Problem 1 (60 points). Use Google Colab for this problem. 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

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

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.

(a) (5 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 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 point) Define what are support samples in an SVM.

(c) (3 points) 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')
x, y = ds.data, ds.target

x_train, x_val, y_train, y_val = train_test_split(x, y,
                        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 28×28. You can check it by plotting

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

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

Construct training (80%) and validation (20%) datasets from the arrays x, y by sampling the images and labels randomly. You should make sure that each class has an equal number of samples. Why did we not construct a test dataset here?

(d) (15 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 γ 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. Note that the input data for an SVM is a vector of 784, not an image of size 28×28. Provide the validation accuracy and the 10-class confusion matrix. Note down the ratio of the number of support samples to the total number of training samples for your trained classifier. **If training takes too long or runs out of memory, you can down-sample the original 28×28 images to 14×14 (remember to reshape it to 196 before training the SVM), and/or reduce the size of the training set.**

(e) (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. For instance, what does the parameter named "shrinking" in svm.SVC do? Investigate and explain what optimization algorithm is used to fit the SVM in scikit-learn.

(f) (5 points) The mathematical formulation of the SVM above is for a binary classifier. The MNIST dataset 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.

(h) **The following two parts are computationally intensive. Down-sample all images to 14×14 and create a training dataset using only 500 images from the full MNIST dataset. Make sure that the training dataset is balanced, i.e., pick 50 images per digit. Similarly, pick an additional 500 images (50 images/digit) to form the validation 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. It has been found that the mammalian visual cortex consists of cells well-modeled by Gabor functions (named after Dennis Gabor, a Hungarian physicist who invented holography). 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

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

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, this is different from a standard convolutional filter. Convolving the original image $I(x, y)$ with the filter $g(x, y)$ will result in two sets of co-efficients, 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.

```python
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)
```

(j) (20 points) Run the above code a few times with different parameters for F, θ 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

```python
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 $14 \times 14 = 196$ pixel image into a vector of length $196 \times 36 = 7056$. 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.

deep learning library except for downloading the data). **Work on this problem on your personal computer before moving to Colab, this will help during debugging.**

(a) (5 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.**

(b) (10 points) We will next implement different parts of a typical neural network. First write a 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 $a = 784$ 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 $b > 1$. 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
```

(c) (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.

(d) (10 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^{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

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

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 for all the three layers. Consider the function for the linear layer. **Use a batch-size $b = 1$ for this part.** The forward function for the linear layer implements

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

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) (10 points) You will now train your neural network. 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 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)
    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
```

You can pick the learning rate to be `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.

(g) (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!

(h) (15 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. Plot the training loss, training error, validation loss and the validation error as a function of the number of weight updates.