Homework #5

ECE 461/661: Introduction to Machine Learning
Prof. Carlee Joe-Wong and Prof. Gauri Joshi
Due: April 1, 2020 at 8:59PM PT / 11:59PM ET

Please remember to show your work for all problems and to write down the names of any students that you collaborate with. The full collaboration and grading policies are available on the course website: https://www.andrew.cmu.edu/course/18-661/.

Your solutions should be uploaded to Gradescope (https://www.gradescope.com/) in PDF format by the deadline. We will not accept hardcopies. If you choose to hand-write your solutions, please make sure the uploaded copies are legible. Gradescope will ask you to identify which page(s) contain your solutions to which problems, so make sure you leave enough time to finish this before the deadline. We will give you a 30-minute grace period to upload your solutions in case of technical problems.

IMPORTANT: Please see the end of this homework for important notes on the submission of your solution to Problem 4. We are accepting it separately to allow auto-grading from Gradescope.

1 Perceptrons *[10 points]*

A perceptron learns a binary linear classifier with weight vector $\mathbf{w}$ (we omit the bias weight $\mathbf{b}$ for simplicity). That is, it learns a classifier that predicts $\hat{y} \in \{+1, -1\}$ as

$$\hat{y} = \text{sign}(\mathbf{w}^T \mathbf{x}_t)$$

The perceptron training algorithm we present is based on an *online learning* model, that is, it processes the data sequentially (one sample at a time), updating the weight vector at each step. The perceptron algorithm is as follows:

1 Set $t = 0$ and initialize the weight vector to all zeros: $\mathbf{w}_0 = \mathbf{0}$.

2 Given an example $\mathbf{x}_t$, predict $\hat{y}_t = \text{sign}(\mathbf{w}_t^T \mathbf{x}_t)$.

3 Let y_t be the true label of $\mathbf{x}_t$. If we make a mistake, i.e, $y_t \neq \hat{y}_t$ we update the weights as

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + y_t \mathbf{x}_t$$

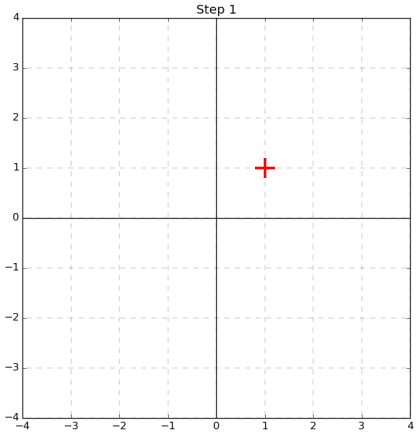
If we don't make a mistake, then we don't update the weight.

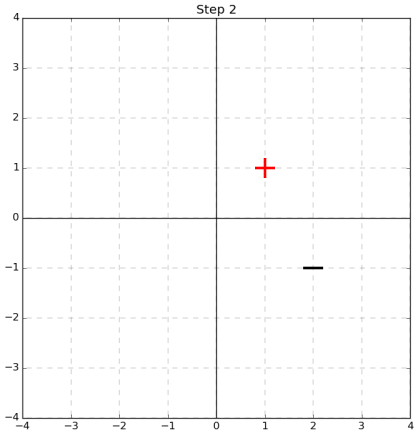
4 If there are more samples to train on, then $t \leftarrow t + 1$, go back to step 2.

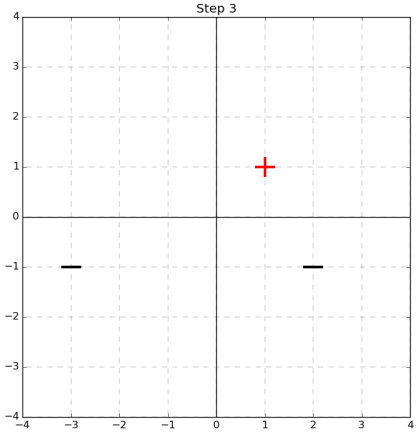
In the context of neural networks, a perceptron is an artificial neuron using the Heaviside step function as the activation function. As a linear classifier, the single-layer perceptron is the simplest feedforward neural network. In this problem, we go through the perceptron algorithm step by step for a toy problem. Imagine that we have $\mathbf{x}_t \in \mathbb{R}^2$, and we encounter the following data points in order:

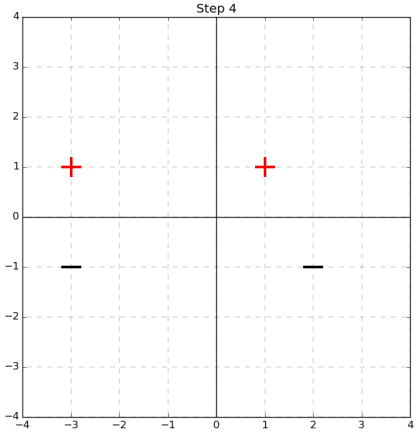
$\mathbf{x}[1]$	$\mathbf{x}[2]$	y
1	1	1
2	-1	-1
-3	-1	-1
-3	1	1

a. ***[8 points]*** Starting with $\mathbf{w} = [0 \quad 0]^T$, use the perceptron algorithm to learn on the data points in the order from top to bottom. Draw the perceptron's linear decision boundary after observing each data point is given by the graphs below. Be sure to show which side is classified as positive.

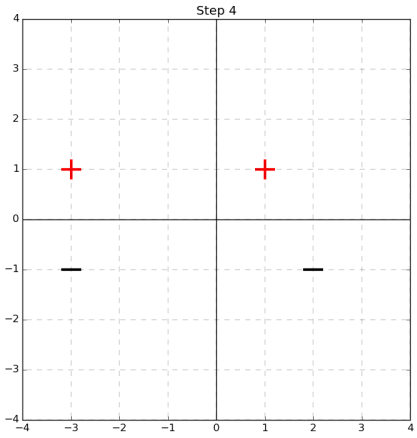








b. ***[2 points]*** Does our learned perceptron maximize the margin between the training data and the decision boundary (i.e., the minimum distance between a training point and the boundary)? If not, draw the linear decision boundary that maximizes the margin on the graph below.



2 Neural Networks as Universal approximators *[20 points]*

In this problem you will experiment with designing a Multi Layer Perceptron (MLP) in the Tensorflow Playground. For each of the following questions, you will need to submit a screenshot including the network configuration, test loss, and your parameter settings (including the DATA column on the left). For each part of this question, you will be asked to classify a different dataset (available in the DATA column).

Figure 1 is a sample screenshot similar to what you are going to submit in your writeup on Gradescope. You can change the network structure (number of layers, and number of neurons in each layer), and adjust the following hyperparameters: learning rate, activation, regularization type and regularization rate. Please leave other parameters fixed: leave ratio of training to test data to 50%, leave Noise at 0, and Batch size at 10. After you configure your network and parameter settings, you will use the "play" button to train the network and check whether the test loss can drop below the 0.1 threshold.

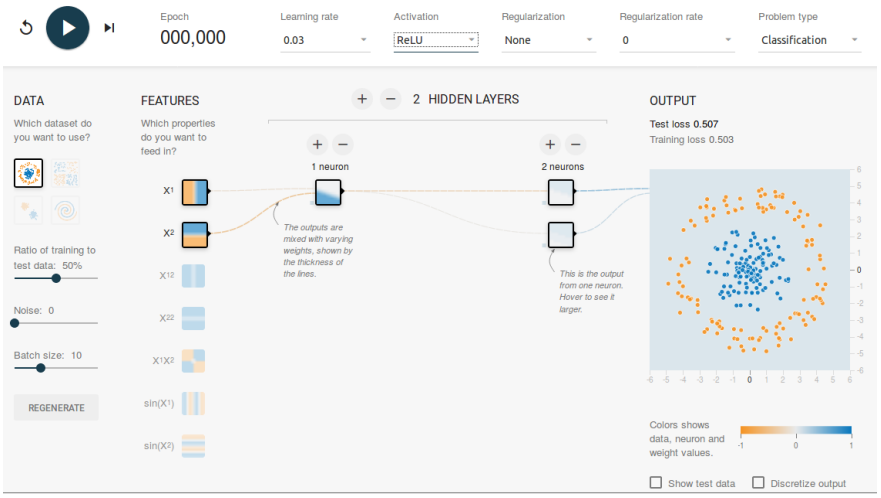


Figure 1: Sample screenshot

a. *[4 points]* Circle data set: Can you obtain a test loss of less than 0.1 using only the raw input features? i.e., using only X_1 and X_2 as features? Please share a screenshot of your network as described.

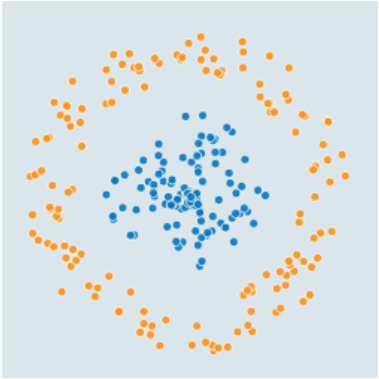


Figure 2: Circle data set

Now suppose you use features X_1^2 and X_2^2, instead of raw features X_1 and X_2. Your objective now is to obtain a test loss of less than 0.1 in as few neurons as possible. How many neurons would you need to classify the dataset? Justify your answer.

b. *[4 points]* Exclusive-Or data set: Consider the following method to train your neural network model with raw features X_1, X_2. Set the activation function to be tanh, use a learning rate of 0.03 and don't use any regularization. Use a neural network with two hidden layers with the first hidden layer having 3 neurons and the second layer having 2 neurons. Do you obtain a test loss of less than 0.1?

Now change the learning rate to 0.3 and report the final test loss. Repeat the same for learning rate = 3 now. How does the test loss change as you increase the learning rate? Briefly explain why.

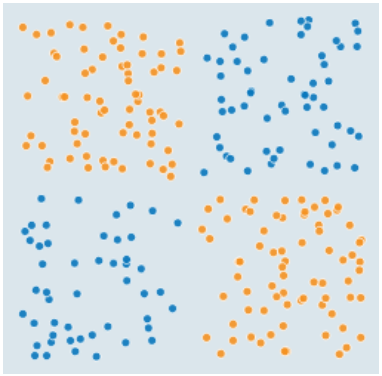


Figure 3: Exclusive Or data set

Find another network architecture that also gives you a test loss of less than 0.1 on this dataset. Please share a screenshot of this network as described earlier.

c. *[4 points]* Gaussian data set: Can you obtain test loss of less than 0.1 using only the raw inputs? i.e., using only X_1 and X_2? Try to use as few neurons as possible. Report the number of neurons used in your model and justify your answer.

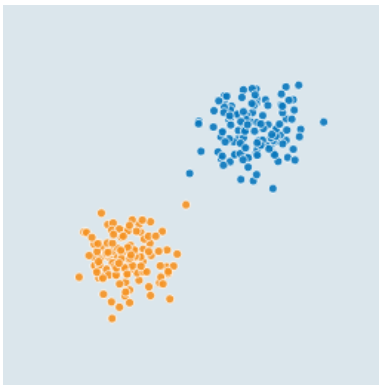


Figure 4: Gaussian data set

Now suppose you use features X_1^2, X_2^2 instead of X_1, X_2. Can you still obtain a test loss of less than 0.1? Justify your answer.

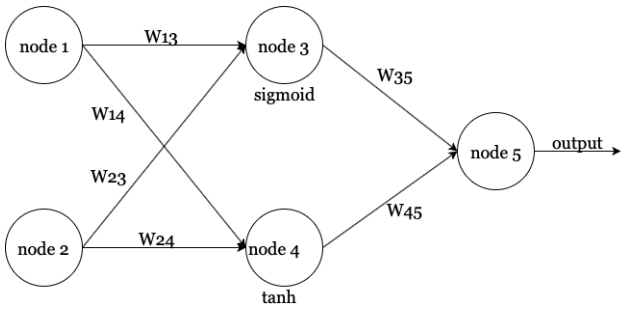
d. *[8 points]* Spiral data set: Can you obtain test loss of less than 0.1 using only the raw inputs? i.e., using only X_1 and X_2? Please share your screenshot of the network as described.

Figure 5: Spiral data set

Now you are allowed to use any/all of the 7 different input features (X_1, X_2, X_1^2, X_2^2, X_1X_2, $\sin(X_1)$, and $\sin(X_2)$). Can you obtain test loss of less than 0.1 for the Spiral data set (Figure 5) with fewer neurons? Please share your screenshot of the network as described.

3 Chain Rule *[10 points]*

Consider the following network with 2 input nodes, 2 hidden nodes and 1 output node. Nodes 3 and 4 use sigmoid and tanh activations respectively, and the other nodes use linear activation (i.e., their inputs are equal to their outputs).



a. *[3 points]* Forward propagation. Compute the output of node 5 with the following settings:

1. The inputs of node 1 and node 2 are 1.0 and -1.0 respectively.
2. $W_{13} = 0.7$, $W_{14} = -0.9$, $W_{23} = 1.2$, $W_{24} = -1.1$, $W_{35} = 0.5$, $W_{45} = -0.3$
3. There is no bias term in this network.
4. $tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$ and $\sigma(x) = \frac{1}{1+e^{-x}}$

b. *[7 points]* Run one iteration of backward propagation of the network. We use the same settings as above. You will also need the following information to compute the gradients:

1. The loss function is $L(y, \hat{y}) = (\hat{y} - y)^2$, where y is the output of node 5 and $\hat{y}$ is the desired output. In this part, we let $\hat{y} = 1.0$.
2. The learning rate is set to 0.1.

After one iteration, report the weight values in the table below (keep only 2 decimal digit):

Table 1: Weights in the network

W_{13}	W_{14}	W_{23}	W_{24}	W_{35}	W_{45}

4 Neural Networks for MNIST Digit Recognition *[40 points]*

In this problem, you will implement a neural network to classify handwritten digits using raw pixels as features. You will be using the classic MNIST digits dataset. The state-of-the-art error rate on this dataset using various learning methods is around 0.5% (see this leaderboard for details). However, you will be implementing a simple neural network and should, with appropriate hyperparameter settings, get a test error of approximately 5% (accuracy above 95%) or better. Specifically, you will implement a neural network with an input layer, one or more hidden layers, and an output layer. Please pay careful attention to the following implementation details.

4.1 Layer Implementation *[10 points]*

In this section, you will be implementing three types of layers for your network: a dense layer that is fully connected with the inputs (the input layer), followed by a ELU layer (a hidden layer), and ending with a SoftmaxCrossEntropy layer (the output layer). We have implemented the ELU layer for you, so you can take it as given.

4.1.1 Dense Layer

A Dense layer is fully connected with the input features. You are going to implement the forward propagation, backward propagation, parameter initialization and parameter update in the Dense Layer. More specifically, you need to implement the following functions:

- `Dense.__call__(x)` This function computes the forward propagation for the dense layer given a batch of inputs `x`. Complete this function in `numpynet/layer.py`.
- `Dense.bprop(grad)` This function computes the backward propagation through the Dense layer, using the chain rule. The input of this function is the gradient coming from the previous layer. The use of this function to implement the backpropagation should be similar to the following:

```
import numpynet.layer.Dense as Dense
# structure: input --> hidden layer 1 --> hidden layer 2 --> output
hidden_layer_1 = Dense(10, 20)
hidden_layer_2 = Dense(20, 30)

# Backward Propagation through layers
grad = YOUR_LOSS_FUNCTION.bprop(*args, **kwargs)
grad = hidden_layer_2.bprop(grad)
grad = hidden_layer_1.bprop(grad)
```

  Complete this function in `numpynet/layer.py`.

- `Dense.update(lr)` This function updates all parameters inside the dense layer using stochastic gradient descent (SGD). The input of this function is the learning rate `lr`. Hint: We will be dealing with a batch of data in section 5.3, therefore, you need to sum the changes in the weight parameters dw from

all training instances in a batch and divide it by the number of instances in a batch, a.k.a. the batch size. The same thing to be done with bias.

Complete this function in `numpynet/layer.py`.

- `Dense._parameter_init()` This function initialize all the weights and bias of this layer. Initialization is very important for model training. A bad initialization can cause a training failure. For this part, you are going to implement Xavier initialization, which means your weights should be sampled from a centered normal distribution with standard deviation $\sqrt{\frac{2}{dim_i+dim_o}}$ where dim_i is the number of input dimensions and dim_o is the number of output dimensions. Your bias term should be initialized with all zeros. Complete this function in `numpynet/layer.py`.

4.1.2 ELU Layer

The Exponential Linear Unit (ELU) is a variation of ReLU activation that eliminates the zero gradient problem when the input is less than 0. Using ELU as an activation function tends to produce faster convergence with more accurate results than using ReLU. Unlike other activation functions, ELU has an extra α constant, which should be positive number (An usual choice of α is 1 or 0.9). The expression of an ELU unit is shown as follows:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$$

The following functions in `numpynet/layer.py` are already implemented for you (you do not have to modify these):

- `ELU.__call__(x)` This is the forward propagation to compute the activation.
- `ELU.bprop()` This is the backward propagation through the ELU activation function.

4.1.3 SoftmaxCrossEntropy

This layer combines the Softmax function together with the Cross Entropy Function. The inputs of this layer are the logits and the true one-hot encoded labels. Recall the softmax function

$$y_i = \frac{e^{z_i}}{\sum_j^D e^{z_j}},$$

where D is the dimension of the input. The Cross Entropy function is given by

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i^D \hat{y}_i \log y_i,$$

where $\hat{\mathbf{y}}$ is the vector of true labels and $\mathbf{y}$ is the output probability distribution on the labels. Plugging the softmax into the cross entropy equation, you will have the forward propagation for this layer. Based on the above equation, complete the `SoftmaxCrossEntropy.__call__(logits, labels)` in `numpynet/layer.py`. During the implementation, notice that you are going to compute the log following the exponential operation, which may cause overflows. One solution to this is to use the log-sum-exp trick:

$$\log \sum_i^n e^{x_i} = max(\mathbf{x}) + \log \sum_i^n e^{x_i - max(\mathbf{x})}$$

You can find the proof of this formula here. The backward propagation through the SoftmaxCrossEntropy layer can be found by computing the first-order derivatives:

$$\begin{aligned}\frac{\partial L}{\partial z_i} &= \frac{\partial}{\partial z_i}[-\sum_i^D \hat{y}_i \log \frac{e^{z_i}}{\sum_j^D e^{z_j}}] = \frac{\partial}{\partial z_i}[-\sum_i^D \hat{y}_i z_i + \sum_i^D \hat{y}_i \log \sum_j^D e^{z_j}] \\ &= \frac{\partial}{\partial z_i}[-\sum_i^D \hat{y}_i z_i + \log \sum_j^D e^{z_j}] \text{ (only one } \hat{y}_i \text{ is 1)} \\ &= -\hat{y}_i + \frac{e^{z_i}}{\sum_j^D e^{z_j}} = -\hat{y}_i + y_i.\end{aligned}$$

Based on the above derivation, complete the `SoftmaxCrossEntropy.bprop()` in `numpynet/layer.py`. In case you can not find the question in this section, here is the summary:

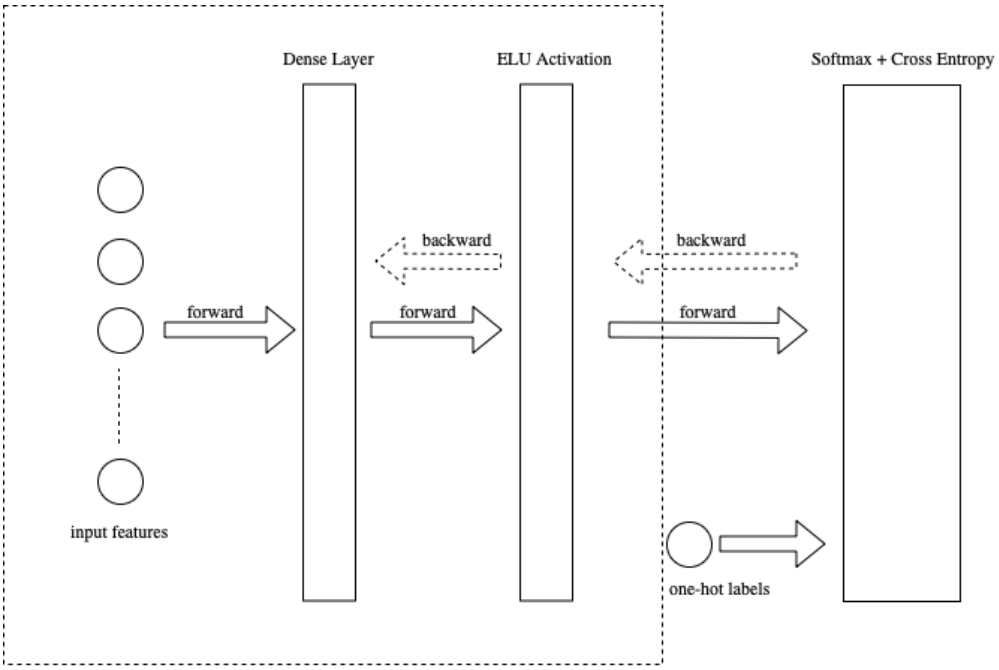
- Implement the `SoftmaxCrossEntropy.__call__(logits, labels)` function.
- Implement the `SoftmaxCrossEntropy.bprop()` function.

4.2 Build the model *[10 points]*

After you have implemented all layer objects, you can build you own model. You will be using the Dense layer, ELU activation, ReLU activation with the Loss function SoftmaxCrossEntropy to recognize the digits. The input should take a matrix of size (B, 784) where B is your batch size and 784 is the number of features for each image of MNIST. Design a model with 3 hidden layers. The number of neurons in the three layers are $[256, 64, 10]$ (i.e., 256 neurons in first hidden layer, 64 neurons in second and 10 neurons in third). Each hidden layer is followed by ELU activation as shown in the graph below. The output of last layer should connect to the SoftmaxCrossEntropy layer that you built above to compute the loss. Complete the in `Model()` object in `hw5.py` by implementing:

- `Model.build_model()`, which builds all layers using the layer object from `numpynet`
- `Model.__call__()`, which runs the forward propagation through all layers.
- `Model.bprop(logits, labels, istraining=True)`, which runs the backward propagation through all layers to compute the gradients for all parameters of the model.
- `Model.update_parameters(lr)`, which updates all trainable parameters in each layer. The input of this function is the learning rate.

The graph below shows one way to connect the layers to build your network. You do not need to follow this architecture to build your own model.



Model Architecture

4.3 Training and Evaluation *[10 points]*

4.3.1 Dataset

Type the following command in the terminal to get the dataset

```
$ cd hw5_release
$ tar -xvf dataset.tar.gz
```

You will be training the model which you just built in Section 3.2. We will be using Mini batch Stochastic Gradient Descent (SGD) to train the model. During training, instead of applying SGD on one single data point, you apply SGD on multiple data points (called a *mini batch* of points).

4.3.2 Helper Functions

For the convenience, we have provided you with several helper functions. A sample use can be shown as follows. You do NOT have to use the following functions if you have your own implementation.

```
from numpynet.utils import Dataloader, one_hot_encoding, load_MNIST, save_csv
from numpynet.function import Softmax


# Use load_MNIST() to load the dataset.
# You can specify the path of the dataset if it
# is under the 'dataset/' by passing the argument into the load_MNIST such as
# load_MNIST(path='dataset/')
train_X, train_y= load_MNIST(path='dataset/', name='train')
val_X, val_y= load_MNIST(path='dataset/', name='val')
test_X = load_MNIST(path='dataset/', name='test')
```

```
# Use one_hot_encoding() to create one-hot version
# of your labels
one_hot_train_y = one_hot_encoding(train_y)

# Create a generator that iterates over the features and labels
# If shuffle=True, it will shuffle the dataset every time before iteration
train_dataloader = Dataloader(X=train_X, y=train_y, batch_size=32, shuffle=True)

#Sameple use of train dataset
for i, (features, labels) in enumerate(train_dataloader):
    #do something

# Create a generator that iterates over the features only
# DO NOT shuffle test dataset
test_dataloader = Dataloader(train_X, batch_size=32, shuffle=False)

#Sameple use of test dataset
for i, features in enumerate(test_dataloader):
    #do something

# Save an np.ndarray variable into submission.csv
x = np.array([0, 1])
save_csv(x)

# You can use Softmax() to do the inference if you want, or you can
# use your own implemenation of Softmax function
x = Softmax(x, axis=-1)
```

For detailed implementation, please refer to `numpynet/utils.py`

4.3.3 Training Steps

Complete the `train()` and `inference()` functions in `hw5.py` following:

- **Step 1** Train the model with 200 epochs. Record your training loss per instance (total loss / size of dataset) and training accuracy for each epoch (NOT for each batch). You should run the code in the order of forward propagation, backward propagation, and finally parameter update. For each iteration, you will feed the model with a mini batch. You can choose your own batch size.
- **Step 2** For every two epochs, run the inference on the validation dataset and record your validation loss per instance and validation accuracy. Do **NOT** update the parameters using the validation dataset. This validation serves to monitor the model in case of underfitting and overfitting.

Notice that training MNIST will NOT require any GPU resources because it is very simple and the dataset is fairly small. Each epoch should not cost you more then 2 seconds if your model is not too deep. If you find one epoch takes more than 5 seconds, you probably want to revisit your code before training the whole dataset. We suggest that you use a batch size of 32 and a learning rate of 0.2 (which changes to 0.02 after 160 epochs). The suggested hyperparameter settings are present in the *main* function of *hw5.py*. You are free to use your own hyperparameter settings as well.

After training, **plot and save the following TWO graphs**. In the first graph, plot two curves, the training loss and validation loss, with the number of training epochs on the x-axis. In the second graph, plot two curves, training accuracy and validation accuracy, against the number of training epochs. Correctly label your graphs with proper titles, the names of the axes, and legends of each curve. Save the plotted graphs into two PNG files.

4.4 Testing *[10 points]*

Test your trained model on the test dataset. Use your `inference()` implementation to finish testing. The size of test dataset is 10,000, so your prediction result should be a numpy array of shape (10000,) filled with labels, for example: `array([0, 1, 2, ...])`. Use the provided `save_csv()` helper function to save the prediction result of the test dataset into `submission.csv`. (You do not need to specify the name; the function will save the input numpy array into `submission.csv`.)

Important tips:

- Do NOT shuffle the test dataset;
- Gradescope will compare your predictions with true labels. You will get full scores if your accuracy is above 95%.

5 Submission Instructions

- Summarize you answers into a single PDF file excluding any code or files from Problem 4 and submit to Assignment HW5 on Gradescope.
- Create a zip file with the following structure and submit the zip file to Assignment HW5-coding on gradescope. For this part, you can submit as many times as you like before the deadline. Please do NOT include cache files like `numpynet/__pycache__/function.cpython-37.pyc`, files generated by the OS like `__MACOSX/numpynet/._.DS_Store` or your vscode settings json files.

```
andrew_id_hw5code.zip
├── numpynet
│   ├── __init__.py
│   ├── layer.py
│   ├── utils.py
│   └── function.py
├── hw5.py
├── submission.csv
├── loss_curve.png
└── accuracy_curve.png
```