Machine Learning & Data Mining
Set 4

Caltech CS/CNS/EE 155
January 26th, 2022

Dallas Taylor

Google Colab Links

- Problem 2 : https://colab.research.google.com/drive/1$_B$b6vB3U2ZdwDe8ssW3Alnt80Pj79PAP?usp=sharing
- Problem 3 : https://colab.research.google.com/drive/14hOuwNsMvkmSzAVRGGsvY5hGzZDY2HzH?usp=sharing

Dallas Taylor

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.: *These two networks perform differently because the neural network at the first link has its weights initialized as random values between -1 and 1 while the second link has its weights all initialized as 0. This results in the network at the first link producing highly accurate results (train and test loss* ~0.001*) while the network at the second link does not learn properly and is never able to update its weight from the initial 0 value (train and test loss* ~0.5*). This can be explained by noticing that the gradient of the ReLU activation function at 0 is also 0 (by the definition of the max function as we are taking max (0, x)). Thus, when our initial weights are randomly initialized, we are able to avoid ReLU saturation (as would be noticed if using only negative or 0-valued weights). Additionally, when our initial weights are 0, the activations for each layer of our second network will be 0, producing corresponding gradients of 0 between each layer. This pattern also continues across iterations, resulting in the weights permanently staying at a value of 0.*

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 models using sigmoid activation functions learned much slower than those using ReLU activation. This is due to the fact that sigmoid activation functions operate by performing exponential operations, which take up much more time to compute than the simple max function utilized in ReLU. Sigmoid activations also cause models to train slower due to the fact that the gradients are very small everywhere (approaches 0 at* $\pm\infty$*) and thus only small updates are applied to the weights at each iteration.*

When considering the network at the first link, we see that our desired boundary begins to form at epoch 700 *(vs epoch* 100 *for ReLU). We additionally see that our boundary edges are much smoother, most likely due to the use of soft thresholds.*

When considering the network at the second link, we can see that even though our second network did actually begin to properly learn the data around epoch 3200 *(vs how ReLU activation caused this function to saturate at 0 and halt learning), that we still see a much lower level of learning than our first network with sigmoid. This decrease in speed similarly correlates to how our weights were initialized as 0 in this network (as our gradients will be much smaller than if our weights randomly initialized). Thus, our sigmoid activation function will still be able to operate, but will operate very slowly due to near-zero sigmoid gradients.*

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 loop through all of our negative examples before the positive examples when using ReLU activation, we will see that our model will produce a very large negative bias. Thus, the positive points will most likely continue to be classified as negative due to our bias term. Furthermore, we will see that sums calculated during an iteration will be negative, producing a value of 0 through ReLU activation. Thus, we will "break" our model and observe saturation at 0. This leaves us with a model that is no longer able to properly learn and update weights.*

Dallas Taylor

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.:

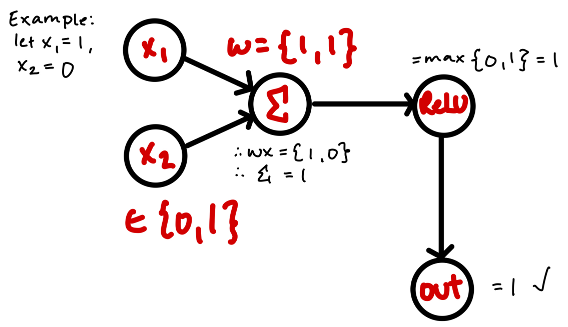


Figure 1: Fully Connected Network with ReLU Activation - OR

It can be seen that our network above operates properly to calculate our OR functionality. We can see that we would have four possible inputs to our network:

- $x_1 = 0, x_2 = 0 : \sum \boldsymbol{wx} = 0 \cdot 1 + 0 \cdot 1 = 0$ *(false)*
- $x_1 = 1, x_2 = 0 : \sum \boldsymbol{wx} = 1 \cdot 1 + 0 \cdot 1 = 1$ *(true)*
- $x_1 = 0, x_2 = 1 : \sum \boldsymbol{wx} = 0 \cdot 1 + 1 \cdot 1 = 1$ *(true)*
- $x_1 = 1, x_2 = 1 : \sum \boldsymbol{wx} = 1 \cdot 1 + 1 \cdot 1 = 0$ *(true)*

Dallas Taylor

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 is 2 layers. A network with fewer layers than this is unable to compute XOR because each layer of our ReLU activation is only able to compute linear classifiers (and our XOR function is not linearly separable). Thus, in order to calculate our XOR function, we require our first layer to split our dataset (for example we could compute the AND and OR functions in our hidden layer) and our second layer is then able to properly classify the data (use outputs from AND & OR in the previous layer to determine correct classification).*

Dallas Taylor

2 Depth vs Width on the MNIST Dataset [25 Points, 6 EC 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: 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.

If you use Google Colab (recommended), you won’t need to install anything.

If you want to run PyTorch locally, 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.

Write down the version numbers for both **torch** and **torchvision** that you have installed. On Google Colab, you can find version numbers by running:

```
!pip list | grep torch
```

Solution A:

Torch: 1.10.0+cu111

Torchvision: 0.11.1+cu111

Colab Link : *https://colab.research.google.com/drive/1$_{\mathrm{B}}$b6vB3U2ZdwDe8ssW3Alnt80Pj79PAP?usp=sharing*

Dallas Taylor

Problem B: The Data [5 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 *and* 28, *respectively. Each row within our datasets represent each input image. Each value within each input image represent the grey-scale intensities within the image (*0 *is black and* 1 *is white). There is a total of* 60000 *images in the training set and* 10000 *images in the testing set.*

Problem C: Modeling Part 1 [10 Points]

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

- **Flatten:** Flattens any tensor into a single vector
- **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)

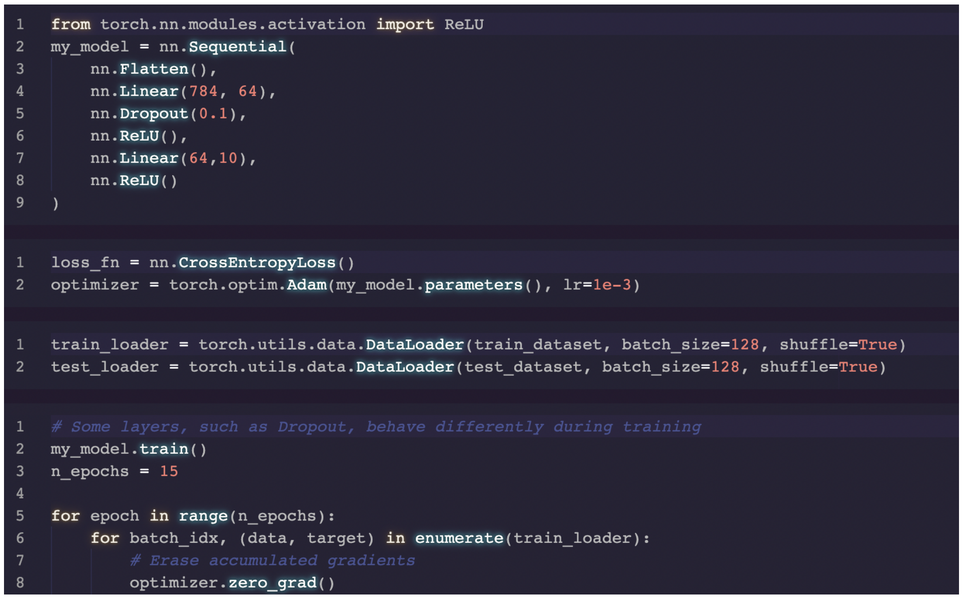
A sample network with 20 hidden units is in the sample code file. (Note: You may use multiple layers as long as the total number of hidden units are within the limit. Activations, Dropout, and your last Linear layer do not count toward your hidden unit count, because the final layer is "observed" and 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. You can tinker with the network architecture by swapping around layers and parameters.

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.

Solution C:

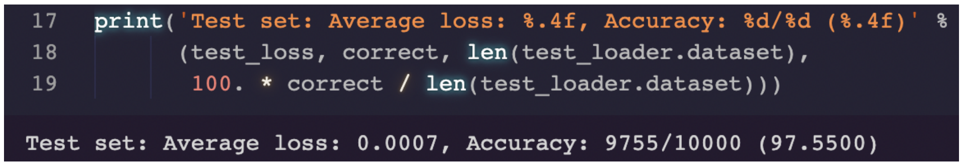


```
from torch.nn.modules.activation import ReLU
my_model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 64),
    nn.Dropout(0.1),
    nn.ReLU(),
    nn.Linear(64,10),
    nn.ReLU()
)
```

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(my_model.parameters(), lr=1e-3)
```

```
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=128, shuffle=True)
```

```
# Some layers, such as Dropout, behave differently during training
my_model.train()
n_epochs = 15

for epoch in range(n_epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        # Erase accumulated gradients
        optimizer.zero_grad()
```



```
print('Test set: Average loss: %.4f, Accuracy: %d/%d (%.4f)' %
        (test_loss, correct, len(test_loader.dataset),
         100. * correct / len(test_loader.dataset)))
```

```
Test set: Average loss: 0.0007, Accuracy: 9755/10000 (97.5500)
```

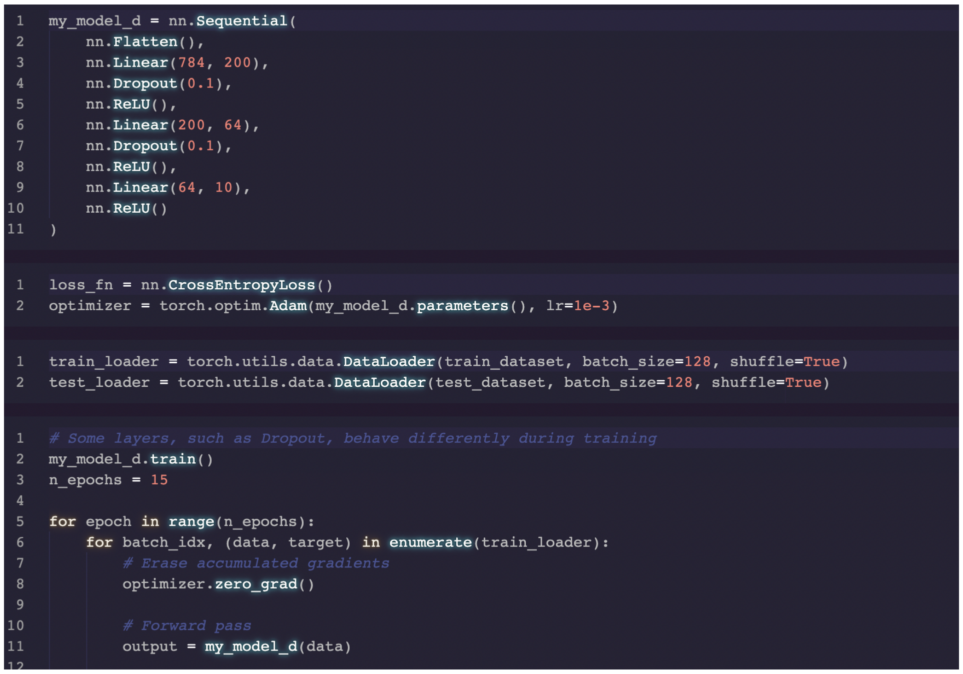
Note that the rest of the loop through n_epochs is the same code as presented in the sample code training loop.

Dallas Taylor

Problem D: Modeling Part 2 [8 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:

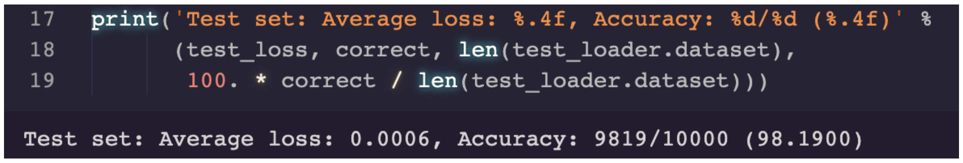


```
my_model_d = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 200),
    nn.Dropout(0.1),
    nn.ReLU(),
    nn.Linear(200, 64),
    nn.Dropout(0.1),
    nn.ReLU(),
    nn.Linear(64, 10),
    nn.ReLU()
)
```

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(my_model_d.parameters(), lr=1e-3)
```

```
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=128, shuffle=True)
```

```
# Some layers, such as Dropout, behave differently during training
my_model_d.train()
n_epochs = 15

for epoch in range(n_epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        # Erase accumulated gradients
        optimizer.zero_grad()

        # Forward pass
        output = my_model_d(data)
```



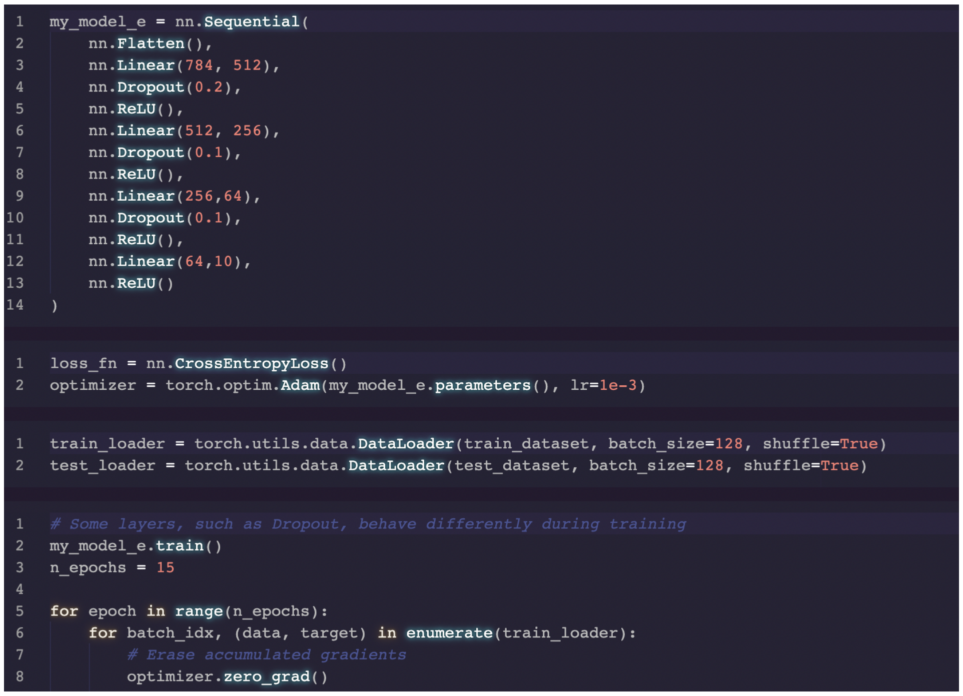
```
print('Test set: Average loss: %.4f, Accuracy: %d/%d (%.4f)' %
        (test_loss, correct, len(test_loader.dataset),
         100. * correct / len(test_loader.dataset)))

Test set: Average loss: 0.0006, Accuracy: 9819/10000 (98.1900)
```

Note that the rest of the loop through n_epochs is the same code as presented in the sample code training loop.

Problem E: Modeling Part 3 [6 EC 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:

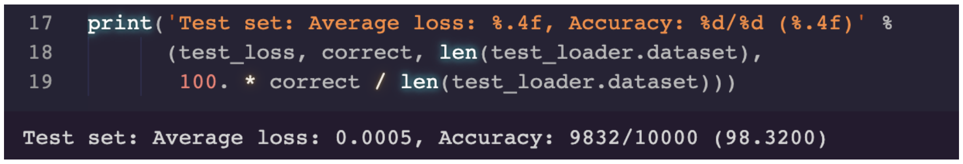


```python
my_model_e = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 512),
    nn.Dropout(0.2),
    nn.ReLU(),
    nn.Linear(512, 256),
    nn.Dropout(0.1),
    nn.ReLU(),
    nn.Linear(256,64),
    nn.Dropout(0.1),
    nn.ReLU(),
    nn.Linear(64,10),
    nn.ReLU()
)
```

```python
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(my_model_e.parameters(), lr=1e-3)
```

```python
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=128, shuffle=True)
```

```python
# Some layers, such as Dropout, behave differently during training
my_model_e.train()
n_epochs = 15

for epoch in range(n_epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        # Erase accumulated gradients
        optimizer.zero_grad()
```



```python
print('Test set: Average loss: %.4f, Accuracy: %d/%d (%.4f)' %
        (test_loss, correct, len(test_loader.dataset),
         100. * correct / len(test_loader.dataset)))
```

```
Test set: Average loss: 0.0005, Accuracy: 9832/10000 (98.3200)
```

Note that the rest of the loop through n_epochs is the same code as presented in the sample code training loop.

Dallas Taylor

3 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 8×8 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:

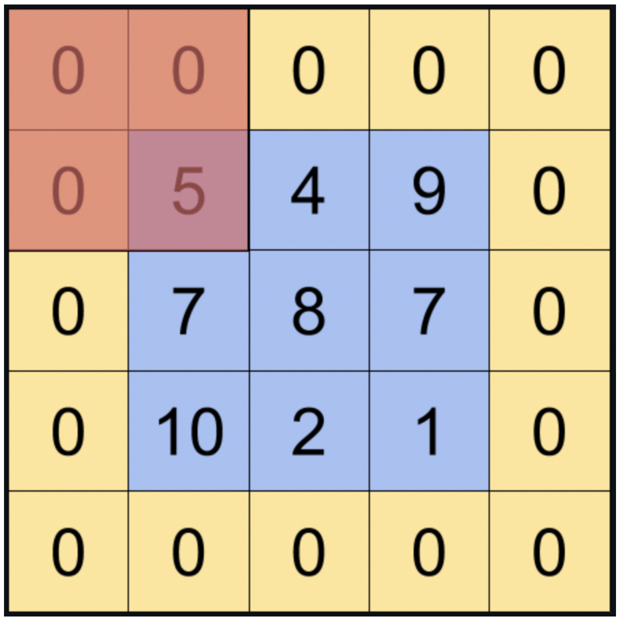


Figure: A convolution being applied to a 2×2 patch (the red square) of a 3×3 image that has been zero-padded to allow convolutions past the edges of the image.

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: *One benefit to this zero-padding scheme is that we are able to produce outputs that have the same dimensions as our input layer. Thus, this would prove beneficial when considering a model that has the requirement of an output of the same size as the input.*

One drawback to this zero-padding scheme is that it obviously causes an increase in the amount of computation required to train our model (and as dimensions increase this cost can become a great burden).

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 (i.e. how much the filter is displaced after each application): 1
- No zero-padding

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

Solution B.: *The number of parameters in this layer is* $(5 \times 5 \times 3 + 1) \times 8 = 608$ *parameters. Note that this is because we learn* 8 *filters of shape* $(5, 5, 3)$ *plus* 1 *for the bias term.*

Colab Link : `https://colab.research.google.com/drive/14hOuwNsMvkmSzAVRGGsvY5hGzZDY2HzH?usp=sharing`

Problem C [3 points]: What is the shape of the output tensor? Remember that convolution is performed over the first two dimensions of the input only, and that a filter is applied to all channels.

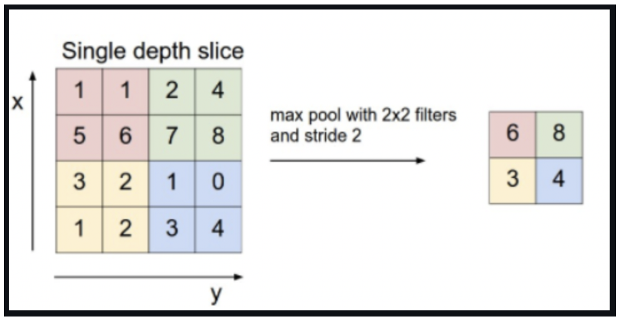
Solution C.: *The shape of the output tensor is* $\{$*(input dimension 0 - (filter dimension 0 - 1)), (input dimension 1 - (filter dimension 1 - 1)), number of filters* $\}$.

Thus, we have that our output tensor is of shape $\{(32 - (5 - 1)), (32 - (5 - 1)), 8\} = \{28, 28, 8\}$.

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):



1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

6	8
3	4

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, where an animal may appear at various angles/locations of the image, and the image contains small amounts of noise (e.g. some pixels may be missing). Why might pooling be advantageous given these properties of our dataset?

Solution F.: *Pooling may be advantageous since we are able to minimize the effect that noisy pixels have on our model's learning (since it can be "hidden" by max-pooling or averaging since we pull generalized information about each unit our pool examines).*

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 (e.g., 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.

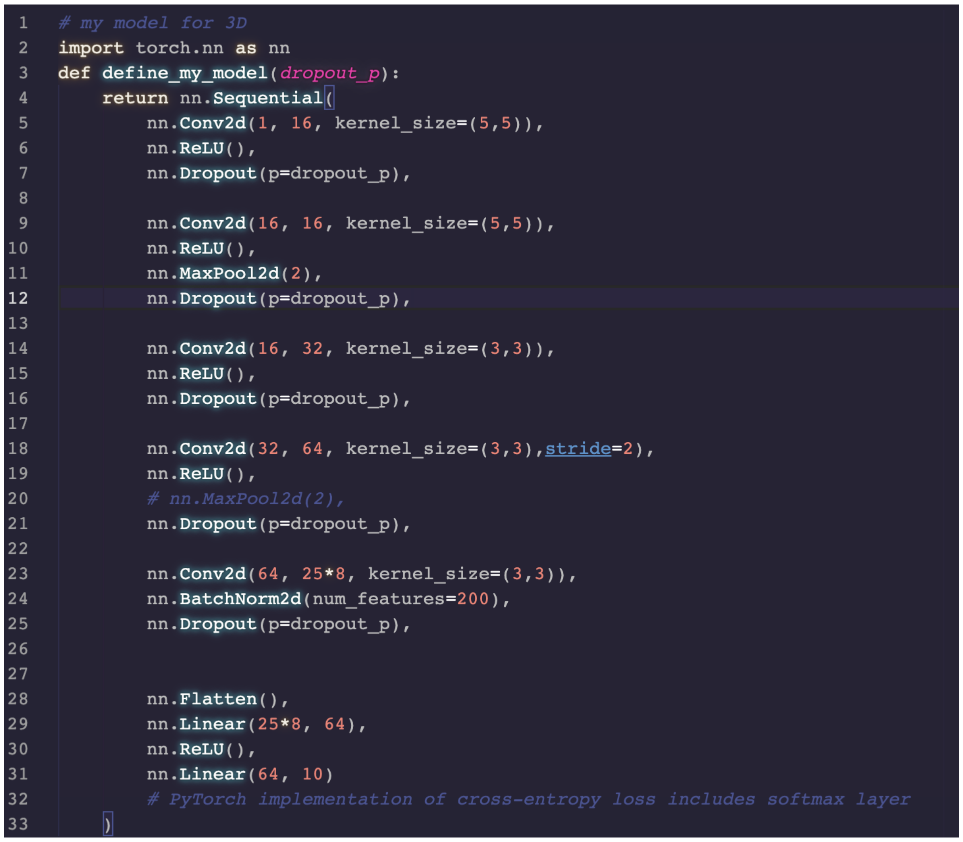
Discuss what you found to be the most effective strategies in designing a convolutional network. Which regularization method was most effective (dropout, layerwise regularization, batch norm)?

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

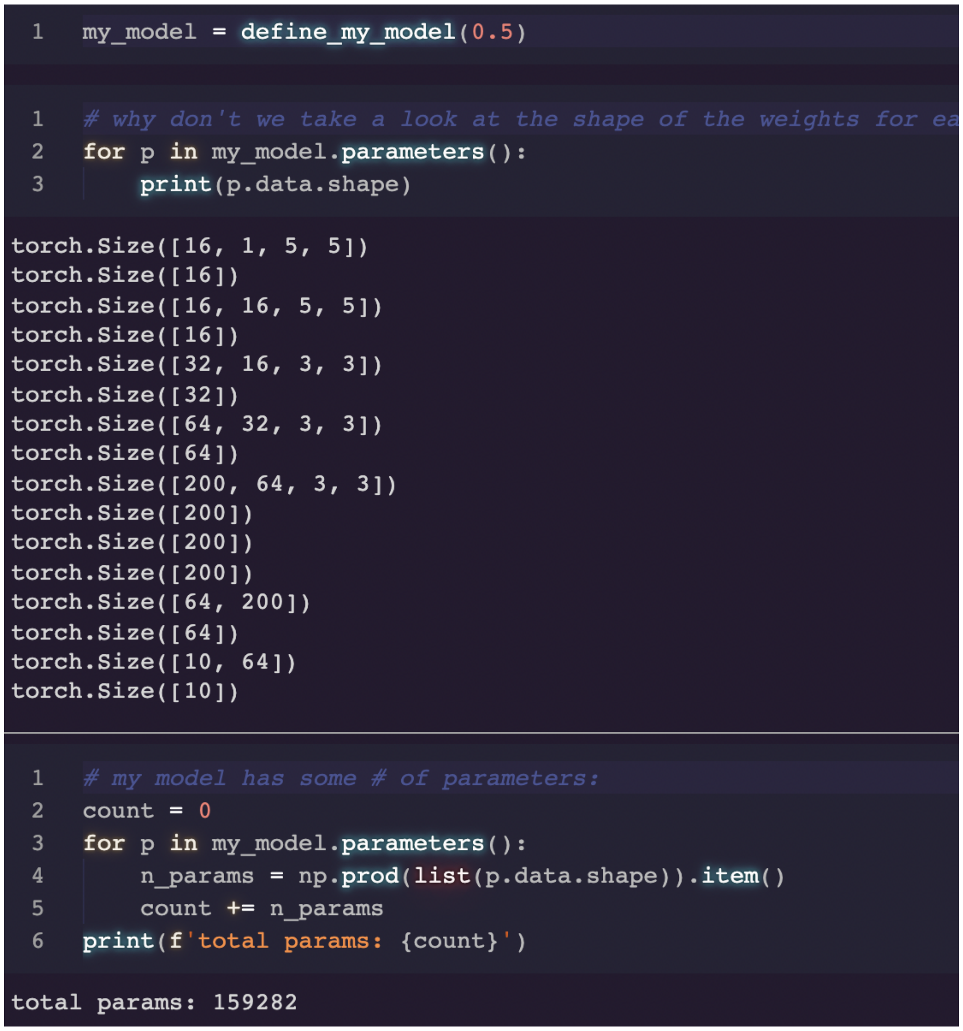
Hints:

- 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 G:



```python
# my model for 3D
import torch.nn as nn
def define_my_model(dropout_p):
    return nn.Sequential(
        nn.Conv2d(1, 16, kernel_size=(5,5)),
        nn.ReLU(),
        nn.Dropout(p=dropout_p),

        nn.Conv2d(16, 16, kernel_size=(5,5)),
        nn.ReLU(),
        nn.MaxPool2d(2),
        nn.Dropout(p=dropout_p),

        nn.Conv2d(16, 32, kernel_size=(3,3)),
        nn.ReLU(),
        nn.Dropout(p=dropout_p),

        nn.Conv2d(32, 64, kernel_size=(3,3),stride=2),
        nn.ReLU(),
        # nn.MaxPool2d(2),
        nn.Dropout(p=dropout_p),

        nn.Conv2d(64, 25*8, kernel_size=(3,3)),
        nn.BatchNorm2d(num_features=200),
        nn.Dropout(p=dropout_p),


        nn.Flatten(),
        nn.Linear(25*8, 64),
        nn.ReLU(),
        nn.Linear(64, 10)
        # PyTorch implementation of cross-entropy loss includes softmax layer
    )
```



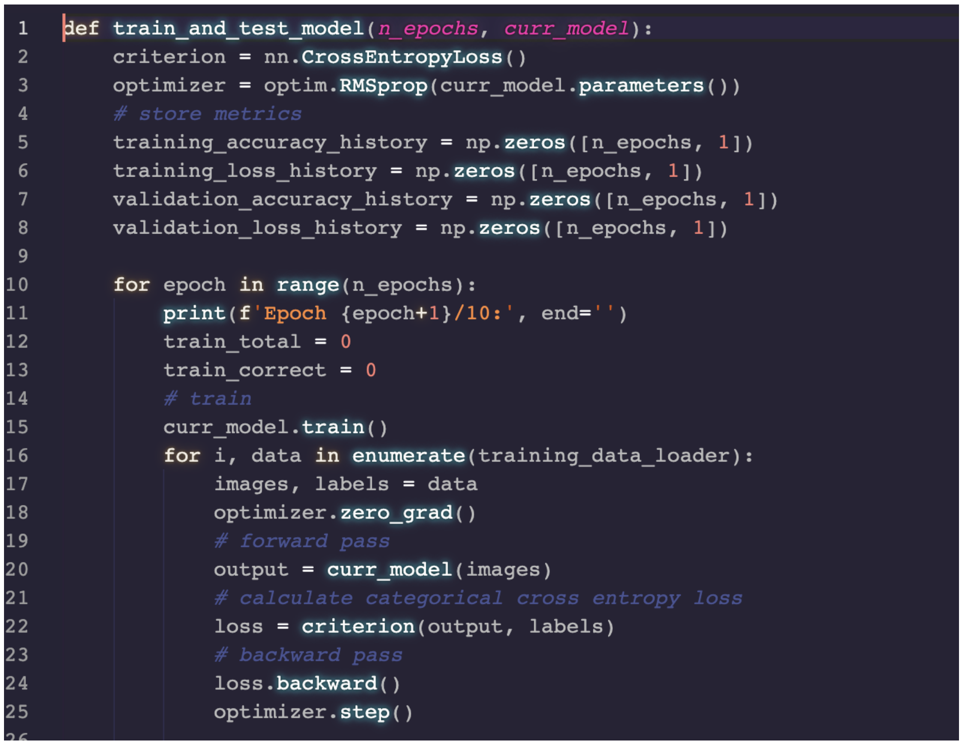
```
my_model = define_my_model(0.5)
```

```
# why don't we take a look at the shape of the weights for ea
for p in my_model.parameters():
    print(p.data.shape)
```

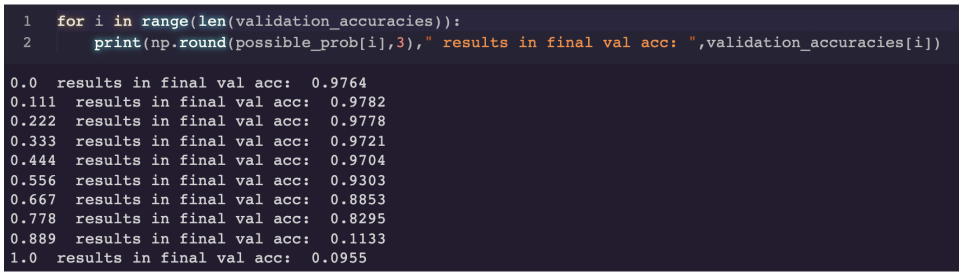
```
torch.Size([16, 1, 5, 5])
torch.Size([16])
torch.Size([16, 16, 5, 5])
torch.Size([16])
torch.Size([32, 16, 3, 3])
torch.Size([32])
torch.Size([64, 32, 3, 3])
torch.Size([64])
torch.Size([200, 64, 3, 3])
torch.Size([200])
torch.Size([200])
torch.Size([200])
torch.Size([64, 200])
torch.Size([64])
torch.Size([10, 64])
torch.Size([10])
```

```
# my model has some # of parameters:
count = 0
for p in my_model.parameters():
    n_params = np.prod(list(p.data.shape)).item()
    count += n_params
print(f'total params: {count}')
```

```
total params: 159282
```



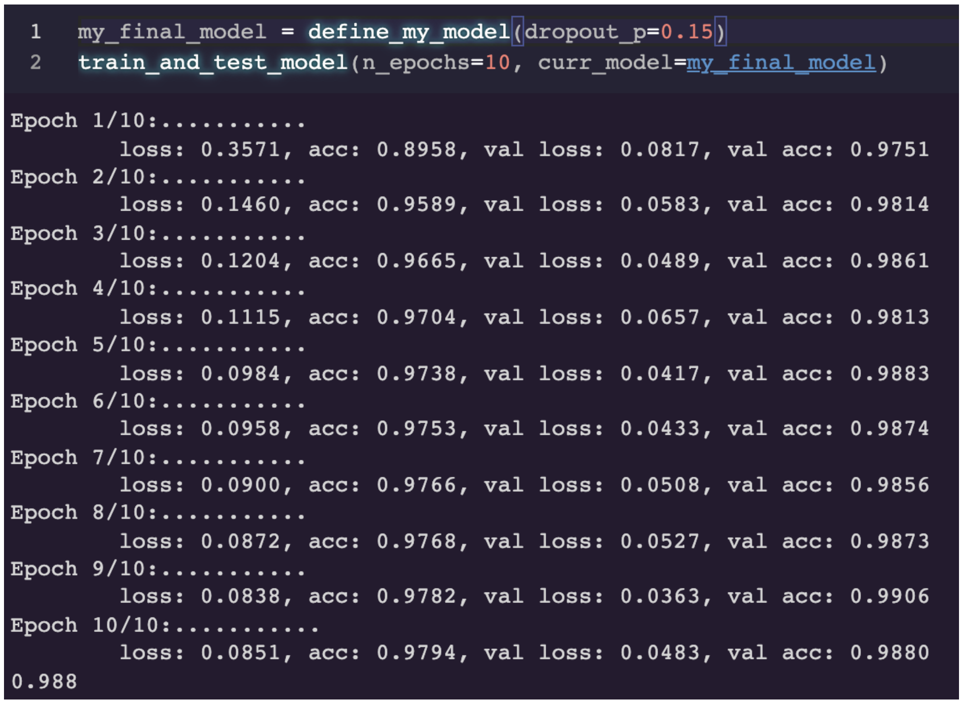
```python
def train_and_test_model(n_epochs, curr_model):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.RMSprop(curr_model.parameters())
    # store metrics
    training_accuracy_history = np.zeros([n_epochs, 1])
    training_loss_history = np.zeros([n_epochs, 1])
    validation_accuracy_history = np.zeros([n_epochs, 1])
    validation_loss_history = np.zeros([n_epochs, 1])

    for epoch in range(n_epochs):
        print(f'Epoch {epoch+1}/10:', end='')
        train_total = 0
        train_correct = 0
        # train
        curr_model.train()
        for i, data in enumerate(training_data_loader):
            images, labels = data
            optimizer.zero_grad()
            # forward pass
            output = curr_model(images)
            # calculate categorical cross entropy loss
            loss = criterion(output, labels)
            # backward pass
            loss.backward()
            optimizer.step()
```



```python
for i in range(len(validation_accuracies)):
    print(np.round(possible_prob[i],3)," results in final val acc: ",validation_accuracies[i])
```

```
0.0  results in final val acc:  0.9764
0.111  results in final val acc:  0.9782
0.222  results in final val acc:  0.9778
0.333  results in final val acc:  0.9721
0.444  results in final val acc:  0.9704
0.556  results in final val acc:  0.9303
0.667  results in final val acc:  0.8853
0.778  results in final val acc:  0.8295
0.889  results in final val acc:  0.1133
1.0  results in final val acc:  0.0955
```

Figure 2: Test Accuracy Across Dropout Probabilities



```
my_final_model = define_my_model(dropout_p=0.15)
train_and_test_model(n_epochs=10, curr_model=my_final_model)

Epoch 1/10:..........
        loss: 0.3571, acc: 0.8958, val loss: 0.0817, val acc: 0.9751
Epoch 2/10:..........
        loss: 0.1460, acc: 0.9589, val loss: 0.0583, val acc: 0.9814
Epoch 3/10:..........
        loss: 0.1204, acc: 0.9665, val loss: 0.0489, val acc: 0.9861
Epoch 4/10:..........
        loss: 0.1115, acc: 0.9704, val loss: 0.0657, val acc: 0.9813
Epoch 5/10:..........
        loss: 0.0984, acc: 0.9738, val loss: 0.0417, val acc: 0.9883
Epoch 6/10:..........
        loss: 0.0958, acc: 0.9753, val loss: 0.0433, val acc: 0.9874
Epoch 7/10:..........
        loss: 0.0900, acc: 0.9766, val loss: 0.0508, val acc: 0.9856
Epoch 8/10:..........
        loss: 0.0872, acc: 0.9768, val loss: 0.0527, val acc: 0.9873
Epoch 9/10:..........
        loss: 0.0838, acc: 0.9782, val loss: 0.0363, val acc: 0.9906
Epoch 10/10:..........
        loss: 0.0851, acc: 0.9794, val loss: 0.0483, val acc: 0.9880
0.988
```

Figure 3: Final Model Accuracy, Using Dropout Prob = 0.15

When designing my CNN, I found that the analyzing different dropout probabilities was very helpful and effective when first building my network, as it allowed me to evaluate the model across different parameters quickly. I also found that adding more convolutional layers also provided an effective strategy to increasing accuracy. When building my network, I discovered that including the BatchNorm layer also proved highly effective in the regularization of my data (with the largest effect on observed accuracy).

Note that for my final model, I utilized a Dropout Probability of 0.15 *since I saw that the model performed best when our Dropout Probability is around* $0.111\ldots$*.*

Additionally note that any code that is not seen is extremely similar to that of the sample code provided and thus I left out these lines for the sake of simplicity in the report.