

## Policies

- Due 9 PM, February 7<sup>th</sup>, via Gradescope.
- You are free to collaborate on all of the problems, subject to the collaboration policy stated in the syllabus.
- You should submit all code used in the homework. We ask that you use Python 3.6+ and PyTorch version 1.4.0 for your code, and that you comment your code such that the TAs can follow along and run it without any issues.
- This set requires the installation of PyTorch. There will be a recitation and office hour dedicated to helping you install these packages if you have problems.

## Submission Instructions

- Submit your report as a single .pdf file to Gradescope.
- In the report, **include any images generated by your code** along with your answers to the questions.
- Submit your code by **sharing a link in your report** to your Google Colab notebook for each problem (see naming instructions below). Make sure to set sharing permissions to at least "Anyone with the link can view". **Links that can not be run by TAs will not be counted as turned in.** Check your links in an incognito window before submitting to be sure.

## TA Office Hours

- **Julio Arroyo**
  - Sunday, 2/5: 3:00 pm - 4:00 pm
  - Monday, 2/6: 8:00 pm - 9:00 pm
- **Sreemanti Dey**
  - Friday, 2/3: 7:00 pm - 8:00 pm
  - Monday, 2/6: 7:00 pm - 8:00 pm

## 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 converged rapidly, whereas the second network stayed around 50% training and test loss.*

*This is because the second network started with all 0 weights, which is saddle point in the loss function, so the gradient is 0 and gradient descent never changes the weights. The first network, starting with random weights, behaved "normally".*

### 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.:** *The first model learned a smoother looking decision boundary compared to ReLU, and it also learned much slower. The smoothness is because sigmoid has a smooth shape compared to ReLU's sharp corner. The difference in speed is because the sigmoid outputs get squished to be between 0 and 1, whereas ReLU outputs can be as large as the inputs, so sigmoid requires larger weights which took a while to reach starting from weights near zero.*

*The second model stayed with all its weights near zero for a long time (like part A), though it eventually changed to a linear decision boundary with all the weights after the first hidden layer being identical.*

### 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 why this is particularly important for ReLU networks (i.e. it has ReLU activation

functions between its hidden layers, but its output layer is still softmax/tanh/linear), 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? Do not assume that the derivative of ReLU at 0 is implemented as 0. *Hint: this is called the “dying ReLU” problem, although it is possible with other activation functions.*

**Solution C:** With 500 samples with the same label, the model will learn to output only that label. The simplest way to do that is by introducing a large negative bias in some hidden layer, so that the ReLU outputs from that layer are always zero and the following layer can just have a bias of whatever the (constant) label should be. The result is that an entire layer is pushed well below the zero point in ReLU, into the region where ReLU is flat. Then there is no gradient, so the optimizer doesn't know what to do when presented with the +1 labels.

**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  $\text{OR}(x_1, x_2)$  is defined as:

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

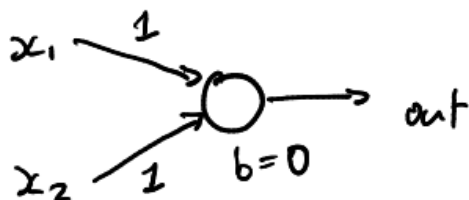
$$\text{OR}(0, 1) \geq 1$$

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

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

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]**

Late hours used: 6

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:

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

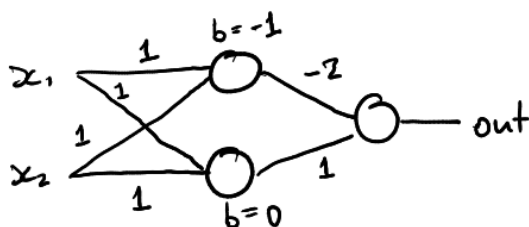
$$\text{XOR}(0, 1) \geq 1$$

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

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.:**



---

## 2 Inside a Neural Network [17 Points]

*Relevant Materials: Lectures on Deep Learning*

Although this is no longer the peak of the pandemic, coronavirus datasets are still very salient due to the number of people in the US that are still being affected by the disease. In this problem, you will investigate the workings of a simple neural network by designing linear neural nets to classify coronavirus cases.

### 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 later on, 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:**

*torch: 1.13.1+cu116*

*torchvision: 0.41.1+cu116*

### Problem B: The Data [5 Points]

Load and preprocess the tabular dataset, "COVID-19\_Case\_Surveillance\_Public\_Use\_Data\_Subset.csv." This is a small subset of the CDC's Covid-19 Case Surveillance data (<https://data.cdc.gov/Case-Surveillance/COVID-19-Case-Surveillance-Public-Use-Data/vbim-akqf>). Limit the number of input variables in your final dataset to be between 7 and 12, and use the "death\_yn" column as the dependent variable. You might find various features in the pandas package useful here.

Explain your preprocessing decisions.

**Solution B:** *I chose to use sex, current status (probable or lab-confirmed), age group, race/ethnicity, hospitalization, ICU, and presence of medical conditions as input variables.*

*For sex, current status, hospitalization, ICU, and medical conditions, there are only two possible values, so I converted them to 1 or 0. For age group and race/ethnicity, there are multiple values, so I used one-hot encoding.*

**Problem C: Linear Neural Network [5 Points]**

Now, use PyTorch's "Sequential" class to make a neural network with one linear layer of size 5 and a binary output layer (with softmax, since this is a classification task). Do not use any activation function in your linear layer. Choose an appropriate optimizer, learning rate, and loss function (either Adam or RMSProp will probably work best as an optimizer). Use these to train and test your model on your dataset—look at the problem 3 sample code to see how to train and test.

Finally, visualize the weight vectors in your model with a heatmap.

Make sure to show training losses and test accuracy clearly in your notebook.

**Solution C:** *Colab notebook*

*Training loss: 0.0023*

*Test loss: 0.0024*

**Problem D: 2-Layer Linear Neural Network [5 Points]**

Finally, create and train a 2-layer linear neural network and assess its performance on your dataset. It is expected that the 1-layer and 2-layer models have very similar losses—why is this the case?

**Solution D:** *Its performance is identical to the 1-layer network. This is because there's no activation function, so the 2-layer network is just a composition of 2 linear operations, which is still a linear operation and can thus be accomplished by a single layer.*

### 3 Depth vs Width on the MNIST Dataset [23 Points]

*Relevant Materials: Lectures on Deep Learning*

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: The Data [3 Points]

Load the MNIST dataset using torchvision; see the problem 3 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 A.:** *The images are 28x28 pixels. The values in the array are brightness of the pixels. There are 60,000 images in the training set and 10,000 in the testing set.*

#### Model submission instructions:

For each problem 3C-3E and 4G there should be a separate notebook. In your notebook, include the code you used to train your model and make sure your results are visible.

#### Problem B: 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.

**Important note on stochasticity:** For problems 3C-3E and 4G, you might notice that your model’s accuracy fluctuates every time you train it. This is caused by weight initialization, shuffled mini-batching for SGD, dropout probabilities, etc. You may want to consider controlling the effects of randomness by **manually setting the seed**. In any case, when we say “achieve test accuracy of at least  $x$ ”, we mean that your model should achieve the stated accuracy more than half the times you train it.

*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 B:** [Colab notebook](#)

### Problem C: 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 C:** [Colab notebook](#)

### Problem D: 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 D:** [Colab notebook](#)



## 4 Convolutional Neural Networks [40 Points]

*Relevant Materials: Lecture on CNNs*

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

Consider a convolutional network in which we perform a convolution over each  $8 \times 8$  patch of a  $20 \times 20$  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:

|   |    |   |   |   |
|---|----|---|---|---|
| 0 | 0  | 0 | 0 | 0 |
| 0 | 5  | 4 | 9 | 0 |
| 0 | 7  | 8 | 7 | 0 |
| 0 | 10 | 2 | 1 | 0 |
| 0 | 0  | 0 | 0 | 0 |

Figure: A convolution being applied to a  $2 \times 2$  patch (the red square) of a  $3 \times 3$  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:** *One benefit is that we get an output the same size as the input (otherwise the output would be shrinking with every layer).*

*One drawback is that this reduces the translational invariance of the process, since if the image were actually larger the pixels beyond the edge wouldn't really be black.*

### 5 x 5 Convolutions

Consider a single convolutional layer, where your input is a  $32 \times 32$  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? What is the shape of the output tensor?

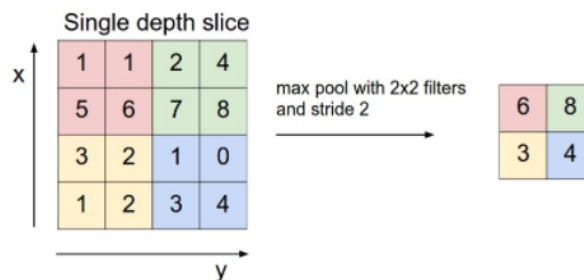
**Solution B.:** *Numer of parameters:  $5 \cdot 5 \cdot 3 \cdot 8 + 8 = 608$  (one bias term for each filter).*

*Shape of the output tensor:  $28 \times 28 \times 8$  (losing 2 pixels from each edge because of no padding).*

## 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  $2 \times 2$  filter (the max function is applied to  $2 \times 2$  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 C [3 points]:**

1. Apply  $2 \times 2$  average pooling with a stride of 2 to each of the above images.
2. Apply  $2 \times 2$  max pooling with a stride of 2 to each of the above images.

**Solution C.:**

1. *Average pooling:*

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

2. *Max pooling:*

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

**Problem D [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 D.:** Pooling would smooth out the noise, so that pixels that should be a certain color are more consistently that color.

## Hyperparameter Tuning

The performance of neural networks depends to a large extent on the choice of hyperparameters. However, theory does not give us a systematic way of how to pick them. In practice, there are at least three different strategies to tune a network.

1. **"Babysitting":** When tuning a model, you can manually choose a set of hyperparameters, monitor in- and out-of-sample performance, adjust according to intuition, and train again.
2. **Grid search:** For each hyperparameter (learning rate, dropout probability, etc), you define a *range* of values to search, and an *interval* at which to sample points. For example, you could try learning rates  $lr = \{10^{-1}, 10^{-2}, \dots, 10^{-5}\}$  and dropout probabilities  $p = \{0, 0.1, \dots, 0.5\}$ , and then train your model for each  $(lr, p)$  pair, and keep the setting that yields best results.
3. **Random search:** You can again define a search range, but instead of testing points at pre-determined intervals, you could sample points at random and train your model with those hyperparameters. You

could take it one step further and implement a “coarse-to-fine” approach: i.e. define a large search space, sample and test a few hyperparameters, and then repeat the process inside a finer search space around those hyperparameters that yielded best results in the first pass.

**Problem E [5 points]:** Pick one of the three hyperparameter tuning strategies, and explain at least one strength and one limitation of using that strategy to tune your model.

**Solution E.:** *With babysitting, a strength is that you can use information about how neural networks work to identify behavior like overfitting and adjust hyperparameters accordingly. A limitation is that you have to monitor the training and can't do other work.*

## PyTorch implementation

**Problem F [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 nonlinearities (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 probabilities) 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 F:** *Colab notebook*

| <i>Dropout Prob.</i> | <i>Accuracy</i> |
|----------------------|-----------------|
| 0                    | 0.9826          |
| 0.1                  | 0.9782          |
| 0.2                  | 0.9771          |
| 0.3                  | 0.9728          |
| 0.4                  | 0.9741          |
| 0.5                  | 0.9779          |
| 0.6                  | 0.9649          |
| 0.7                  | 0.9612          |
| 0.8                  | 0.9209          |
| 0.9                  | 0.7935          |

*I found that dropout of 0.1 was more reliable than 0 when training for more than one epoch, so I use that in my final model. This is probably because overfitting doesn't happen yet with such a small amount of training.*

*Batch normalization made a huge difference in the performance of my model, though I don't think it helped much to prevent overfitting.*

*Final test accuracy: 0.9883.*