Coding (5 points)
Your task is to implement neural networks and train them to perform classification on different types of data. You will write the following code:

A function for training a model on a training dataset for **one epoch** (in src/run_model.py)
A function for evaluating a trained model on a validation/testing dataset (in src/run_model.py)
A function which trains (over multiple epoch), validates, or tests a model as specified (in src/run_model.py)
Two fully connected and two convolutional neural networks (in src/models.py)

Here is a cheat sheet of commonly used methods:
https://pytorch.org/tutorials/beginner/ptcheat.html

Here is a comparison of PyTorch and Numpy methods:
https://github.com/wkentaro/pytorch-for-numpy-users

Training on MNIST (1 point)

Let us train neural networks to classify handwritten digits from the MNIST dataset and analyze the accuracy and training time of these neural networks.

Build the model architecture: Create a neural network with **two fully connected** (aka, dense) hidden layers of **size 128, and 64**, respectively. Your network should have **a total of four layers**: an input layer that takes in examples, two hidden layers, and an output layer that outputs a **predicted class** (**10 possible classes**, one for each digit class in MNIST). Your hidden layers should **have a ReLU activation function**. Your last (output) layer should be a linear layer with one node per class (in this case 10), and the predicted label is the node that has the max value. *Hint: In PyTorch the **fully connected layers are called torch.nn.Linear().**

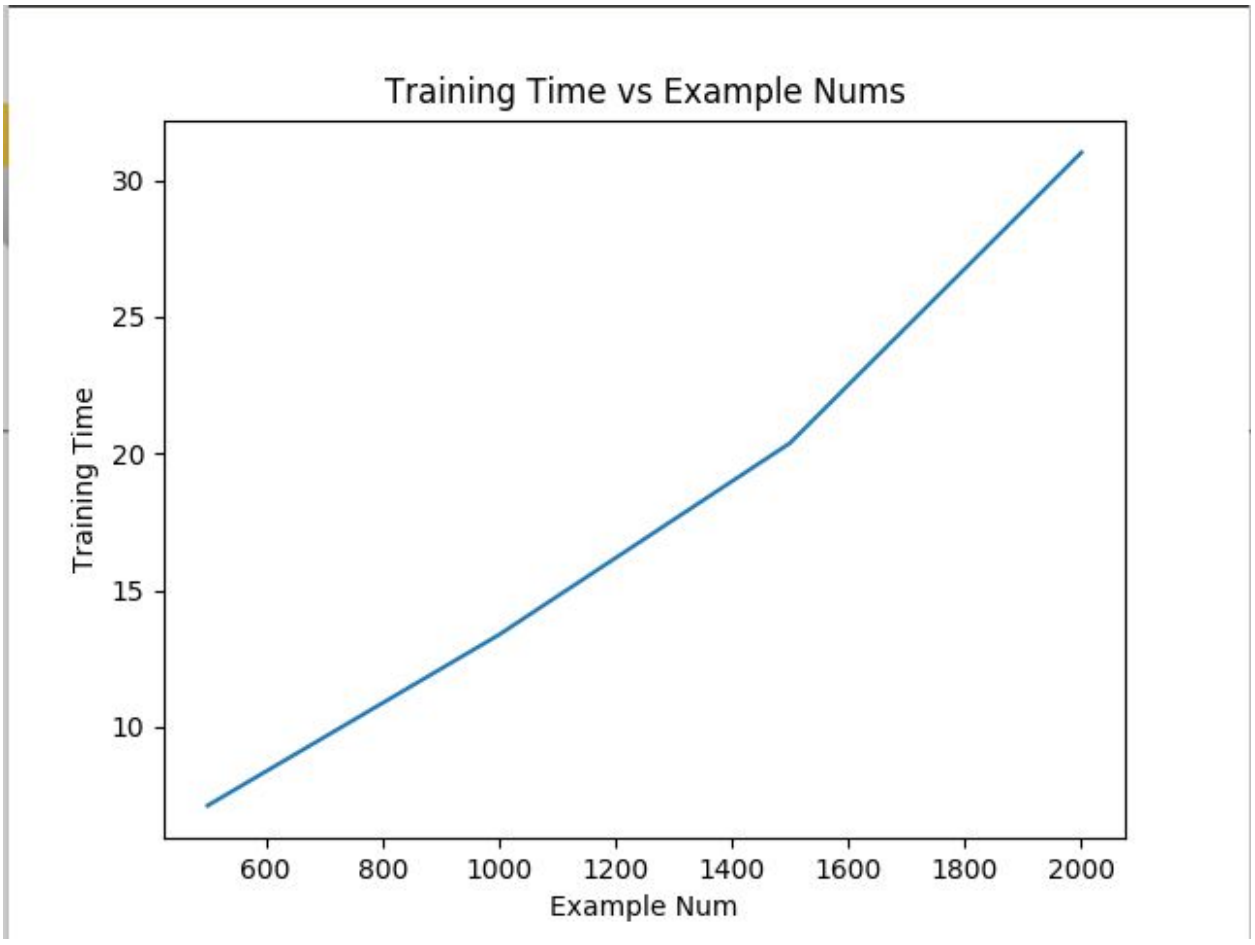
Use these training parameters: When you train a model, train for **100 epochs** with **batch size of 10** and use cross entropy loss. In PyTorch's CrossEntropyLoss class, **the softmax operation is built in**, therefore you do not need to add a softmax function to the output layer of the network. Use the SGD optimizer with a learning rate of **0.01**.

Making training sets: Create training datasets of each of these sizes {500, 1000, 1500, 2000} from MNIST. Note that you should be selecting examples in such a way that you minimize bias, i.e., make sure all ten digits are equally represented in each of your training sets. To do this, you can use load_mnist_data function in load_data.py where you can adjust the number of examples per digit and the amount of training / testing data.

*Hint: To read your MNIST dataset for training, you need to use a PyTorch DataLoader. To do so, you should use a custom PyTorch Dataset class. We included the class definition for you in the HW (MyDataset in data/my_dataset.py) You can see more details about using custom dataset in this blog or github repo). When creating a DataLoader set the shuffle property to True.

Train one model per training set: Train a new model for each MNIST training set you created and test it on a subset of the MNIST testing set (1000 samples). Use the same architecture for every model. For each model you train, record the loss function value every epoch. Record the time required to train for 100 epochs. From python's built in time module, use time.time().

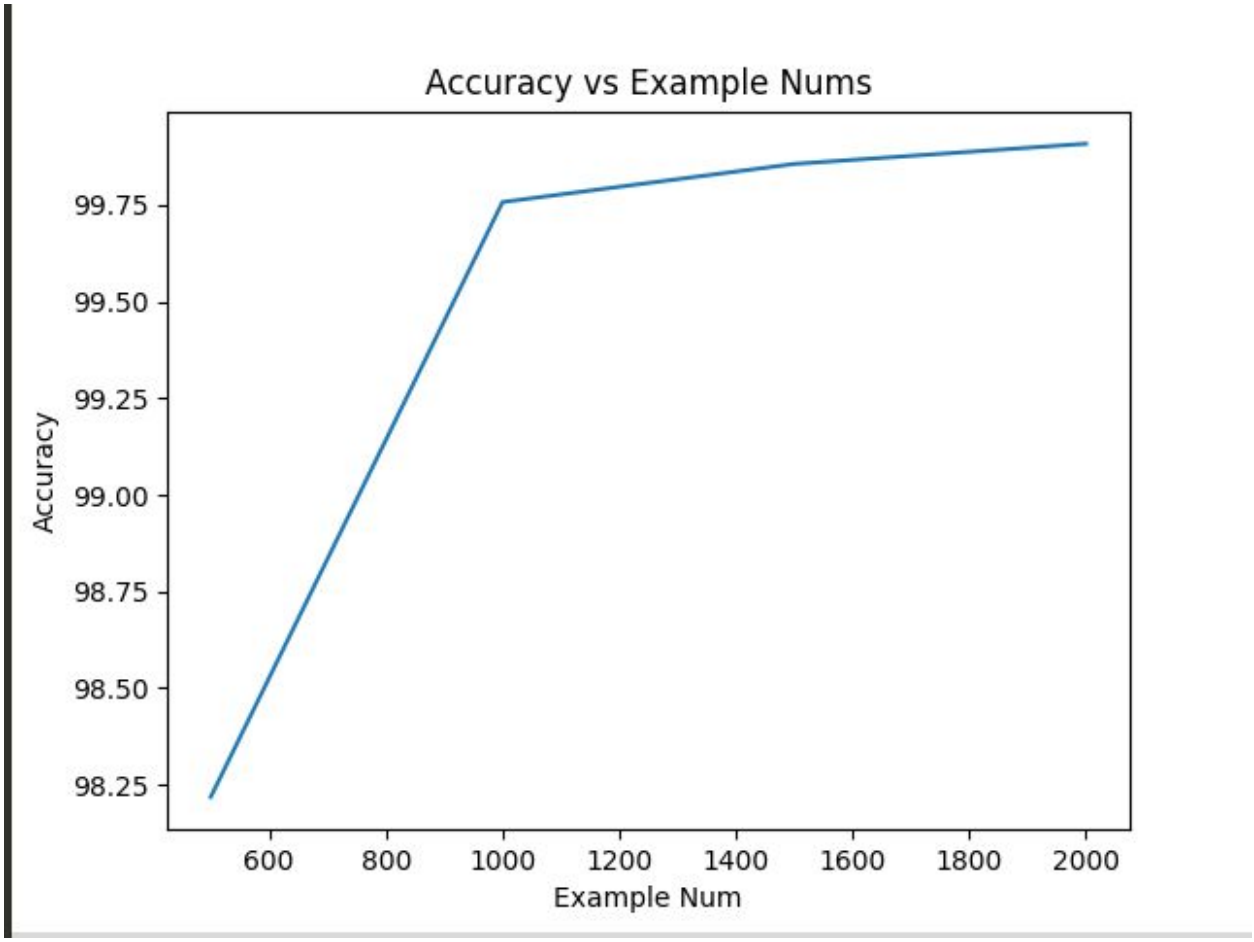
(0.25 points) Given the data from your 4 trained models, create a graph that shows the amount of training time along the y-axis and number of training examples along the x-axis.



(0.25 points) What happens to your training time as the number of training examples increases? Roughly how many hours would you expect it to take to train on the full MNIST training set using the same architecture on the same hardware you used to create the graph in question 1?

Answer: as the number increases, the training time increases almost linearly. Therefore,

Given that the size of MNIST is 60000 for training takes 30s * 60000/2000 = 900s = 1/4 hour
(0.25 points) Create a graph that shows **classification accuracy** on your testing set on the y-axis and number of training examples on the x-axis.



(0.25 points) What happens to the accuracy as the number of training examples increases?

Answer: Accuracy will converge to 100% as training sample size increases.

Exploring DogSet (.5 points)

DogSet is a subset from a popular machine learning dataset called ImageNet (more info here and here) which is used for image classification. The DogSet dataset is available here. (Note: you need to be signed into your @u.northwestern.edu google account to view this link). As its name implies, the entire dataset is comprised of images of dogs and labels indicating what dog breed is in the image. The metadata, which correlates any particular image with its label and partition, is provided in a file called dogs.csv. We have provided a general data loader for you (in data/dogs.py), but you may need to adopt it to your needs when using PyTorch. Note: You

need to use the dataset class we provided in MNIST questions to be able to use a PyTorch DataLoader

Validation sets: Thus far, you have only used "train" and "test" sets. But it is common to use a third partition called a "validation" set. The validation set is used during training to determine how well a model generalizes to unseen data. The model does not train on examples in the validation set, but periodically predicts values in the validation set while training on the training set. Diminishing performance on the validation set is used as an early stopping criterion for the training stage. Only after training has stopped is the testing set used. Here's what this looks like in the context of neural networks: for each epoch a model trains on every example in the training partition, when the epoch is finished the model makes predictions for all of the examples in the validation set and a loss is computed. If the difference between the calculated loss for this iteration and the previous is below some epsilon for N number of epochs in a row, then training stops and we move onto the testing phase.

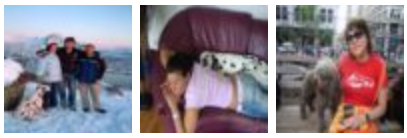
(0.25 points) In Dogset, how many are in the train partition, the valid partition and the test partition? What is the color palette of the images (greyscale, black & white, RBG)? How many dog breeds are there?

Answer:

train: (7665, 64, 64, 3)
test: (555, 64, 64, 3)
valid: (2000, 64, 64, 3)

There are 10 breeds in total. Color Palette is RGB.

(0.25 points) Select one type of breed. Look through variants of images of this dog breed. Show 3 different images of the same breed that you think are particularly challenging for a classifier to get correct. Explain why you think these three images might be challenging for a classifier.



Answer: I think these pictures are difficult to classify, because there are bipeds in them, and the bipeds are bad because the naive dog classifier will be tricked!

Training a model on DogSet (2 points)

Build the model architecture: Create a neural network with **two fully connected** (aka, dense) hidden layers of size **128, and 64**, respectively. Note that you should be **flattening your NxNxC** image to 1D for your input layer (where N is the height/width, and C is the number of color channels). Your network should have a total of four layers: an input layer that takes in examples, two hidden layers, and an output layer that outputs a predicted class (**one node for each dog class in DogSet**). Your hidden layers should have a ReLU activation function. Your last (output) layer should be a linear layer with one node per class, and the predicted label is the node that has the max value.

Use these training parameters: Use **a batch size of 10** and the cross entropy loss. Use the SGD optimizer with a learning rate of **1e-5**.

When to stop training: Stop training after **100 epochs** or when the validation loss decreases by less than **1e-4**, whichever happens first.

Training, Testing, Validation sets: You should use training examples from train partition of DogSet. Validation should come from the valid partition and testing examples should come from the test partition.

(0.5 points) How many connections (weights) does this network have?
Answer: 64*64*128+ 128*64 + 64*10 = 533120

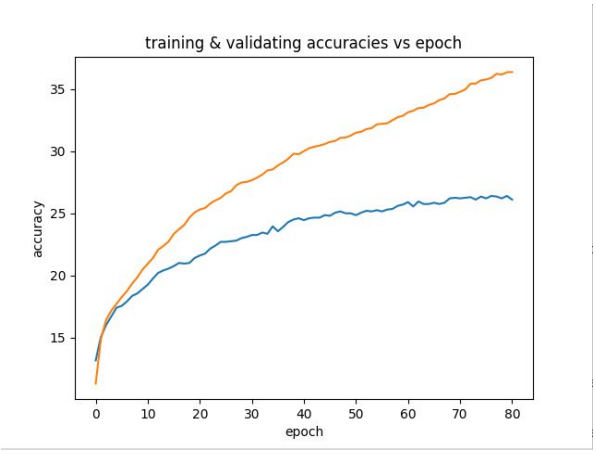
(1.0 point) Train a model on DogSet. After every epoch, record four things: **the loss** of your model on the training set, **the loss** of your model on the validation set, and **the accuracy** of your model on the training and validation sets.

Report the number of epochs your model trained, before terminating.
Answer: Total number: 81

Make a graph that has **both training and validation loss** on the y-axis and epoch on the x-axis.

Note: Sorry I forgot to put the legends, the blue line is the validation set loss, the orange is the training set loss.

Make a graph that has both **training and validation accuracy** on the y-axis and epoch on the x-axis.



Note: Sorry I forgot to put the legends, the blue line is the validation set accuracy, the orange is the training set accuracy.

Report the accuracy of your model on the testing set.
Answer: 24.14%

(0.5 points) Describe the interaction between training loss, validation loss and validation accuracy. When do you think your network stopped learning something meaningful to the problem? Why do you think that? Back up your answer by referring to your graphs.

Answer: as training loss goes down, the validation loss should go down, and as a result, validation accuracy goes down. The network will stop working ??
[Question]

Convolutional layers (2 points)

Convolutional layers are layers that sweep over and subsample their input in order to represent complex structures in the input layers. For more information about how they work, see this blog post. Don't forget to read the PyTorch documentation about Convolutional Layers (linked above).

(0.5 points) Convolutional layers produce outputs that are of different size than their input by representing more than one input pixel with each node. If a 2D convolutional layer has 3 channels, batch size 16, input size (32, 32), padding (4, 8), dilation (1, 1), kernel size (8, 4), and stride (2, 2), what is the output size of the layer?(40-7, 48-3)/2. 15,

[Quesion]
Answer: output size of the convolutional layer is (containing max pooling): 3 * 17 * 23
Assume the filter size is (3, 32, 32)

(0.5 point) Combining convolutional layers with fully connected layers can **provide a boon in scenarios** involving learning from images. Using a similar architecture to the one used in question 8, replace each of your first two hidden layers with a convolutional layer, and add a fully connected layer to output predictions as before. The number of filters (out_channels) should be 16 for the first convolutional layer and 32 for the second convolutional layer. When you call the PyTorch convolutional layer function, leave all of the arguments to their default settings except for kernel size and stride. Determine reasonable values of kernel size and stride for each layer and report what you chose. Tell us how many connections (weights) this network has.

Answer: For convolutional layer, kernel size: 5x5, stride size: 1x1; For max pooling layer: kernel size: 2x2, stride 2x2
So in total: 16*(64)*(64) + 32*(64-4)/2*(64-4)/2 + 5408 = 99744 weights.

(1 point) Train your convolutional model on DogSet. After every epoch, record four things: the loss of your model on the training set, the loss of your model on the validation set, and the accuracy of your model on both training and validation sets.

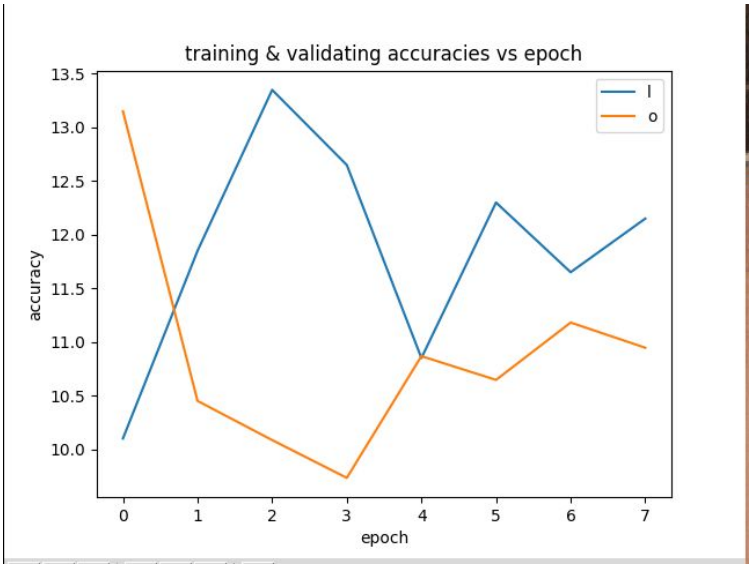
Report the number of epochs your model trained, before terminating.
Total Epoch Number: 8

Make a graph that has both training and validation loss on the y-axis and epoch on the x-axis.

Make a graph that has both training and validation ac

Blue line is the Validation loss, Orange line is the Training Loss

Accuracy on the y-axis and epoch on the x-axis.



Blue line is the Validation Accuracy, Orange line is the Training Accuracy

Report the accuracy of your model on the testing set.

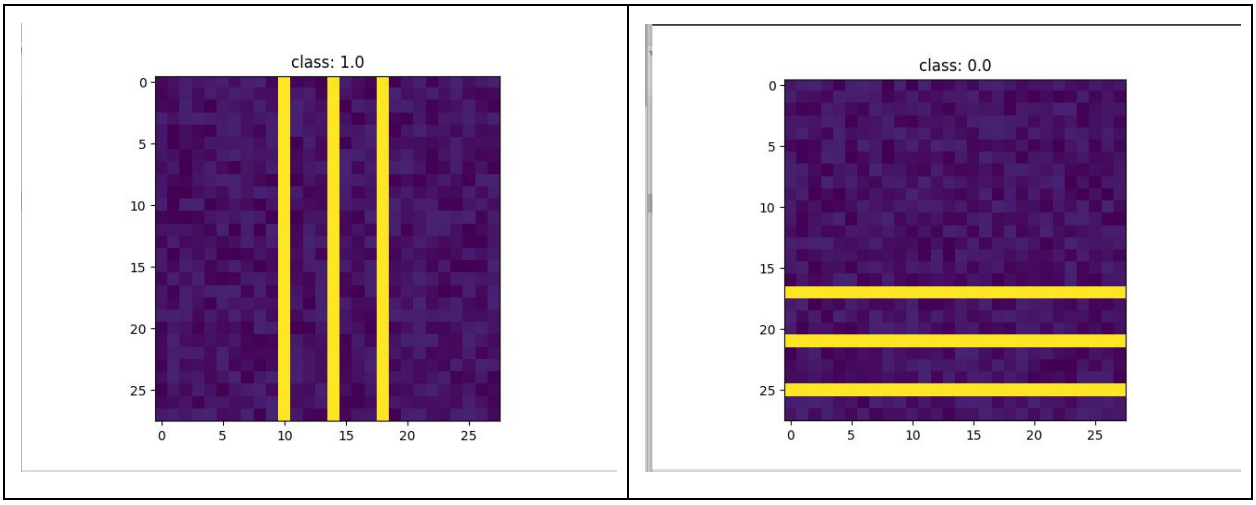
Accuracy: 13.873873873873874
(I am not sure if there is anything wrong, but it took me almost 10 min to train...)

Digging more deeply into convolutional networks (2 points)

The most important property of convolutional networks is their capability in **capturing shift invariant patterns**. You will investigate this property by training a convolutional network to classify simple synthesized images and visualizing the learned kernels.

Exploring the synthesized dataset: Download the synth_data file, unzip it, and put it in /data directory. synth_data contains 10000 images of simple patterns, divided into 2 classes (5000 images per class). Use the load_synth_data function in data/load_data.py to load the training features (images) and labels.

(1 point) Go through a few images and plot two examples (1 from each class). What is the common feature among the samples included in each class? What is different from one sample to the next in each class? What information must a classifier rely on to be able to tell these classes apart?



Answer: The common feature in each class is the orientation of the edges. Within each class, the difference between each image is the position of the edges. The classifier should be able to use orientation, instead of the position of edges, to tell classes apart.

Build the classifier: Create a convolutional neural network including three convolutional layers and a linear output layer. The numbers and sizes of filters should be as follows:

First layer: 2 filters of size (5,5)

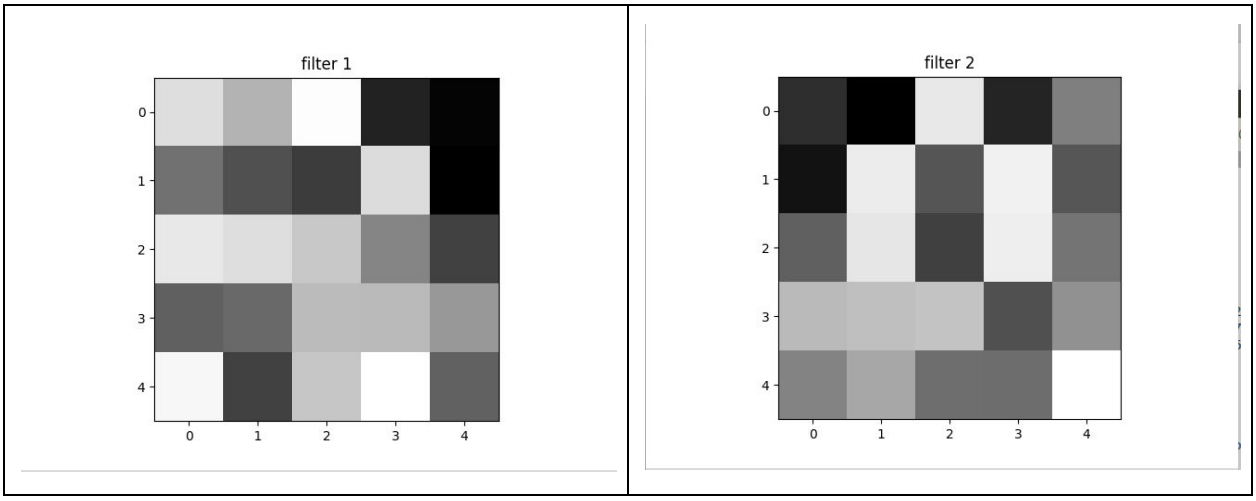
Second layer: 4 filters of size (3,3)

Third layer: 8 filters of size (3,3)

Use strides of size (1,1) and ReLU activation functions in all convolutional layers. Each convolutional layer should be followed by max-pooling with a kernel size of 2. Use an output linear layer with two nodes, one for each class (note that for binary classification you can use a single node with a sigmoid activation function and binary cross entropy loss, but using softmax and cross entropy keeps the code simpler in this homework).

Training parameters: Use a cross entropy loss and the SGD optimizer. Set the batch size to 50 and learning rate to 1e-4. Train the network for 50 epochs.

(1 point) Once the network is trained extract and plot the weights of the two kernels in the first layer. Do these kernels present any particular patterns? If so, what are those patterns and how are they related to the classification task at hand and the classifier performance? Note that since the model is randomly initialized (by default in PyTorch), the shape of kernels might be different across different training sessions. Repeat the experiment a few times and give a brief description of your observations. [Question]



As can be seen, the kernels roughly represent the horizontal and vertical orientations of the edges. These filters will keep edges with the corresponding orientations in images. However there are other features such as corners will be maintained, which might degrade the performance of the classifier.

Thinking about deep models (1.5 points)

(0.25 points) For any binary function of binary inputs, is it possible to construct some deep network built using only perceptron activation functions that can calculate this function correctly? If so, how would you do it? If not, why not?

Answer: No, it is not possible, Xor needs Non-linear decision boundary and combinations of perceptrons can only learn binary decision boundaries.

(0.25 points) Is it possible to learn any arbitrary binary function from data using a network build only using linear activation functions? If so, how would you do it? If not, why not?

Answer: Yes, by using a Multi-Layer Network, the Xor Problem can be sovled, and by adjusting the structure of the network, any binary function can be learned.

(1 point) An adversarial example is an example that is designed to cause your machine learning model to fail. Gradient descent ML methods (like deep networks) update their weights by descending the gradient on the loss function L(X,Y,W) with respect to W. Here, X is a training example, Y is the true label and W are the weights. Explain how you could create an adversarial example by using the gradient with respect to X instead of W.

Answer: Here, the goal is to find an X vector such that the neuro-network will misclassify. For example, we want to find a picture of digit 9 to be classified as digit 5.

Then, we can define the loss, such as using euclidean distance:

$$C = \frac{1}{2}\|y_{goal} - \hat{y}(\vec{x})\|_2^2$$

Our goal is to minimize this euclidean distance.

And the way we do this is to use Gradient Descent, which is the same as in training a neuronetwork.

Steps:

1. **Find the derivative of the cost function with respect to x**
2. **Forward propagate to calculate loss**
3. **Find the gradient at x using the derive of the cost function**
4. **Backpropagate gradient*learning_rate to x.**