1 Deep Learning Principles [35 Points]

Relevant materials: lectures on deep learning

For problems A and B, we'll be utilizing the Tensorflow Playground to visualize/fit a neural network.

Problem A [5 points]: Backpropagation and Weight Initialization Part 1

Fit the neural network at this link for about 250 iterations, and then do the same for the neural network at this link. Both networks have the same architecture and use ReLU activations. The only difference between the two is how the layer weights were initialized – you can examine the layer weights by hovering over the edges between neurons.

Give a mathematical justification, based on what you know about the backpropagation algorithm and the ReLU function, for the difference in the performance of the two networks.

Solution A.: *The first network's weights are initialized roughly according to a Gaussian distribution between [-1, 1] while the second network's weights are initialized all to 0. Because this second network has weights initialized to 0, the neural network does not learn with each epoch because during backpropagation, the gradient for the Loss function with respect to the weights will always be 0. Therefore, the weights are never updated with each batch and epoch. A neural net is optimized with weight changes and thus, no weight changes means that the neural net does not change with each iteration. Therefore, it will never converge which is why we see poor test performance 0.508 as opposed to the first network which converges and has test loss of 0.001.*

Problem B [5 points]: Backpropagation and Weight Initialization Part 2

Reset the two demos from part i (there is a reset button to the left of the "Run" button), change the activation functions of the neurons to sigmoid instead of ReLU, and train each of them for 4000 iterations.

Explain the differences in the models learned, and the speed at which they were learned, from those of part i in terms of the backpropagation algorithm and the sigmoid function.

Solution B.: *With respect to the first neural network, we see that changing our activation function to sigmoid instead of ReLU greatly slows down the convergence process and even at the end of 4000 epochs, we still have worse test performance (0.002 vs 0.001). The reason why ReLU activation functions converge faster is because it is a non saturating function. In other words, when a neuron's weighted sum is a large or small value, the gradient does not saturate to 0. Therefore, we can learn the features of the data much quicker (larger weight changes). When we analyze the second neural network, we see that changing the activation function to sigmoid instead of ReLU does improve the performance of our model but only very slightly. Our sigmoid activation function produces a non zero value which allows the weights to change during backpropagation, but because we initialize the weights to the same values, each neuron performs the same job and will therefore output the same values. Therefore, we have excess neurons which slows down model convergence.*

Problem C: [10 Points]

When training any model using SGD, it's important to shuffle your data to avoid correlated samples. To illustrate one reason for this that is particularly important for ReLU networks, consider a dataset of 1000 points, 500 of which have positive (+1) labels, and 500 of which have negative (-1) labels. What happens if we train a fully-connected network with ReLU activations using SGD, looping through all the negative examples before any of the positive examples? (Hint: this is called the "dying ReLU" problem.)

Solution C: *If we train a fully-connected network with ReLU activations using SGD but we loop through all the negative examples before any of the positive examples, we experience the dying ReLU problem. During the initial stages of the model training, the network only faces negative labels, and therefore the network is optimized during backpropagation to alter the weights such that the weighted sums to the ReLU activation functions of the neurons are negative. When the weights are altered in this manner to where they only produce negative sums, the ReLU function returns 0. At this point, the gradient of the Loss function with respect to the neuron's weights will also be 0 and will not update. Therefore, once we reach the positive examples, the neurons are essentially "dead" and will no longer update their weights, leading to a poor model.*

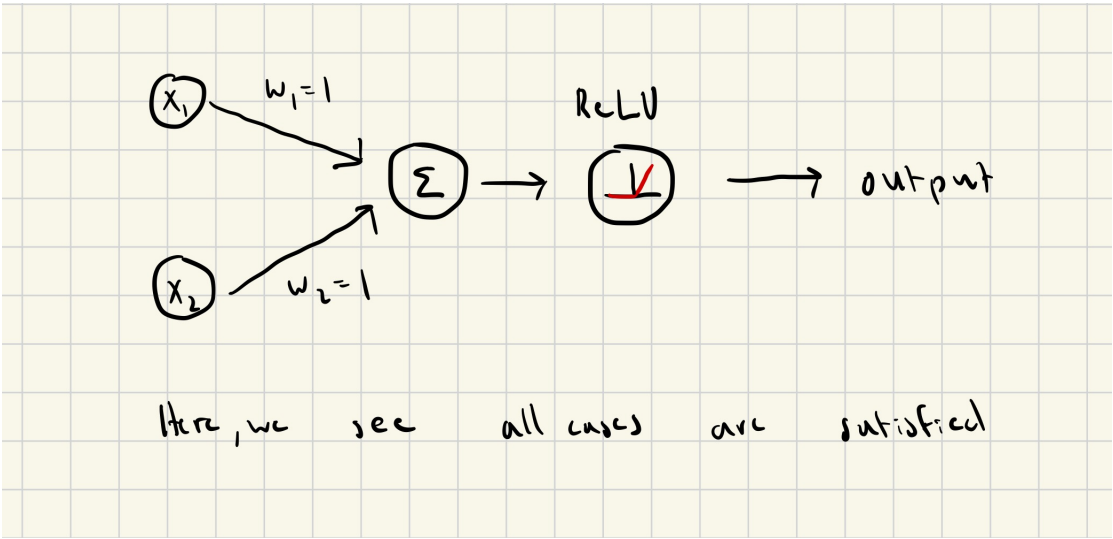
Problem D: Approximating Functions Part 1 **[7 Points]**

Draw or describe a fully-connected network with ReLU units that implements the OR function on two 0/1-valued inputs, x_1 and x_2. Your networks should contain the minimum number of hidden units possible. The OR function OR(x_1, x_2) is defined as:

$$\text{OR}(1,0) \geq 1$$
$$\text{OR}(0,1) \geq 1$$
$$\text{OR}(1,1) \geq 1$$
$$\text{OR}(0,0) = 0$$

Your network need only produce the correct output when $x_1 \in \{0,1\}$ and $x_2 \in \{0,1\}$ (as described in the examples above).

Solution D.:



Problem E: Approximating Functions Part 2 **[8 Points]**

What is the minimum number of fully-connected layers (with ReLU units) needed to implement an XOR of two 0/1-valued inputs x_1, x_2? Recall that the XOR function is defined as:

$$\text{XOR}(1,0) \geq 1$$
$$\text{XOR}(0,1) \geq 1$$
$$\text{XOR}(0,0) = \text{XOR}(1,1) = 0$$

For the purposes of this problem, we say that a network f computes the XOR function if $f(x_1, x_2) = \text{XOR}(x_1, x_2)$ when $x_1 \in \{0,1\}$ and $x_2 \in \{0,1\}$ (as described in the examples above).

Explain why a network with fewer layers than the number you specified cannot compute XOR.

Solution E.: *The minimum number of fully-connected layers needed to implement an XOR of two 0/1-valued inputs x1 and x2 is 2. This is because XOR itself is not a linear function, it is the combination of linearly separable functions OR and AND. Therefore, this combination must occur over two layers. If we had less than 2 layers, we could not compute XOR because we would only have one linear separating line, which cannot implement XOR.*

2 Depth vs Width on the MNIST Dataset [25 Points]

MNIST is a classic dataset in computer vision. It consists of images of handwritten digits (0 - 9) and the correct digit classification. In this problem you will implement a deep network using PyTorch to classify MNIST digits. Specifically, you will explore what it really means for a network to be "deep", and how depth vs. width impacts the classification accuracy of a model. You will be allowed at most N hidden units, and will be expected to design and implement a deep network that meets some performance baseline on the MNIST dataset.

Problem A: Installation [2 Points]

Before any modeling can begin, PyTorch must be installed. PyTorch is an automatic differentiation framework that is widely used in machine learning research. We will also need the **torchvision** package, which will make downloading the MNIST dataset much easier.

To install both packages, follow the steps on
https://pytorch.org/get-started/locally/#start-locally. Select the 'Stable' build and your system information. We highly recommend using Python 3.6+. CUDA is not required for this class, but it is necessary if you want to do GPU-accelerated deep learning in the future.

Once you have finished installing, write down the version numbers for both **torch** and **torchvision** that you have installed.

Solution A:

https://colab.research.google.com/drive/147aqE_-iddhiCT4TSli9mYwxW6Z1Blg9?usp=sharing

torch: 2.1.0+cu121

torchvision: 0.16.0+cu121

Problem B: The Data [3 Points]

Load the MNIST dataset using torchvision; see the problem 2 sample code for how.

Image inputs in PyTorch are generally 3D tensors with the shape (no. of channels, height, width). Examine the input data. What are the height and width of the images? What do the values in each array index represent? How many images are in the training set? How many are in the testing set? You can use the **imshow** function in matplotlib if you'd like to see the actual pictures (see the sample code).

Solution B.: *The height and width of the images are 28 pixels each. We know this because we have 784 input features, which is 28 squared. The rows of the array represent a an image and the columns/indices of the array represent features of the image/number such as intensity. There are 60,000 total images in the training set and 10,000 images in the test set.*

Problem C: Modeling Part 1 [8 Points]

Using PyTorch's "Sequential" model class, build a deep network to classify the handwritten digits. You may **only** use the following layers:

- **Linear:** A fully-connected layer
- **ReLU (activation):** Sets negative inputs to 0
- **Softmax (activation):** Rescales input so that it can be interpreted as a (discrete) probability distribution.
- **Dropout:** Takes some probability and at every iteration sets weights to zero at random with that probability (effectively regularization)

A sample network with 20 hidden units is in the sample code file. (Note: activations, Dropout, and your last Linear layer do not count toward your hidden unit count, because the final layer is "observed" and not *hidden*.)

Use categorical cross entropy as your loss function. There are also a number of optimizers you can use (an optimizer is just a fancier version of SGD), and feel free to play around with them, but RMSprop and Adam are the most popular and will probably work best. You also should find the batch size and number of epochs that give you the best results (default is batch size = 32, epochs=10).

Look at the sample code to see how to train your model. PyTorch should make it very easy to tinker with your network architecture.

Your task. Using at most 100 hidden units, build a network using only the allowed layers that achieves test accuracy of at least 0.975. Turn in the code of your model as well as the best test accuracy that it achieved.

Hint: for best results on this problem and the two following problems, normalize the input vectors by dividing the values by 255 (as the pixel values range from 0 to 255).

Solution C:

https://colab.research.google.com/drive/147aqE_-iddhiCT4TSli9mYwxW6Z1Blg9?usp=sharing

```
model = nn.Sequential(
    # In problem 2, we don't use the 2D structure of an image at all. Our network
    # takes in a flat vector of the pixel values as input.
    nn.Flatten(),
    nn.Linear(784, 100),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(100, 10),
)
```

Test set: Average loss: 0.0025, Accuracy: 9776/10000 (97.7600)

Problem D: Modeling Part 2 [6 Points]

Repeat problem C, except that now you may use 200 hidden units and must build a model with at least 2 hidden layers that achieves test accuracy of at least 0.98.

Solution D:

```
model = nn.Sequential(
    # In problem 2, we don't use the 2D structure of an image at all. Our network
    # takes in a flat vector of the pixel values as input.
    nn.Flatten(),
    nn.Linear(784, 140),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(140, 60),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(60, 10)
)
```

Test set: Average loss: 0.0023, Accuracy: 9803/10000 (98.0300)

Problem E: Modeling Part 3 [6 Points]

Repeat problem C, except that now you may use 1000 hidden units and must build a model with at least 3 hidden layers that achieves test accuracy of at least 0.983.

Solution E:

```
model = nn.Sequential(
    # In problem 2, we don't use the 2D structure of an image at all. Our network
    # takes in a flat vector of the pixel values as input.
    nn.Flatten(),
    nn.Linear(784, 400),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(400, 300),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(300, 200),
    nn.ReLU(),
    nn.Dropout(0.25),
    nn.Linear(200, 70),
    nn.ReLU(),
    nn.Dropout(0.25),
    nn.Linear(70, 10)
)
```

Test set: Average loss: 0.0033, Accuracy: 9835/10000 (98.3500)

3 Convolutional Neural Networks [40 Points]

Problem A: Zero Padding **[5 Points]**

Consider a convolutional network in which we perform a convolution over each 8×8 patch of a 20×20 input image. It is common to zero-pad input images to allow for convolutions past the edges of the images. An example of zero-padding is shown below:

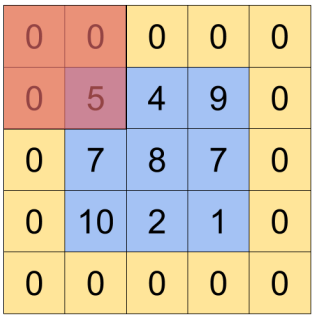


Figure: A convolution being applied to a 2×2 patch (the red square) of a 3×3 image that has been zero-padded to allow convolutions past the edges of the image.

What is one benefit and one drawback to this zero-padding scheme (in contrast to an approach in which we only perform convolutions over patches entirely contained within an image)?

Solution A: *A benefit to the zero-padding scheme is that the feature map will be the same dimensions as the input. With zero-padding, we produce a 3x3 feature map, but without it we produce a 2x2 feature map. A drawback is that the process is computationally expensive as we have to process irrelevant zeros.*

5 x 5 Convolutions

Consider a single convolutional layer, where your input is a 32×32 pixel, RGB image. In other words, the input is a $32 \times 32 \times 3$ tensor. Your convolution has:

- Size: $5 \times 5 \times 3$
- Filters: 8
- Stride: 1
- No zero-padding

Problem B [2 points]: What is the number of parameters (weights) in this layer, including a bias term?

Solution B.: *Each filter has* $5 * 5 * 3 + 1 = 76$ *weights. Then, for eight filters, we have* $76 * 8 = 608$ *weights.*

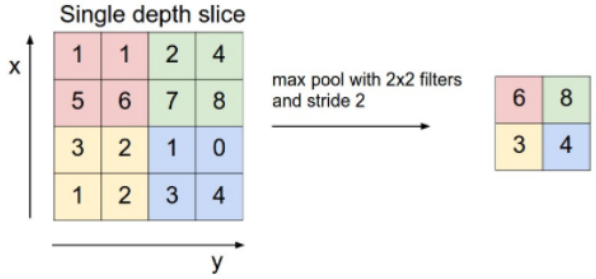
Problem C [3 points]: What is the shape of the output tensor?

Solution C.: *The output size for both width and height will be* $\frac{32-5}{1} + 1 = 28$. *Then, our final shape is 28 x 28 x 3.*

Max/Average Pooling

Pooling is a downsampling technique for reducing the dimensionality of a layer's output. Pooling iterates across patches of an image similarly to a convolution, but pooling and convolutional layers compute their outputs differently: given a pooling layer B with preceding layer A, the output of B is some function (such as the max or average functions) applied to patches of A's output.

Below is an example of max-pooling on a 2-D input space with a 2×2 filter (the max function is applied to 2×2 patches of the input) and a stride of 2 (so that the sampled patches do not overlap):



Average pooling is similar except that you would take the average of each patch as its output instead of the maximum.

Consider the following 4 matrices:

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Problem D [3 points]:

Apply 2×2 average pooling with a stride of 2 to each of the above images.

Solution D.:

$$\begin{bmatrix} 1 & 0.5 \\ 0.5 & 0.25 \end{bmatrix}, \begin{bmatrix} 0.5 & 1 \\ 0.25 & 0.5 \end{bmatrix}, \begin{bmatrix} 0.25 & 0.5 \\ 0.5 & 1 \end{bmatrix}, \begin{bmatrix} 0.5 & 0.25 \\ 1 & 0.5 \end{bmatrix}$$

Problem E [3 points]:

Apply 2×2 max pooling with a stride of 2 to each of the above images.

Solution E.:

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Problem F [4 points]:

Consider a scenario in which we wish to classify a dataset of images of various animals, taken at various angles/locations and containing small amounts of noise (e.g. some pixels may be missing). Why might pooling be advantageous given these distortions in our dataset?

Solution F.: *Some pixels are likely distorted since our images contain small amounts of noise. Therefore, after the feature map is created, we can mitigate the distorted output pixels by pooling. This helps mitigate the adverse effect that the noise has by taking an average or a maximum over a given area.*

PyTorch implementation

Problem G [20 points]:

Using PyTorch "Sequential" model class as you did in 2C, build a deep *convolutional* network to classify the handwritten digits in MNIST. You are now allowed to use the following layers (but **only** the following):

- **Linear:** A fully-connected layer
  - In convolutional networks, Linear (also called dense) layers are typically used to knit together higher-level feature representations.
  - Particularly useful to map the 2D features resulting from the last convolutional layer to categories for classification (like the 1000 categories of ImageNet or the 10 categories of MNIST).
  - Inefficient use of parameters and often overkill: for A input activations and B output activations, number of parameters needed scales as $O(AB)$.
- **Conv2d:** A 2-dimensional convolutional layer
  - The bread and butter of convolutional networks, conv layers impose a translational-invariance prior on a fully-connected network. By sliding filters across the image to form another image, conv layers perform "coarse-graining" of the image.
  - Networking several convolutional layers in succession helps the convolutional network knit together more abstract representations of the input. As you go higher in a convolutional network, activations represent pixels, then edges, colors, and finally objects.
  - More efficient use of parameters. For N filters of $K \times K$ size on an input of size $L \times L$, the number of parameters needed scales as $O(NK^2)$. When N, K are small, this can often beat the $O(L^4)$ scaling of a Linear layer applied to the L^2 pixels in the image.
- **MaxPool2d:** A 2-dimensional max-pooling layer
  - Another way of performing "coarse-graining" of images, max-pool layers are another way of ignoring finer-grained details by only considering maximum activations over small patches of the input.
  - Drastically reduces the input size. Useful for reducing the number of parameters in your model.
  - Typically used immediately following a series of convolutional-activation layers.
- **BatchNorm2d:** Performs batch normalization (Ioffe and Szegedy, 2014). Normalizes the activations of previous layer to standard normal (mean 0, standard deviation 1).
  - Accelerates convergence and improves performance of model, especially when saturating non-linearities (sigmoid) are used.
  - Makes model less sensitive to higher learning rates and initialization, and also acts as a form of regularization.

  - Typically used immediately before nonlinearity (Activation) layers.
- **Dropout:** Takes some probability and at every iteration sets weights to zero at random with that probability
  - An effective form of regularization. During training, randomly selecting activations to shut off forces network to build in redundancies in the feature representation, so it does not rely on any single activation to perform classification.
- **ReLU (activation):** Sets negative inputs to 0
- **Softmax (activation):** Rescales input so that it can be interpreted as a (discrete) probability distribution.
- **Flatten:** Flattens any tensor into a single vector (required in order to pass a 2D tensor output from a convolutional layer as input into Linear layers)

Your tasks. Build a network with only the allowed layers that achieves **test accuracy of at least 0.985**. You are required to use categorical cross entropy as your loss function and to train for 10 epochs with a batch size of 32. Note: your model must have fewer than 1 million parameters, as measured by the method given in the sample code. Everything else can change: optimizer (RMSProp, Adam, ???), initial learning rates, dropout probabilities, layerwise regularizer strengths, etc. You are not required to use all of the layers, but *you must have at least one dropout layer and one batch normalization layer in your final model*. Try to figure out the best possible architecture and hyperparameters given these building blocks!

In order to design your model, you should train your model for 1 epoch (batch size 32) and look at the final **test accuracy** after training. This should take no more than 10 minutes, and should give you an immediate sense for how fast your network converges and how good it is.

Set the probabilities of your dropout layers to 10 equally-spaced values $p \in [0, 1]$, train for 1 epoch, and report the final model accuracies for each.

You can perform all of your hyperparameter validation in this way: vary your parameters and train for an epoch. After you're satisfied with the model design, you should train your model for the full 10 epochs.

In your submission. Turn in the code of your model, the test accuracy for the 10 dropout probabilities $p \in [0, 1]$, and the final test accuracy when your model is trained for 10 epochs. We should have everything needed to reproduce your results.

Discuss what you found to be the most effective strategies in designing a convolutional network. Which regularization method was most effective (dropout, layerwise regularization, batch norm)?

Do you foresee any problem with this way of validating our hyperparameters? If so, why?

Hints:

- You are provided with a sample network that achieves a high accuracy. Starting with this network, modify some of the regularization parameters (layerwise regularization strength, dropout probabil-

ities) to see if you can maximize the test accuracy. You can also add layers or modify layers (e.g. changing the convolutional kernel sizes, number of filters, stride, dilation, etc.) so long as the total number of parameters remains under the cap of 1 million.

- You may want to read up on successful convolutional architectures, and emulate some of their design principles. Please cite any idea you use that is not your own.
- To better understand the function of each layer, check the PyTorch documentation.
- Linear layers take in single vector inputs (ex: *(784,)*) but Conv2D layers take in tensor inputs (ex: *(28, 28, 1)*): width, height, and channels. Using the transformation `transforms.ToTensor()` when loading the dataset will reshape the training/test X to a 4-dimensional tensor (ex: *(num_examples, width, height, channels)*) and normalize values. For the MNIST dataset, *channels=1*. Typical color images have 3 color channels, 1 for each color in RGB.
- If your model is running slowly on your CPU, try making each layer smaller and stacking more layers so you can leverage deeper representations.
- Other useful CNN design principles:
  - CNNs perform well with many stacked convolutional layers, which develop increasingly large-scale representations of the input image.
  - Dropout ensures that the learned representations are robust to some amount of noise.
  - Batch norm is done after a convolutional or dense layer and immediately prior to an activation/nonlinearity layer.
  - Max-pooling is typically done after a series of convolutions, in order to gradually reduce the size of the representation.
  - Finally, the learned representation is passed into a dense layer (or two), and then filtered down to the final softmax layer.

Solution G.:

https://colab.research.google.com/drive/1KuZ3cDRoSZ01wOb11YiU4PdugBRzvQ0W?usp=sharing

Model:

```
model = nn.Sequential(
    nn.Conv2d(1, 32, kernel_size=(3,3)),
    nn.BatchNorm2d(num_features=32),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Dropout(p=0.1),

    nn.Conv2d(32, 16, kernel_size=(3,3)),
    nn.BatchNorm2d(num_features=16),
    nn.ReLU(),
    nn.Dropout(p=0.1),

    nn.Conv2d(16, 8, kernel_size=(3,3)),
    nn.BatchNorm2d(num_features=8),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Dropout(p=0.1),

    nn.Flatten(),
    nn.Linear(128, 64),
    nn.BatchNorm1d(num_features=64),
    nn.ReLU(),
    nn.Linear(64, 10)
    # PyTorch implementation of cross-entropy loss includes softmax layer
)
```

Test accuracy for varying Dropout parameters (0.1 - 1.0):

0.9842, 0.9840, 0.9810, 0.9762, 0.9728, 0.9569, 0.9414, 0.8697, 0.7899, 0.0975

Final Test Accuracy after 10 epochs:

0.9916

When designing the convolutional network, I found the hints provided in the problem very helpful. I first tried to modify the regularization parameters (layerwise regularization strength, dropout probabilities). What I realized was that 2-D Batch Normalization was the most effective regularization method and so I used it after each successive convolution that I implemented. I even used batch normalization after my final linear layer. This greatly improved test accuracy. Another thing I realized was that decreasing my dropout parameter increased test accuracy. This could be due to batch normalization acting as a form of regularization, making it less important to have a high dropout parameter. Besides from regularization tactics, I added another convolutional layer to the sample code because the problem hints said that CNN's perform well with many stacked convolutional layers, and they are also an efficient use of parameters.

A potential problem with this way of validating our hypothesis is that we are biasing our model on the validation data. Although we are not directly feeding the model the validation data for training purposes, we are reading the validation errors and tuning our hyperparameters to increase test accuracy. In the future, it might be better to perform cross-validation to ensure less bias.