})) \\
v_2 &\leftarrow v_2 - \eta \frac{\partial l_i}{\partial a_L} o_3^T = v_2 - \eta \frac{\partial l}{\partial v_2} = v_2 - \eta(-ye^{-y\hat{y}} o_2 \sigma(-y\hat{y})) \\
w_1 &\leftarrow w_1 - \eta \frac{\partial l_i}{\partial a_2} o_1^T = w_1 - \eta \frac{\partial l}{\partial w_1} = w_1 - \eta(\frac{-ye^{-y\hat{y}}}{1+e^{-y\hat{y}}}(v_1 \mathbb{I}[a_1 > 0] x_1 + v_2 \mathbb{I}[a_2 > 0] x_2)) \\
w_2 &\leftarrow w_2 - \eta \frac{\partial l_i}{\partial a_2} o_1^T = w_2 - \eta \frac{\partial l}{\partial w_2} = w_2 - \eta(\frac{-ye^{-y\hat{y}}}{1+e^{-y\hat{y}}}(v_1 \mathbb{I}[a_1 > 0] x_2 + v_2 \mathbb{I}[a_2 > 0] x_3))
\end{aligned}$$

Problem 3: Fashion MNIST (50pts + 15pts bonus)

Part I - Multi-Layer Perceptron (MLP) (38pts)

3.1 Neural nets (24pts)

3.1.1 Linear layer (6pts)
In attached code

3.1.2 ReLU (4pts)
In attached code

3.1.3 Mini-batch stochastic gradient descent (2pts)
In attached code

3.1.4 Backpropagation (4pts)
In attached code

3.1.5 Gradient checker (4pts)
In attached code

3.1.6 Screenshots and plots (4pts) [40sec] After finishing all five modules above, run the below command to get a result file `MLP_lr0.01_b5.json`. Take a screenshot of your output log, like in `screenshot_run_b5.png`

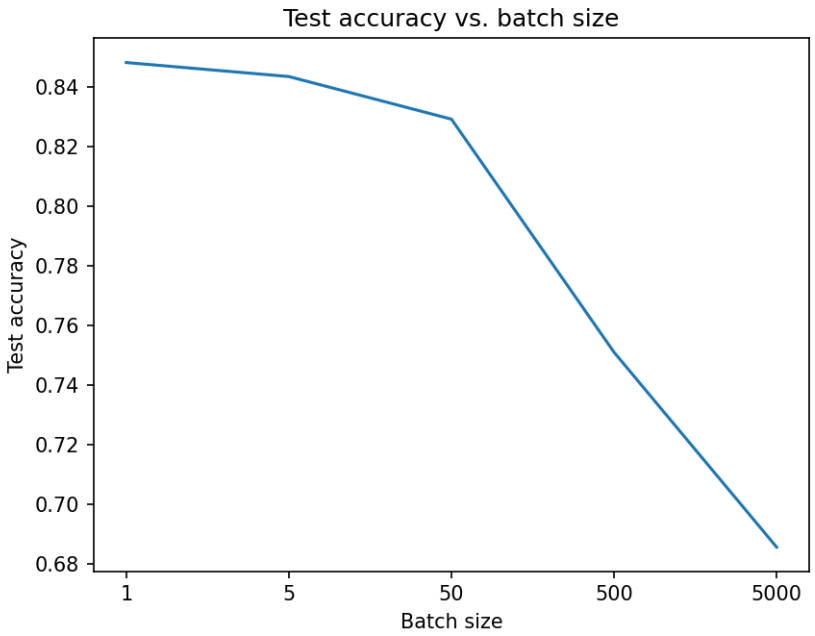
(included in the zip file). (2pts) Plot the training accuracy *vs.* epochs using `plot_train_process.py`. (2pts) For the magnitude check, you should expect the two numbers are of the same magnitude. For the four gradient checks, you should expect the number from the backpropagation to be the same as that from the approximation. For training accuracy, the curve should grow as the number of epochs increases. Some fluctuation in the middle is acceptable.

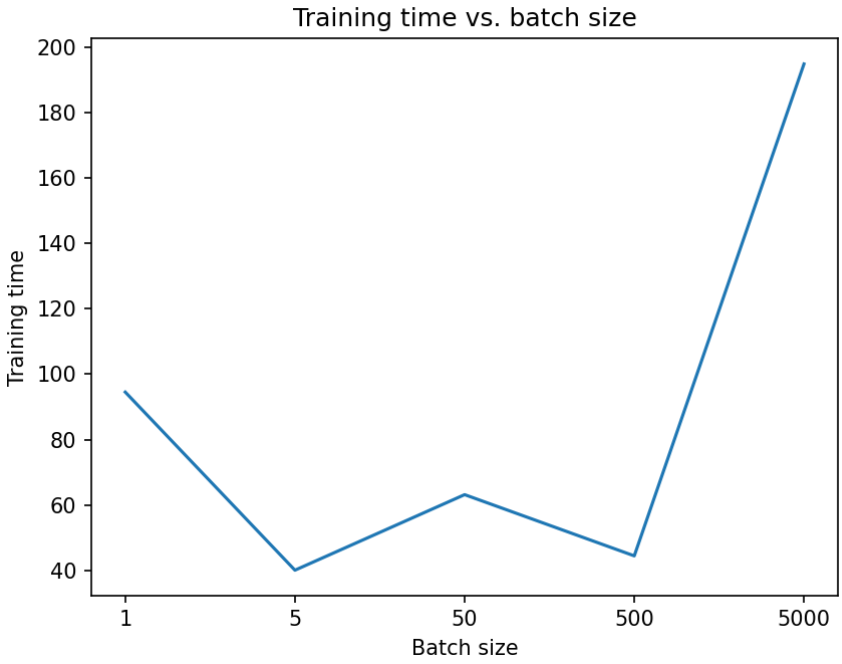
```
python neural_network.py --minibatch_size 5 --check_gradient --check_magnitude
```

```
Training data size: 10000, Validation data size: 2000, Test data size: 10000
Check the magnitude (L1-norm of layer L1) of gradient with batch size 50: 117.085383 and with batch size 5k: 117.085383
Check the gradient of W in the L1 layer from backpropagation: 0.000000 and from approximation: 0.000000
Check the gradient of b in the L1 layer from backpropagation: 0.129579 and from approximation: 0.129579
Check the gradient of W in the L2 layer from backpropagation: 0.004011 and from approximation: 0.004011
Check the gradient of b in the L2 layer from backpropagation: 0.121975 and from approximation: 0.121975
At epoch 1
100%|| 2000/2000 [00:01<00:00, 1206.32it/s]
Training loss at epoch 1 is 6.05685388211968
Training accuracy at epoch 1 is 0.7893
Validation accuracy at epoch 1 is 0.7765
At epoch 2
100%|| 2000/2000 [00:01<00:00, 1502.11it/s]
Training loss at epoch 2 is 5.173094948088359
Training accuracy at epoch 2 is 0.8187
Validation accuracy at epoch 2 is 0.8015
At epoch 3
100%|| 2000/2000 [00:01<00:00, 1453.17it/s]
Training loss at epoch 3 is 4.5864981980615225
Training accuracy at epoch 3 is 0.841
Validation accuracy at epoch 3 is 0.8145
At epoch 4
100%|| 2000/2000 [00:01<00:00, 1416.63it/s]
Training loss at epoch 4 is 4.335750555964994
```

3.2 Batch size (14pts) [6min] We now explore the effect of different batch size on training. Run the command from the previous part with 5 different batch sizes: 1, 5, 50, 500, 5000.

3.2.1 Accuracy and time (4pts) Plot testing accuracy and training time *w.r.t.* batch size using `plot_batch.py`.





3.2.2 Epochs and gradient updates (4pts) What is the number of training epochs required to get the best model for each batch size? How many gradient updates are required get the best model for each batch size? (We define a gradient update to be a single stochastic gradient descent step, where the gradient may have been computed over some mini-batch).

Batch size 1 ran for 16 epochs to get 13 epochs with the best test accuracy. Since it runs 10000 gradient updates per epoch, it takes around 160,000 gradient updates to get the best model.

Batch size 5 ran for 13 epochs to get 10 epochs with the best test accuracy. Since it runs 2000 gradient updates per epoch, it requires about 26,000 gradient updates.

Batch size 50 ran for 46 epochs to get 43 epochs with the best test accuracy. Since it runs 200 gradient up-

dates per epoch, it takes around 9200 gradient updates.

Batch size 500 ran for 42 epochs to get 39 epochs with the best test accuracy. Since it runs 20 gradient updates per epoch, it takes about 840 gradient updates.

Batch size 5000 ran for 161 epochs to get 158 epochs with the best test accuracy. Since it runs 2 gradient updates per epoch, it takes about 322 gradient updates.

3.2.3 Analysis (6pts) Answer the following question based on the previous plots and results:

(i). Does smaller batch size guarantee faster training? Why do you think this is the case?
Generally, a smaller batch size uses less epochs to get to the best test accuracy and takes less time to converge than a larger batch size, thus having a faster training time, as shown by the results from the previous part. This may be the case due to smaller sizes taking much less time to compute small batches such as 5 and 1 which would just be SGD, which is very fast in comparison to GD. Even batch sizes 50 and 500 took around the same amount of epochs, and the batch size of 5000 was the outlier as an overly large batch size. Also, noise may potentially also speed things up by bouncing out of saddle points or local minima instead of being stuck in them.

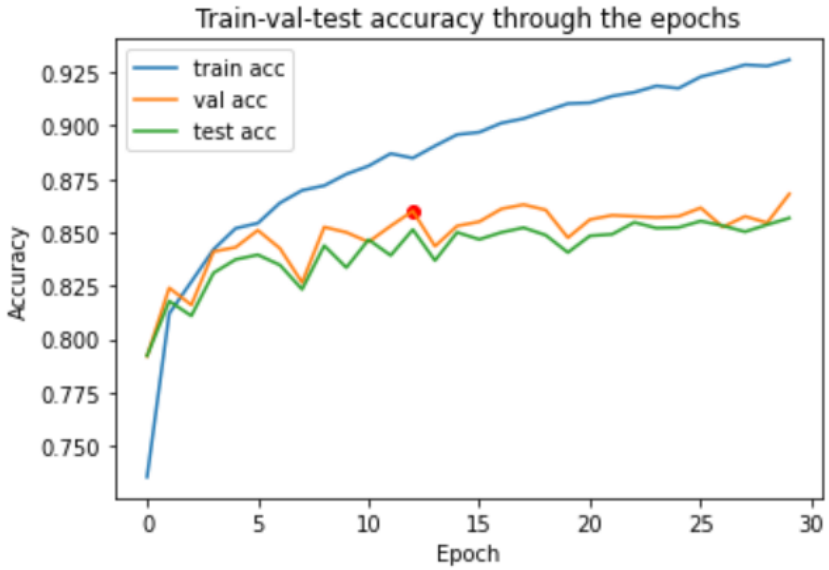
(ii). Does larger batch size imply higher test accuracy? Why do you think this is the case?
No, as the batch size increased the test accuracy actually decreased. This may be the case because each batch requires much more computation compared to smaller batch sizes and each larger batch size contains less noise compared to smaller batches and therefore may waste time being stuck in a local minimum instead of being able to jump out of it. It may also fail to generalize to other models as well.

(iii). Does larger batch size imply less gradient updates to converge? Why do you think this is the case?
Yes, as each gradient update now covers a larger batch size, rather than a smaller batch size requiring many more updates in order to converge. Since it covers larger batches each update, larger batch sizes subsequently take less updates to converge rather than small step sizes such as 1, with a large number of gradient updates per epoch.

Part II - Explore on Colab (12pts)

3.3 The effect of early-stopping (6pts) [3.5min] We provided built-in early-stopping in the starter code, but did not explore the effect that it has on training. In this question, we will explore the effect of early-stopping. To do this, we evaluate the training, validation and test accuracy for each training epoch. We then plot the training, validation and test accuracy throughout the training process to see how early-stopping works. We train 30 epochs on a batch size of 5 without early-stopping.

(i). Plot the training accuracy vs. the number of epochs. On the same graph, plot the validation and test accuracy vs. the number of epochs. Also, mark the point on the test accuracy curve where we previously early-stopped. (We provide the plotting code, you only need to run it). (2pts)



(ii). What is the trend of training and test accuracy after the early-stopped point? (2pts)

After early stopping, the training accuracy continues to increase with the epochs, while test accuracy seems to stagnate and doesn't show a trend of strong increasing like training accuracy, as it stabilizes.

(iii). Based on the plot, what do you think could go wrong if the patience parameter for early-stopping is too small? (Recall that if the patience parameter is set to k epochs, then training will terminate if there is no improvement in the validation accuracy for k epochs in a row.) (2pts)

Based on the plot, it seems that an overly small parameter for early-stopping may result in terminating training too early, resulting in the testing accuracy not being at its maximum point. In this plot, this issue doesn't occur, but it may be possible to have some fluctuation around the same area before increasing test accuracy again if there's some noise.

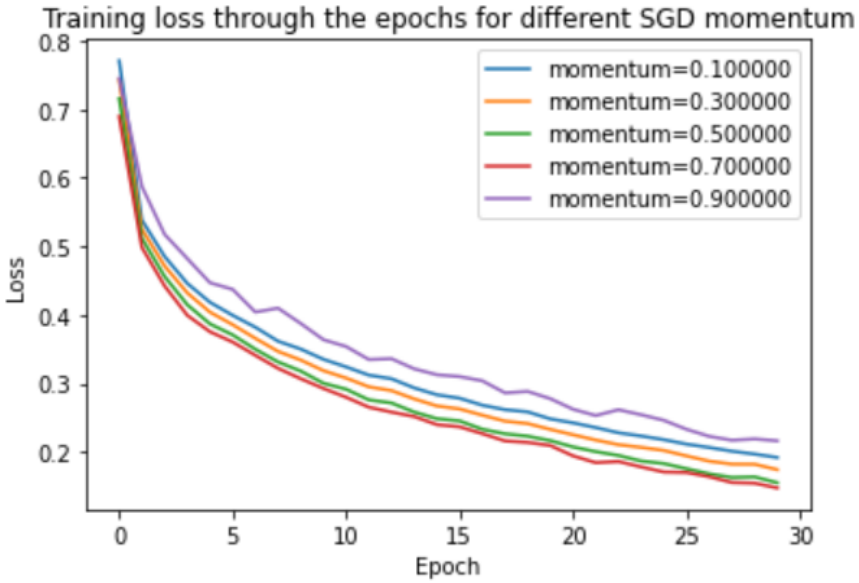
3.4 SGD with momentum (6pts) [14min] We mentioned in class that adding a "momentum" term, which encourages the model to continue along the previous gradient direction helps the network to converge. Concretely, with an initial velocity $\boldsymbol{v} = \mathbf{0}$, we update the gradient by

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} + \nabla F(\boldsymbol{w})$$
$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \boldsymbol{v}$$

where η is the learning rate and α is the factor describing how much weight we put on the previous gradients. $\alpha = 0$ is equivalent to gradient update without momentum.

We provide the `sgd_with_momentum` function with argument α. In this question, we will explore how the momentum factor α effects training loss and test accuracy.

(i). Run the code to call the function `sgd_with_momentum` with α = 0.1, 0.3, 0.5, 0.7, 0.9. We store the training loss and test accuracy returned by the function. We visualize the training loss by plotting 5 curves for the 5 different values of α on the same graph. Each curve has the epoch on the x-axis and the training loss on the y-axis. (4pts)



(ii). Based on this, what is a suitable value of α? Therefore, how should training ideally rely on previous gradients for better convergence? (2pts)

A suitable value of α would be 0.7, since it gives the smallest training loss in the given epochs, since the momentum helps direct the gradient in the right direction, leading to faster convergence. It seems that both overly small or large values of α converge slower, with the largest momentum of 0.9 converging the slowest and having the highest training loss at the end of the epochs, with momentum=0.1 trailing behind it. Therefore, training should rely on previous gradients to an extent, but not too much or too little, but erring slightly more on the relying side, such as 0.7 in this case, allowing the previous gradients to nudge the gradient in the correct direction and converge faster but not relying completely on the previous gradient either.

Part III (Bonus question) - Exploring Out-of-Distribution (OOD) generalization on Colab (15pts) [15min]

3.5 Translation and rotation (15pts) We modify the original test datapoints, by moving up the images of the test set by 4 pixels. By doing this, we create a new translated test set. Run the code to see the original and modified images. You will notice that to a human eye, the new test set is not any more difficult than the original test set.

(i). But can our MLP model still do well on the new test set? What's the test accuracy on the two-layer MLP? (1pt)

Our MLP model does worse on the new test set with a test accuracy of 0.45559999346733093.

(ii). What if we try a different model architecture? For example, in class we saw that convolutional neural networks are good at dealing with image translation. We replace the first linear layer with a convolutional layer of 64 kernels with size 7×7, followed by max-pooling. This can be done with just one or two lines of code in TensorFlow. Calculate the number of parameters of the CNN model and the original 2-layer MLP model (show your calculation), and verify that the CNN has fewer parameters than the MLP model. (3pts)

Parameters of CNN: (7*7*1+1 [bias])*64 + 0 [no params for pooling] + 0 [no params for flattening to 7744 elements] + (7744+1)*10 [dense layer with 10 nodes] = 80,650 parameters
Parameters of MLP: (784+1)*128 [layer 1 with 128 nodes] + (128+1)*10 [layer 2 with 10 nodes] = 100480+1290 = 101,770 parameters
As expected, the CNN parameters of 80,650 are less than the 101,770 parameters of the original 2-layer MLP model, so the CNN model is more efficient in terms of parameters.

(iii). Train the 2-layer CNN on the original training data and test it on both the original and the translated test sets. What is the in-domain test accuracy and the translated test accuracy of the CNN model? Can you provide some intuition behind these numbers? (2pts) [6min]

in-domain test accuracy: 0.8708999752998352
translated test accuracy: 0.5270000100135803
The in-domain test accuracy is better than the translated test accuracy as expected, since the original training data equips the CNN model to better classify on the non-altered test set, while translating the data by 4 may cause a huge difference in whether the model can classify correctly, since the pixels in the translated set may be cut off and not show the full image, unlike the original test set, resulting in a worse test accuracy.

(iv). Going one step further, we can make the CNN deeper by adding one more convolutional layer of 64×128 (64 input channels and 128 output channels) kernels with size 2×2 between the two existing layers, followed by max-pooling. Verify that the deeper 3-layer CNN model has fewer parameters than the 2-layer CNN model (show your calculation). (3pts)

Parameters of CNN: (7*7*1+1 [bias])*64 + 0 [no params for pooling] + (2*2*64+1 [bias])*128 + 0 [no params for pooling] + 0 [no params for flattening to 3200 elements] + (3200+1)*10 [dense layer with 10 nodes] = 68,106 parameters
As expected, the deeper 3-layer CNN model has less parameters than that of the 2-layer CNN model.

(v). What is the in-domain and the translated test accuracy of the deeper 3-layer CNN model? Provide some intuition on why it is better than the 2-layer CNN on the translated set. (3pts) [9min] You will notice that the deeper model takes some time to train. If we trained on GPUs instead of CPUs, training would be much faster. This is because GPUs are much more efficient at matrix computations, which is exactly what neural network training demands.

In-domain test accuracy: 0.8718000054359436
Out-of-domain test accuracy: 0.5652999877929688

It may be better than the 2-layer CNN due to having more layers to extract more features of the translated test data and capture more complexity, so the 3-layers can identify more details of the images and thus have a higher chance of classifying the image correctly and a higher test accuracy.

Next, we create 3 more OOD test sets by rotation. We rotate the images in the original test set by 90, 80 and 270 degrees.

(vi). Provide the test accuracy of the 2-layer MLP model, the 2-layer CNN model and the 3-layer CNN model on the three rotation test sets. Are the 2-layer CNN and the 3-layer CNN still doing well? (3pts)

Rotation 90 degrees:
MLP model accuracy: 0.020800000056624413
2-layer CNN model accuracy: 0.05009999871253967
Deeper 3-layer CNN model accuracy: 0.052799999713897705

Rotation 180 degrees:
MLP model accuracy: 0.18690000474452972
2-layer CNN model accuracy: 0.21649999916553497
Deeper 3-layer CNN model accuracy: 0.04650000110268593

Rotation 270 degrees:
MLP model accuracy: 0.05530000105500221
2-layer CNN model accuracy: 0.04740000143647194
Deeper 3-layer CNN model accuracy: 0.04650000110268593

Both forms of CNN are not doing well on the rotated test tests. The 2-layer CNN model seems to do better than the 3-layer by a good amount for the 180-rotated test set and slightly better for the 270-rotated test set. However, overall all test accuracies, whether MLP or 2-layer or 3-layer CNN models, have massively decreased due to the rotation in the test sets.

Deliverables for Problem 3: Code for 3.1 as a separate Python file `neural_networks.py`. Screenshot for 3.1. Plots for part 3.1, 3.2, 3.3 and 3.4. Number of epochs and gradient updates for part 3.2. Analysis and explanation for parts 3.2, 3.3, 3.4. For the optional bonus question 3.5, submit the test accuracy on in-domain and OOD test sets of the three models, the number of parameters of the three models, and the explanations we ask for.