UNIVERSITY OF PENNSYLVANIA

ESE 546: PRINCIPLES OF DEEP LEARNING

HOMEWORK 1

Read the following instructions carefully before beginning to work on the homework.

- You will submit solutions typeset in LaTeX on Gradescope (strongly encouraged). You can use hw_template.tex on Canvas in the "Homeworks" folder to do so. If your handwriting is unambiguously legible, you can submit PDF scans/tablet-created PDFs.
- Clearly indicate the name and Penn email ID of all your collaborators on your submitted solutions.
- Start a new problem on a fresh page and mark all the pages corresponding to each problem. Failure to do so may result in your work not graded completely.
- For each problem in the homework, you should mention the total amount of time you spent on it. This helps us keep track of which problems most students are finding difficult.
- You can be informal while typesetting the solutions, e.g., if you want to draw a picture feel free to draw it on paper clearly, click a picture and include it in your solution. Do not spend undue time on typesetting solutions.
- Each problem/sub-problem will list the number of points it is worth. In addition, it will list the point breakdown between the manually graded answers in your PDF Report **(Report)** and the autograded code **(Code)** in your python files. Please note that some sub-problems have both manually graded and autograded sections. **Make sure your submission includes both.**
- Run your code with **Python >= 3.9** to avoid incompatibility issues with the autograder.
- **For each programming problem, you will be given a starter code template**. It is important to use this template as it will help make sure that your code will work with the autograder. In addition, it provides the code to export any additional files required by the autograder. The files provided are:
  - Problem 1: SVM_skeleton_code.py. You must rename this file to pennkey_hw1_problem1.py for submission.
  - Problem 3: NN_skeleton_code.py. You must rename this file to pennkey_hw1_problem3.py for submission.
- This assignment will have three Gradescope submission assignments. **You must submit to all three.** The files to submit for each section are:
  - HW1 PDF
    * The PDF file with all of your written answers. Please note that you will be required to page match your answer to each question. Failure to do so may result in your work not graded completely.
  - HW1 Problem 1 Code:

    - pennkey_hw1_problem1.py
    - grid_search_svm.pkl
  - HW1 Problem 3 Code:
    - pennkey_hw1_problem3.py
    - self_NN_training_error.pkl
    - self_NN_training_loss.pkl
    - self_NN_validation_error.pkl
    - self_NN_validation_loss.pkl
    - pytorch_NN_training_error.pkl
    - pytorch_NN_training_loss.pkl
    - pytorch_NN_validation_error.pkl
    - pytorch_NN_validation_loss.pkl
    - linear.h5
    - pytorch_nn_weights.pth
- Note, we will not accept .ipynb files (i.e., Jupyter notebooks), you should only upload .py files. If you are using Google Colab to do your homework, you can export the notebook to a .py file.
- **This is very important**. Note that your code files will be **autograded** or run by the instructors, so your code should be such that it can be executed independently without any errors to create all output/plots required in the problem.
- When you submit your code to Gradescope, you will see the results of the autograder when it has finished running (typically less than 10 minutes). You can resubmit your code as often as you like until the assignment deadline.
- The PDF can also be submitted multiple times, but it will not be graded until after the submission deadline.

Credit The points for the problems add up to 140. You only need to solve for 100 points to get full credit, i.e., your final score will be min(your total points, 100).

Problem 1 (60 points: 21 Code, 39 Report). In this problem, we will train a support vector machine (SVM) to predict handwritten digits from the MNIST dataset.

An SVM solves an optimization problem for maximizing the margin between two classes. Suppose that we have a binary classification problem where (x_i, y_i) are the data and ground-truth labels respectively and $y_i \in \{-1, 1\}$. We would like to find a hyper-plane that separates the data such that all examples with labels $y_i = +1$ are on side and all examples with labels $y_i = -1$ are on the other side. This involves solving the problem

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\|\theta\|^2 \\ \text{subject to} \quad & y_i(\theta^\top x_i + \theta_0) \geq 1 \quad \forall i = 1, \dots, n; \end{aligned} \tag{1}$$

here θ_0 is the offset parameter and θ is the hyper-plane. You can eliminate the offset parameter by appending a 1 to the data, i.e., feeding in $x' = [x, 1]$ as the data with the same labels.

Code:

For this question, download the skeleton code that we provide: SVM_skeleton_code.py. The skeleton code contains all the python packages you will need to complete this question. Do not update the python package at the top of the skeleton code or add new ones. Doing so will cause the autograder to fail your submission on Gradescope. Complete the starter code methods provided without changing their signatures or return values. Finally, you must rename this file to yourpennkey_hw1_problem1.py for final submission on Gradescope.

Note: You can code on your local machine but if you need more RAM, use Google Colab. If you use Colab, make sure to integrate your code back to the Python starter code format before submitting to Gradescope.

We will train an SVM using the "scikit-learn" library. You can install python packages using 'pip install", for example:

```
[local] pip install scikit-learn scikit-image
[colab] !pip install scikit-learn scikit-image
```

(a) First, download MNIST dataset that we will use to train and test the SVM. Implement the method **get_data()** where you will download the MNIST image data using fetch_openml(). Then, convert the data to numpy. Next, to check what the data looks like and whether you have downloaded the data correctly, plot one of the images by calling plt.imshow() within the method. However, note that the downloaded data in x are in the form of a vector of length 784. This is really the flattened matrix of size 28×28. Hence, to plot the data as an image, we have to first reshape the vector to an array with shape 28×28. For example,

```python
import matplotlib.pyplot as plt
a = x[0].reshape((28,28))
plt.imshow(a)
```

If you see an image of a handwritten digit, you have successfully downloaded the MNIST dataset. Finally, return the MNIST numpy data and target.

(b) Fitting SVMs requires a decent amount of RAM. We will therefore downsample the original 28×28 images to 14×14. Implement the method **resize()** where you will downsample the 28x28 images to 14x14 using cv2.resize() method. To check what the downsampled image looks like compared to the original image we saw in (a), call plt.imshow() on one of the resized data.

```
# example code for downsampling one image
import cv2
b = cv2.resize(b, (14,14))
plt.imshow(b)
```

(c) Next, implement the method **subsample()**. Here, you will create a dataset of 1000 random samples for each of the ten digits, resulting in a total of 10,000. We are doing this step simply to reduce the amount of time it takes to fit the SVM).

(d) To put the data preprocessing steps for the MNIST data above into one pipeline, implement the method **data_preprocessing()**. In this function, you will call subsample() to get the desired number of 10,000 samples and resize() to downsample your data. We will then construct our actual training dataset (80%) and validation dataset (20%) using train_test_split(). The function returns the training and validation sets we will use to train an SVM.

(e) Next, implement the method **get_number_of_support_samples()** where you find the total number of support samples for a given SVM classifier.

(f) (11 points) We are finally ready to train an SVM. Implement the method **train_test_SVM()** to train an SVM classifier on `x_train`, `y_train`. Then, test the trained SVM on `x_val`, `y_val`. Create the SVM classifier in scikit-learn using

```
classifier = svm.SVC(C=1.0, kernel='rbf', gamma='auto')
```

Fit the SVM classifier to the training dataset and predict the labels of the validation dataset using the trained classifier. To get an insight into the trained model, we will look at its validation accuracy, confusion matrix, and support vector percentage. More specifically, check the performance of the trained classifier on the validation set using clf.socre() to get validation accuracy. Next, obtain the resulting confusion matrix using the confusion_matrix() method. Finally, obtain the ratio of the total number of support samples (call get_number_of_support_samples() you implemented in (e)) to the total number of training samples for your trained classifier.

(g) (5 points) Next, we will explore different hyperparameters to improve your SVM model. Implement the method **grid_search_SVM()**. In this function, use the sklearn.model_selection.GridSearchCV function to pick a better value than the default one for the hyperparameters C, gamma, and kernel. Show all the hyper-parameters tried by the method and their accuracies. Get the best hyperparameters and SVM with the highest validation accuracy using best_estimator_ attribute. Save the best SVM as *grid_search_svm.pkl* and upload it to the autograder.

(h) (5 points) **The following part is computationally intensive.**

The default kernel in svm.SVC is a radial basis function. The MNIST dataset consists of images and since images have local regularities we can build a better classifier by exploiting them. The mammalian visual cortex consists of cells that can be modeled as Gabor functions (named after Dennis Gabor, a

Hungarian physicist who invented holography). See https://en.wikipedia.org/wiki/Gabor_filter for examples.

Let us represent each image as a function $I(x, y)$, this function gives the intensity at pixel location (x, y). A Gabor filter is given by a function

$$g(x, y;\ \theta, F, \sigma_x, \sigma_y) = \exp\left(i\ 2\pi F p\right)\ \exp\left(-\pi\left(\frac{p^2}{\sigma_x^2} + \frac{q^2}{\sigma_y^2}\right)\right)$$

where $p = x\cos\theta + y\sin\theta$ and $q = -x\sin\theta + y\cos\theta$. First, note that this filter is a complex function. Convolving the original image $I(x, y)$ with the filter $g(x, y)$ will result in two sets of coefficients, one real and the other imaginary. The parameters we will be concerned with are:

- F this is the spatial frequency of the filter,
- θ the rotation angle of the Gaussian,
- σ_x, σ_y: standard deviation of the kernel in the X and Y directions, and
- the parameter "bandwidth" in the code below is inversely related to the standard deviation fixed the frequency.

You can read this webpage for a simple introduction to these filters (this is given in the OpenCV format). You can also read this more mathematical tutorial on Gabor filters which is given in the scikit-image format that we discussed above.

We will use the scikit-image library which implements a smaller machine learning-specific set of image processing functions. Alternatively, you can also use the cv2.getGaborKernel function in OpenCV. But for this homework, we will use scikit-image library's gabor_kernel and gabor functions provided in your skeleton code.

First, prepare a training dataset of 100 samples per class and a validation set of 100 images per class using the function subsample() your wrote previously given the training and validation datasets you have resized to 14 x 14 and used previously.

Next, before we begin training an SVM with Gabor filter convolved images, we want to first visualize how the gabor kernel looks and the result of convolving and image with the filter. An example code is below:

```
from skimage.filters import gabor_kernel, gabor
import numpy as np

freq, theta, bandwidth = 0.1, np.pi/4, 1
gk = gabor_kernel(frequency=freq, theta=theta, bandwidth=bandwidth)
plt.figure(1); plt.clf(); plt.imshow(gk.real)
plt.figure(2); plt.clf(); plt.imshow(gk.imag)

# convolve the input image with the kernel and get co-efficients
# we will use only the real part and throw away the imaginary
# part of the co-efficients
image = x_train[0].reshape((14,14))
coeff_real, _ = gabor(image, frequency=freq, theta=theta,
                      bandwidth=bandwidth)
plt.figure(1); plt.clf(); plt.imshow(coeff_real)
plt.figure(2); plt.clf(); plt.imshow(image)
```

Add the resulting images to your report with your observations.

Next, we want to diversify the values for for F, θ and bandwidth parameters to increase the number of filters and create 16 different gabor kernels and see how the filter changes in shape and size. Use the following values:

```
theta = [np.pi/4, np.pi/2, 3 * np.pi / 4, np.pi]
frequency = [0.05, 0.25]
bandwidth = [0.1, 1]
```

This gives a total of 16 filters in the filter-bank. Out of the 16 filters, plot 8 of real and imaginary coefficients as images to see that it gives you a good spread of different filters. You want a diverse filter bank that can capture different rotations and scales. Add the images to your report with your observations.

Now, instead of considering the pixel intensities of the MNIST images as the features for training the SVM, the real co-efficients of the Gabor filter-bank will be used to train the SVM. Hence, for each image, we will now convolve the image using each of the 16 filters, converting 14×14 = 196 pixel image into a vector of length 196×16 = 3136. Implement this is **apply_gfilter()** that you will call to convolve your training and validaiton datasets with the 16 filters.

Since there are a lot more features now than before, standardize then use PCA to reduce the dimensionality of the dataset to be able to fit the SVM in RAM. Finally, train the SVM on these features and report the best validation accuracy.

Report:

(a) (5 points) It may not always be possible to classify a dataset cleanly into positive and negatively labeled samples, i.e., there may not exist a θ that satisfies all constraints in (1). To handle such cases, we relax the problem formulation. We create a "slack" variable that allows the constraint to be written as

$$\text{subject to} \quad y_i(\theta^\top x_i + \theta_0) \geq 1 - \xi_i;\ \xi_i \geq 0.$$

The variable ξ_i measures the degree to which we can violate the original constraint. We would like to minimize the violation of the original constraints and the slack variable-based formulation of (1) will use a different objective that does so. There can be many such objectives, write down one.

(b) (2 points) What are support samples in an SVM?

(c) (5 points) The mathematical formulation of the SVM that we saw above is for a binary classifier. The MNIST dataset clearly consists of digits from 0-9 and has 10 classes in total. How does svm.SVC handle multiple classes? Can you think of any alternative ways to use binary classifiers to perform multi-class classification?

(d) (5 points) Read the manual of svm.SVC carefully. Identify all the options that you may not have seen in your previous course on SVMs. Libraries that are used in production such as scikit-learn will have numerous knobs to improve the performance; these knobs often implement state of the art research and it is useful to know them. What does the parameter named "shrinking" in svm.SVC do? Explain what optimization algorithm is used to fit the SVM in scikit-learn. Why does fitting SVMs requires a decent amount of RAM? What do the parameters C and γ do? What are their default values?

(e) (1 point) Note down the ratio of the number of support samples to the total number of training samples for your trained classifier.

(f) (4 points) Report the validation error and confusion matrix on the validation data. Do you notice any patterns about what kind of mistakes are being made? Can you explain these mistakes intuitively?

(g) (2 points) Show all the hyperparameters values you tried by the method sklearn.model_selection.GridSearchCV and their respective accuracies. What are the best hyperparameters chosen and why do you think they make sense in producing the best SVM?

(h) (15 points) Add the example images of real and imaginary coefficients returned by calling gabor_kernel(). Add the example of an image and its real coefficient returned by calling gabor(). Plot 8 pairs of real and imaginary coefficients from the 16 filters in the filter-bank as a result of the different parameters for frequency, theta, and bandwidth to see that it gives you a good spread of different filters. Include the description on what you did, explain and analyze of your results. Report your best validation accuracy of an SVM trained on the standardized and PCA of the gabor filters.

Problem 2 (10 points: 10 Report). Prove Jensen's inequality: for any random variable X with expectation μ and a convex, finite function φ

$$\mathop{\mathrm{E}}_{X}\left[\varphi(X)\right] \geq \varphi(\mu).$$

You can assume that the random variable X takes values in a finite set. If you want to prove it in a more general setting, you can assume that the function φ is differentiable.

Problem 3 (70 points: 15 Report, 55 Code; Do this on your laptop). Here, you will develop neural network models in two ways. The first neural network model will be written and trained from scratch You will write code using only Numpy and basic Python (note, you cannot use PyTorch/TensorFlow/other deep learning library except for downloading the data) to develop this self NN model. The second neural network model will be developed with PyTorch. You will check how the resulting models compare by looking at the training and validation loss and errors where you should find the models to have similar performance.

Code:

(a) (4 points) In **get_data()**, download the MNIST dataset using the following code.

```
train = MNIST('./', download=True, train=True)
val = MNIST('./', download=True, train=False)
```

Next, we want to subsample these datasets in **subsample()** to keep only a portion of the datasets and normalize the data.

You should find the downloaded training dataset to have 60,000 images while the validation dataset has 10,000 images spread roughly equally across 10 classes. First, we want to keep only 50% of the images *from each class* for training and validation, i.e., you will end up with 30,000 training images and 5,000 validation images, evenly spread across all classes.

Next, notice that each image data is an array of integers with length 784, where each value represents a pixel of value between 0 and 255. Normalize the data to get all pixel values in the datasets into the range of 0.0 to 1.0. Make sure that the images are flattened before normalization.

Finally, plot four randomly chosen images from your subsampled dataset and check if their labels are correct. This is a good way to make sure that there is nothing wrong in your processed data.

The subsampled training and validation datasets should then be turned into MNISTDataset and consequently Dataloader objects with batch size of 64. The DataLoader should be initialized with "shuffle = True" option. See the example below,

```
train_dataloader = DataLoader(train_dataset, batch_size = 64, shuffle = True)
```

(b) (10 points) We will next implement different parts of a typical neural network from scratch to develop your self NN model.

First write a linear layer in the **class linear_t**; this includes the forward function

$$h^{(l+1)} = h^{(l)} W^\top + b^\top$$

and the corresponding backward function that takes the gradient $\overline{h^{(l+1)}}$ and outputs $\overline{W}$, $\overline{b}$ and $\overline{h^{(l)}}$. You can refer to chapter 4 of the textbook for directions on backpropagation implementation. Remember to write your function in such a way that it takes in a mini-batch of vectors $h^{(l)}$ as the input, i.e., if the feature vector $h^{(l)}$ is a-dimensional, for b images in the mini-batch, your forward function will take as input

$$h^{(l)} \in \mathbb{R}^{b \times a}$$

use

$$W \in \mathbb{R}^{c \times a}, \quad b \in \mathbb{R}^{c}$$

and output a mini-batch of feature vectors of size

$$h^{(l+1)} \in \mathbb{R}^{b \times c}.$$

Note that in this problem we have $a = 784$ because there are 28×28 pixels in MNIST images and $c = 10$ because there are 10 classes in MNIST. You should use numpy to write the forward function; do not use a for loop for computing the mini-batch-ed forward because it will be too slow for the next parts of the problem. You are advised to first write this function for $b = 1$ to understand the process and then you can extend it to $b > 1$. Make sure that you initialize the weights and biases using uniform distribution in [0,1] (see np.random.rand() for definition). Some pseudo code is given below.

```
class linear_t:
    def __init__(self):
        # initialize to appropriate sizes, fill with uniformly distributed
        # random value using np.random.rand() function
        self.w, self.b = ...

    def forward(self, h^l):
        h^{l+1} = ...
        # cache h^l in forward because we will need it to compute
        # dw in backward
        self.hl = h^l
        return h^{l+1}

    def backward(self, dh^{l+1}):
        dh^l, dw, db = ...
        self.dw, self.db = dw, db
        # notice that there is no need to cache dh^l
        return dh^l

    def zero_grad(self):
        # useful to delete the stored backprop gradients of the
        # previous mini-batch before you start a new mini-batch
        self.dw, self.db = 0*self.dw, 0*self.db
```

(c) (5 points) Implement the rectified linear unit (ReLU) layer next in the **class relu_t**. This will take the form of

$$h^{(l+1)} = \max(0, h^{(l)})$$

where the max is performed element-wise on the elements of $h^{(l)}$. Write the forward function and the corresponding backward function.

(d) (10 points) Next, we will write a combined softmax and cross-entropy loss layer in the **class softmax_cross_entropy_t**. This is a layer that first performs the operation

$$h_k^{(l+1)} = \frac{e^{h_k^{(l)}}}{\sum_{k'} e^{h_{k'}^{(l)}}}$$

where $h_k^{(l)}$ is the k^{th} element of the vector $h^{(l)}$. The input to this layer, i.e., $h^{(l)}$ are called the “logits”. The output of this layer is a scalar, it is the negative log-probability of predicting the correct class, i.e.,

$$\ell(y) = -\log\left(h_y^{(l+1)}\right).$$

where y is the true label of the image. For a mini-batch with b images, the average loss will be

$$\ell(\{y_i\}_{i=1,\dots,b}) = -\frac{1}{b}\sum_{i=1}^{b} \log\left(h_{y_i}^{(l+1)}\right).$$

You will again implement a forward function and a backward function for it yourself; remember to implement both functions to take in a mini-batch of inputs. The pseudo-code for the log-softmax layer is similar to that of the fully-connected layer.

```
class softmax_cross_entropy_t:
    def __init__(self):
        self.y, self.prob=...

    def forward(self, h^l, y):
        h^{l+1} = ...
        # compute average loss ell(y) over a mini-batch
        ell = ...
        error = ...
        return ell, error

    def backward(self):
        # as we saw in the notes, the backprop input to the
        # loss layer is 1, so this function does not take any
        # arguments
        dh^l = ...
        return dh^l

    def zero_grad(self):
        self.y, self.prob=...
```

We can also output the error of predictions in the forward function. It is computed as

$$\text{error} = \frac{1}{b}\sum_{i=1}^{b} \mathbf{1}_{\left\{y_i \neq \operatorname{argmax}_k h_k^{(l+1)}\right\}}$$

and measures the number of mistakes the network makes.

(e) (10 points) Before moving on to training, let us check whether we have implemented the forward and backward correctly. Consider the function for the linear layer. In the **class linear_t**, complete the functions **backward_check_dw()**, **backward_check_db()**, and **backward_check_dhm()**. **Use a**

batch-size $b = 1$ for this part. The forward function for the linear layer implements

$$h^{(l+1)} = h^{(l)} W^\top + b^\top$$

which is easy enough. However, we would like to check our implementation of the backward function.

```
def backward(self, dh^{l+1}):
    dh^l, self.dw, self.db = ...
    return dh^l
```

Think carefully about your implementation of the backward function. Notice that if you call the backward function with the argument $\overline{h^{l+1}} = [0, 0, \ldots, 0, 1, 0, 0 \ldots]$, i.e., there is a 1 at the k^{th} element, the function is going to calculate the quantities

$$\texttt{self.dw} = \frac{\partial h_k^{(l+1)}}{\partial W}, \quad \texttt{self.db} = \frac{\partial h_k^{(l+1)}}{\partial b}, \quad \texttt{dh}^{(l)} = \frac{\partial h_k^{(l+1)}}{\partial h^{(l)}}.$$

We now compute the estimate of the derivative using finite-differences, e.g.,

$$\frac{\partial h_k^{(l+1)}}{\partial W_{ij}} \approx \frac{\left(h^{(l)} \left(W + \epsilon\right)^\top\right)_k - \left(h^{(l)} \left(W - \epsilon\right)^\top\right)_k}{2\epsilon_{ij}}$$

where ϵ is a matrix with a Gaussian random variable as the $(ij)^{\text{th}}$ entry and zero everywhere else. In simple words, you can perturb the $(ij)^{\text{th}}$ element of weight W by ϵ_{ij}, compute the right hand-side of the finite-difference estimate above and compare it with the $(ij)^{\text{th}}$ element of your variable `self.dw`.

This idea checks the gradient with respect to only one element of W, namely W_{ij}. Do this for about 10 randomly chosen elements of W and a few (5 should be enough) different entries k of $h_k^{(l+1)}$ and check if the answer matches `self.dw` that you have implemented in the backward function. Repeat this process for the other two gradients.

Do not move on to the next part until you are convinced your implementation of forward/backward is correct for all the three layers. It is essential that the gradient is implemented correctly, your training will not work if the gradient is wrong.

(f) (4 points) You will now train your neural network in **train_self_NN()**. The pseudo-code looks as follows:

```
# load dataset
...

# initialize all the layers
l1, l2, l3 = linear_t(), relu_t(), softmax_cross_entropy_t()
net = [l1, l2, l3]

# train for at least 10,000 iterations
for t in range(10000):
    # 1. sample a mini-batch of size = 64
    # each image in the mini-batch is chosen uniformly randomly from the
    # training dataset
    x, y = ...

    # 2. zero gradient buffer
```

```
    for l in net:
        l.zero_grad()

    # 3. forward pass
    h1 = l1.forward(x)
    h2 = l2.forward(h1)
    ell, error = l3.forward(h2, y)

    # 4. backward pass
    dh2 = l3.backward()
    dh1 = l2.backward(dh2)
    dx = l1.backward(dh1)

    # 5. gather backprop gradients
    dw, db = l1.dw, l1.db

    # 6. print some quantities for logging
    # and debugging
    print(t, ell, error)
    print(t, np.linalg.norm(dw/l1.w), np.linalg.norm(db/l1.b))

    # 7. one step of SGD
    l1.w = l1.w - lr*dw
    l1.b = l1.b - lr*db
```

The hyperparameters to tune here are the number of iterations or weight updates and the learning rate. You should get better than/around 15% training error after 10,000-50,000 weight updates. You can pick the learning rate to be $\texttt{lr} = 0.1$. The target training error is 10%.

Save the training error and loss values for each weight update in lists to be saved as pickle files after the training loop. Make sure you generate the files **self_NN_training_error.pkl** and **self_NN_training_loss.pkl** and upload them to the autograder.

(g) (5 points) Now, we have implemented the training loop. Next, we want update the training loop from (f) to additionally calculate the loss and error on the validation dataset after every 1,000 weight updates i.e. if t % 1000 == 0 for iteration t, get validation loss and error values. Note: Calculate validation for $t \in \{0, 1000, \ldots, N\}$. Make sure the final iteration does not cause an index mismatch. Implement the helper function **validate_self_nn(l1, l2, l3, val_dataloader)** to be called for computing the validation loss and error values. Plot the validation loss and validation error values as a function of the number of weight updates.

Save the trained model in **linear.h5** to be uploaded to the autograder. Additionally, save the validation error and loss values in lists to be saved as pickle files after the training loop. Make sure you generate the files **self_NN_validation_error.pkl** and **self_NN_validation_loss.pkl** and upload them to the autograder.

If everything works as expected, congratulations! You have implemented your own little library for training neural networks, completely from scratch!

(h) (7 points) Repeat the entire process in parts (b)-(g) using the pre-built functions inside PyTorch. You will take help of the code provided in the recitation sessions for this purpose. Train the network for at least 10,000 weight updates this time in **train_pytorch_nn()** and the function to calculate validation loss and error in **validate_pytorch_nn(nn, criterion, val_dataloader)**.

Similarly, plot the training loss and training error for every weight update, as well as validation loss and validation error for every 1,000 weight updates.

Save the training loss, training error, validation loss, and validation error values in lists and save them as pickle files. Make sure you generate the files **pytorch_NN_training_error.pkl**, **pytorch_NN_training_loss.pkl**, **pytorch_NN_validation_error.pkl**, and **pytorch_NN_validation_loss.pkl** and upload them to the autograder.

Report:

(a) (3 point) Add the plots of four randomly chosen MNIST images from your dataset in subsample(), along with their corresponding labels. Are they correct?

(b) (3 points) Add the plots of training loss and training error as a function of the number of weight updates of the neural network you trained from scratch. What do you notice about the trend?

(c) (3 points) Plot the self NN validation loss and validation error as a function of the number of weight updates for every 1,000 weight updates. How do they compare to the training loss and error plots?

(d) (6 points) Plot the PyTorch NN training loss, training error as a function of the number of weight updates, as well as validation loss and the validation error for every 1,000 weight updates. How do they compare to performance of the self NN model you developed from scratch? Explain what you notice between the two models and their results.