**UNIVERSITY OF PENNSYLVANIA**

**ESE 546: PRINCIPLES OF DEEP LEARNING**

**HOMEWORK 1**

---

**Changelog**

- **No changes yet**

---

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.
- You will see an entry of the form "HW 1 PDF" where you will upload the PDF of your solutions. You will also see entries like "HW 1 Problem 1 Code" and "HW 1 Problem 3 Code" where you will upload your solution for the respective problems.
- **For each programming problem/sub-problem, you should create a fresh .py file**. This file should contain **all** the code to reproduce the results of the problem/sub-problem, e.g., it should save the plot that is required (correctly with all the axes, title and legend) as a PDF in the same directory. You will upload the .py file as your solution for "HW 1 Problem 3 Code" or "HW 1 Problem 3 Code". Name your file as pennkey_hw1_problem3.py, e.g., I will name my code as pratikac_hw1_problem3.py. 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 (and I suggest that you don't...), you can export the notebook to a .py file.
- **In addition to submitting the code, you should append the entire Python code for the particular problem to the solution in the PDF. If you are using Latex, you can do something like the screenshot below. The instructors will execute the code to check it. Your code should run without any errors and should create all output/plots required in the problem.**

```
\includepackage{pythonhighlight}

\begin{python}

a = np.array(10)

...

\end{python}
```

**Credit** The points for the problems add up to 110. You only need to solve for 100 points to get full credit, i.e., your final score will be $\min(\text{your total points}, 100)$.

---

**Problem 1 (40 points, Code up on laptop, use Google Colab is it looks like you need more RAM).** In this problem, we will fit the MNIST dataset using a support vector machine (SVM) using the "scikit-learn" library. You can install it using

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

An SVM solves an optimization problem for maximizing the margin between two classes. Support 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

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

here $\theta_0$ is the offset parameter and $\theta$ 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.

(a) (3 point) It may not always be possible to classify a dataset cleanly into positive and negatively labeled samples, i.e., there may not exist a $\theta$ that satisfies all constraints in Eq. (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 $\xi_i$ 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 Eq. (1) will use a different objective that does so. There can be many such objectives, write down one.

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

(c) You can download the dataset using

```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

ds = fetch_openml('mnist_784', as_frame=False)

x, x_test, y, y_test = train_test_split(ds.data, ds.target,
                    test_size=0.2, random_state=42)
```

Check whether you have downloaded the data correctly; the images in x_train and x_val are in the form of a vector of length 784, this is really the flattened matrix $28 \times 28$. You can check it by running the following code.

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

**Fitting SVMs requires a decent amount of RAM (think of why, and write your answer in part (e)).** We will therefore down-sample the original $28 \times 28$ images to $14 \times 14$ using the following code.

```
# code for down-sampling
import cv2
b = cv2.resize(b, (14,14))
plt.imshow(b)
```

In this problem you will create a dataset of 1000 samples for each digit (10,000 samples in total). You can sub-sample `x` to create this dataset (we are doing this step simply to reduce the amount of time it takes to fit the SVM). From within these 10,000 samples, we will construct our actual training dataset (80%) and validation dataset (20%) (you can sample randomly).

(d) (10 points) Create the SVM classifier in scikit-learn using

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

What do the parameters $C$ and $\gamma$ do? What are their default values? Fit the SVM classifier to the data and predict the labels of the validation dataset using the trained classifier. Report the validation error. Note down the ratio of the number of support samples to the total number of training samples for your trained classifier. Run the classifier on `x_test`, `y_test` and report the classification error. Report the 10-class confusion matrix on the test data. Do you notice any patterns about what kind of mistakes are being made? Can you explain these mistakes intuitively?

(e) (2 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.

(f) (3 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?

(g) (5 points) Use the sklearn.model_selection.GridSearchCV function to pick a better value than the default one for the hyper-parameter $C$. Try at least 5 different hyper-parameters. Show all the hyper-parameters tried by the method and their accuracies. How will you pick the best value of the hyper-parameter using your validation set?

(h) **The following part is computationally intensive. In this case, you can use a training dataset of 100 samples/class and a validation set of 100 images/class. Sample `x`, `y` randomly to create this dataset. No need for a test set.**

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,
- $\theta$ the rotation angle of the Gaussian,
- $\sigma_x, \sigma_y$: 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.

```
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[0].reshape((14,14))
coeff_real, _ = gabor(image, frequency=freq, theta=theta,
                      bandwidth=bandwidth)
plt.figure(1); plt.clf(); plt.imshow(coeff_real)
```

(j) (15 points) Run the above code a few times with different parameters for $F, \theta$ and bandwidth to see how the filter changes in shape and size and the corresponding output after convolution. We will create a filter bank that consists of multiple Gabor filters of fixed parameters. Instead of considering the pixel intensities of the MNIST images as the features for training the SVM, the co-efficients of the Gabor filter-bank will be used to train the SVM. You can pick

```
theta = np.arange(0,np.pi,np.pi/4)
frequency = np.arange(0.05,0.5,0.15)
bandwidth = np.arange(0.3,1,0.3)
```

This gives a total of 36 filters in the filter-bank. We therefore have converted a $14\times14$ = 196 pixel image into a vector of length $196\times36$ = 7056. Plot the filter-bank 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. Train the SVM on these features and report the training and validation accuracy.

Increase the number of filters next. You might have to use PCA to reduce the dimensionality of the dataset to be able to fit the SVM in RAM; use scikit-learn to do so.

**Problem 2 (10 points).** Prove Jensen's inequality: for any random variable $X$ with expectation $\mu$ and a convex, finite function $\varphi$

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

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 $\varphi$ is differentiable.

**Problem 3 (60 points, Do this on your laptop).** You will write code to train a neural network completely from scratch using only Numpy and basic Python (note, you cannot use PyTorch/TensorFlow/other deep learning library except for downloading the data).

(a) (0 points) Download the MNIST dataset using the following code.

```
import torchvision as thv
train = thv.datasets.MNIST('./', download=True, train=True)
val = thv.datasets.MNIST('./', download=True, train=False)
print(train.data.shape, len(train.targets))
```

The training dataset has 60,000 images while the validation dataset has 10,000 images spread roughly equally across 10 classes. Take 50% of the images *from each class* for training and validation, i.e., about 30,000 training images and 5,000 validation images, almost evenly spread across all classes with a few minor differences. We will use this smaller dataset in this problem. **Plot the images of a few randomly chosen images from your dataset and if their labels are correct.** This is a good way to make sure that there is nothing wrong in your data.

(b) (5 points) Implement an embedding layer that converts 2D images into a vector. MNIST images are of size $x = h^{(0)} \in \mathbb{R}^{28\times28}$. We will use an embedding layer (like it is done in a vision transformer) to convert this into a vector $h^{(1)} \in \mathbb{R}^{7\times7\times8}$, i.e., down from 784 dimensions to 392 dimensions, as follows. Suppose we have a weight matrix $W \in \mathbb{R}^{4\times4\times8}$ and a bias $b \in \mathbb{R}^8$. The operation that we are interested in is

$$\mathbb{R}^8 \ni h^{(1)}_{i,j} = \sum_{|i'-4i|\leq2, |j'-4j|\leq2} \underbrace{W_{i',j',\cdot}}_{\in\mathbb{R}^8} \underbrace{x_{i',j'}}_{\in\mathbb{R}} + \underbrace{b}_{\in\mathbb{R}^8}.$$

This equation looks a bit complicated but it does something simple. It takes a $4\times$ 4 patch of an MNIST image (there are $7 \times 7 = 49$ such patches), multiplies each of the 16 entries in this patch by the quantity $W_{i',j'} \in \mathbb{R}^8$, adds the bias $b \in \mathbb{R}^8$ and writes down the answer as $h^{(1)}_{i,j}$. You will flatten the vector $h^{(1)} \in \mathbb{R}^{392}$ (note that $392 = 7 \times 7 \times 8$). The above operation was described for a single input image, you will write your forward pass to take a mini-batch of images as input and return the embedding features for the entire mini-batch. And similarly, for the backward pass. 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 $b > 1$.

```
class embedding_t:
    def __init__(self):
        # initialize to appropriate sizes, fill with Gaussian entires
        # normalize to make the Frobenius norm of w, b equal to 1
        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) Next we implement a standard linear layer. This includes the forward function

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

and the corresponding backward function that takes the gradient $\overline{h^{(l+1)}}$ and outputs $\overline{W}, \overline{b}$ and $\overline{h^{(l)}}$. 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 $a = 392$ because there are $7 \times 7 \times 8$ feature after the embedding layer and $c = 10$ because there are 10 classes in MNIST. Some pseudo code is given below.

```
class linear_t:
    def __init__(self):
        # initialize to appropriate sizes, fill with Gaussian entires
        # normalize to make the Frobenius norm of w, b equal to 1
        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
```

(d) (5 points) Implement the rectified linear unit (ReLU) layer next. 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.

(e) (5 points) Next we will write a combined softmax and cross-entropy loss layer. 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 $k^{\text{th}}$ 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 $\mathscr{b}$ images, the average loss will be

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

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. It does not have any parameters to initialize and therefore does not need the zero_grad method.

```
class softmax_cross_entropy_t:
    def __init__(self):
        # no parameters, nothing to initialize

    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
```

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

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

and measures the number of mistakes the network makes.

(f) (10 points) Before moving on to training, let us check whether we have implemented the forward and backward correctly for all the four layers. Consider the function for the linear layer. **Use a batch-size $\mathcal{b} = 1$ for this part.** The forward function for the linear layer implements

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

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 $k^{\text{th}}$ 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 $\epsilon$ 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 $\epsilon_{ij}$, 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.**

(g) (10 points) You will now train your neural network. The pseudo-code looks as follows:

```
# load dataset
...

# initialize all the layers
l1, l2, l3, l4 = embedding_t(), linear_t(), relu_t(), softmax_cross_entropy_t()
net = [l1, l2, l3, l4]

# train for at least 1000 iterations
for t in range(1000):
    # 1. sample a mini-batch of size = 32
    # 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)
    h3 = l4.forward(h2)
    ell, error = l4.forward(h3, y)

    # 4. backward pass
    dh3 = l4.backward()
    dh2 = l3.backward(dh3)
    dh1 = l2.backward(dh2)
    dx = l1.backward(dh1)

    # 5. gather backprop gradients
    dw1, db1 = l1.dw, l1.db
    dw2, db2 = l2.dw, l2.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*dw1
    l1.b = l1.b - lr*db1
    l2.w = l2.w - lr*dw2
    l2.b = l2.b - lr*db2
```

You can pick the learning rate to be $\texttt{lr} = 0.1$. **Plot the training loss and training error as a function of the number of weight updates**. Make sure that the training loss decreases with the number of updates. You should try to get better than/around 15% error on the training dataset after 10,000-50,000 updates.

(h) (5 points) We have implemented the training loop. Write the corresponding code for computing the validation loss and error.

```
def validate(w, b):
    # 1. iterate over mini-batches from the validation dataset
    # note that this should not be done randomly, we want to check
    # every image only once

    loss, tot_error = 0, 0
    for i in range(0, 5000, 32):
        x, y = val.data[i:i+32], val.targets[i:i+32]

        # 2. compute forward pass and error
```

**Plot the validation loss and validation error as a function of the number of weight updates, every 1000 weight updates**.

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

(i) (15 points) Repeat the entire process in parts (b)-(g) using the pre-built functions inside PyTorch. For implementing the embedding layer you will need to use either nn.Embedding or nn.Conv2d with a specific value of stride. 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. Plot the training loss, training error, validation loss and the validation error as a function of the number of weight updates.