Machine Learning & Data Mining
Set 4

Caltech CS/CNS/EE 155
January 26 2024.

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 weights for the first network range anywhere between -1 and 1 while the weights for the second network are all initiliazed to 0. For the first network, the test loss decreased to 0.001 by the 250th iteration, indicating a fast convergence. The second network did not show any learning because both the test loss and train loss were much higher (around 0.5) by the 250th iteration. This makes sense when we consider the backpropagation algorithm. When weights are initialized to 0, every attempt to take the gradient results in 0 as well. Thus, no learning occurs.*

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 network appears to converge around the 1000 epoch mark which is much slower when compared to the ReLU activation. This makes sense when we consider the Sigmoid activation. The sigmoid function returns saturating non-linearities with much smaller gradients when compared to ReLU. Thus, learning happens at a slower rate. The second network shows some learning around epoch 3000, but both the test loss and training loss are still very high (around 0.4) relative to the first network. In contrast to ReLU, the Sigmoid function can return non-zero values even when the weights are initialized to zero which allows the network to begin learning. Even so, this process takes a long time relative to the first network.*

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: *The ReLU activation function is defined as ReLU(x) = max$(0, x)$. Thus, we can see that if we trained a fully-connected network with ReLu activations looping through all the negative examples before any of the positive ones, the ReLU would return 0 until reaching the positive examples. In this case, the network weights may update in such a way that the neurons in the network become inactive for all inputs due to only seeing negative examples, effectively "killling" the neuron. Thus, once the network begins training on the positive examples, it will result in a heavy bias. This is known as the "dying ReLu" problem because when a ReLu neuron is dead, it always outputs 0 for any input and has a gradient of 0 almost everywhere. As a result, it does not contribute to the model learning.*

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:

$$\begin{aligned}
\text{OR}(1,0) &\geq 1\\
\text{OR}(0,1) &\geq 1\\
\text{OR}(1,1) &\geq 1\\
\text{OR}(0,0) &= 0
\end{aligned}$$

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.: *A fully connected network with ReLU units that implements the OR function would contain two input features, x_1 and x_2, and their weights would be initialized to 1. After computing the sum and feeding into the ReLU unit, it should result in the OR function. If the sum is ≥ 1, then the network returns 1. Otherwise, 0.*

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 to implement an XOR function is 2. A network with fewer layers cannot handle the XOR problem because it is not linear. Thus, there would need to be at least two separate boundaries to correctly classify the points.*

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:

torch: 2.1.0

torchvision: 0.16.0

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.: *Images are 28 x 28. Each value in the dataset corresponds to the pixel value (darkness on black-/white scale). The length of the training dataset is 60,000 and the length of the testing dataset is 10,000.*

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*.)

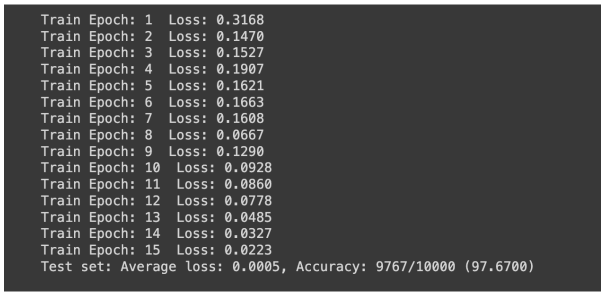
Use categorical cross entropy as your loss function. There are also a number of optimizers you can use (an optimizer is just a fancier version of SGD), and feel free to play around with them, but RMSprop and Adam are the most popular and will probably work best. You also should find the batch size and 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: *mantripragada_ishaan_set4_prob2.ipynb*

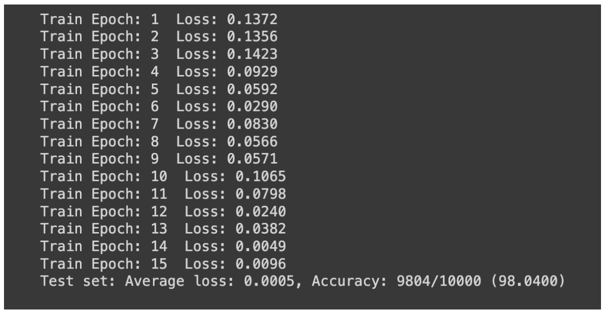


```
Train Epoch: 1  Loss: 0.3168
Train Epoch: 2  Loss: 0.1470
Train Epoch: 3  Loss: 0.1527
Train Epoch: 4  Loss: 0.1907
Train Epoch: 5  Loss: 0.1621
Train Epoch: 6  Loss: 0.1663
Train Epoch: 7  Loss: 0.1608
Train Epoch: 8  Loss: 0.0667
Train Epoch: 9  Loss: 0.1290
Train Epoch: 10  Loss: 0.0928
Train Epoch: 11  Loss: 0.0860
Train Epoch: 12  Loss: 0.0778
Train Epoch: 13  Loss: 0.0485
Train Epoch: 14  Loss: 0.0327
Train Epoch: 15  Loss: 0.0223
Test set: Average loss: 0.0005, Accuracy: 9767/10000 (97.6700)
```

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:

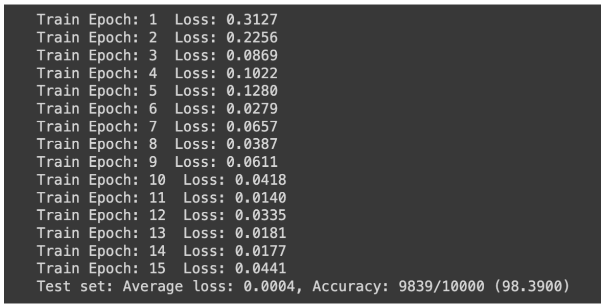


```
Train Epoch: 1  Loss: 0.1372
Train Epoch: 2  Loss: 0.1356
Train Epoch: 3  Loss: 0.1423
Train Epoch: 4  Loss: 0.0929
Train Epoch: 5  Loss: 0.0592
Train Epoch: 6  Loss: 0.0290
Train Epoch: 7  Loss: 0.0830
Train Epoch: 8  Loss: 0.0566
Train Epoch: 9  Loss: 0.0571
Train Epoch: 10  Loss: 0.1065
Train Epoch: 11  Loss: 0.0798
Train Epoch: 12  Loss: 0.0240
Train Epoch: 13  Loss: 0.0382
Train Epoch: 14  Loss: 0.0049
Train Epoch: 15  Loss: 0.0096
Test set: Average loss: 0.0005, Accuracy: 9804/10000 (98.0400)
```

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:



```
Train Epoch: 1  Loss: 0.3127
Train Epoch: 2  Loss: 0.2256
Train Epoch: 3  Loss: 0.0869
Train Epoch: 4  Loss: 0.1022
Train Epoch: 5  Loss: 0.1280
Train Epoch: 6  Loss: 0.0279
Train Epoch: 7  Loss: 0.0657
Train Epoch: 8  Loss: 0.0387
Train Epoch: 9  Loss: 0.0611
Train Epoch: 10  Loss: 0.0418
Train Epoch: 11  Loss: 0.0140
Train Epoch: 12  Loss: 0.0335
Train Epoch: 13  Loss: 0.0181
Train Epoch: 14  Loss: 0.0177
Train Epoch: 15  Loss: 0.0441
Test set: Average loss: 0.0004, Accuracy: 9839/10000 (98.3900)
```

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:

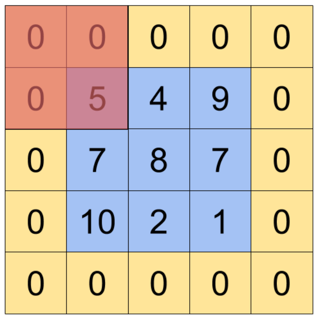


Figure: A convolution being applied to a 2×2 patch (the red square) of a 3×3 image that has been zero-padded to allow convolutions past the edges of the image.

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: *One benefit of zero-padding is that it allows the convolutional filters to be applied to the edge pixels, which means that the spatial dimensions of the output can be maintained after convolution. This is particularly useful in deep CNNs where multiple convolutions are applied, as it prevents the spatial resolution from diminishing too rapidly as you go deeper into the network. One drawback, however, is that by padding with zeros, the convolution may incorporate these zero values into the computation, which can result in border effects where the intensity of the output near the edges might be artificially lower than it would be if the convolution were only applied to the actual image data. This can potentially lead to less accurate detection of features on the borders of the image.*

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.: $(8 \times 5 \times 5 \times 3) + (8 \times 1) = 600 + 8 = 608$

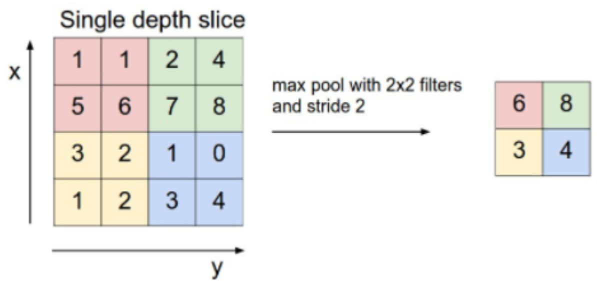
Problem C [3 points]: What is the shape of the output tensor?

Solution C.: $28 \times 28 \times 8$

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):



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.: *Pooling helps the network to achieve spatial invariance to input data, meaning small changes, distortions, or translations in the input image won't affect the output of the pooling layer significantly. This is particularly useful when classifying images of animals that may be in slightly different positions or when some pixels are missing due to noise. Also, by summarizing the presence of features, pooling can sometimes make the network more sensitive to the features that are indeed present, as opposed to being misled by the absence of features due to noise or distortions.*

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?

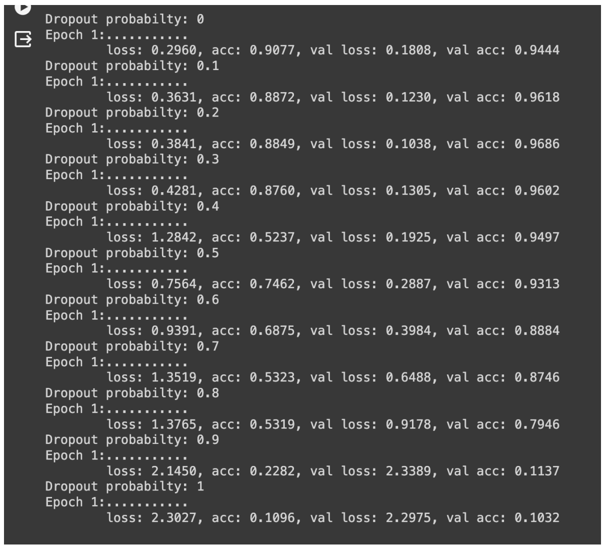
Hints:

- 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 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.

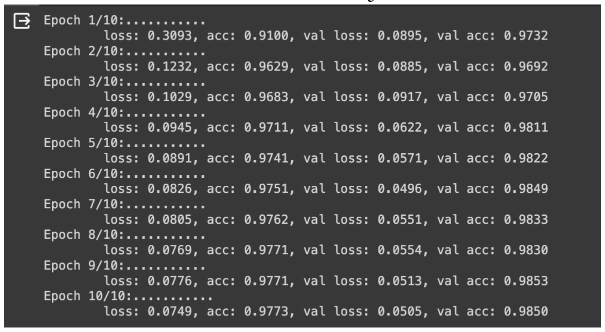
Solution G.: *mantripragada_ishaan_set4_prob3.ipynb*

Dropout Probabilities $p \in [0, 1]$*:*



```
Dropout probabilty: 0
Epoch 1:...........
        loss: 0.2960, acc: 0.9077, val loss: 0.1808, val acc: 0.9444
Dropout probabilty: 0.1
Epoch 1:...........
        loss: 0.3631, acc: 0.8872, val loss: 0.1230, val acc: 0.9618
Dropout probabilty: 0.2
Epoch 1:...........
        loss: 0.3841, acc: 0.8849, val loss: 0.1038, val acc: 0.9686
Dropout probabilty: 0.3
Epoch 1:...........
        loss: 0.4281, acc: 0.8760, val loss: 0.1305, val acc: 0.9602
Dropout probabilty: 0.4
Epoch 1:...........
        loss: 1.2842, acc: 0.5237, val loss: 0.1925, val acc: 0.9497
Dropout probabilty: 0.5
Epoch 1:...........
        loss: 0.7564, acc: 0.7462, val loss: 0.2887, val acc: 0.9313
Dropout probabilty: 0.6
Epoch 1:...........
        loss: 0.9391, acc: 0.6875, val loss: 0.3984, val acc: 0.8884
Dropout probabilty: 0.7
Epoch 1:...........
        loss: 1.3519, acc: 0.5323, val loss: 0.6488, val acc: 0.8746
Dropout probabilty: 0.8
Epoch 1:...........
        loss: 1.3765, acc: 0.5319, val loss: 0.9178, val acc: 0.7946
Dropout probabilty: 0.9
Epoch 1:...........
        loss: 2.1450, acc: 0.2282, val loss: 2.3389, val acc: 0.1137
Dropout probabilty: 1
Epoch 1:...........
        loss: 2.3027, acc: 0.1096, val loss: 2.2975, val acc: 0.1032
```

Final Accuracy:



```
Epoch 1/10:...........
        loss: 0.3093, acc: 0.9100, val loss: 0.0895, val acc: 0.9732
Epoch 2/10:...........
        loss: 0.1232, acc: 0.9629, val loss: 0.0885, val acc: 0.9692
Epoch 3/10:...........
        loss: 0.1029, acc: 0.9683, val loss: 0.0917, val acc: 0.9705
Epoch 4/10:...........
        loss: 0.0945, acc: 0.9711, val loss: 0.0622, val acc: 0.9811
Epoch 5/10:...........
        loss: 0.0891, acc: 0.9741, val loss: 0.0571, val acc: 0.9822
Epoch 6/10:...........
        loss: 0.0826, acc: 0.9751, val loss: 0.0496, val acc: 0.9849
Epoch 7/10:...........
        loss: 0.0805, acc: 0.9762, val loss: 0.0551, val acc: 0.9833
Epoch 8/10:...........
        loss: 0.0769, acc: 0.9771, val loss: 0.0554, val acc: 0.9830
Epoch 9/10:...........
        loss: 0.0776, acc: 0.9771, val loss: 0.0513, val acc: 0.9853
Epoch 10/10:...........
        loss: 0.0749, acc: 0.9773, val loss: 0.0505, val acc: 0.9850
```

When designing my network, I found that a more complex model yielded better results. This makes sense because a more complex model has a higher capacity to learn. Although I did not experience it, an extremely complex model may lead to overfitting. I also found that adding batch norm layers after the 2D convolutional layers was the most effective regularization method. With regards to the dropout probability, a lower dropout led to a better test accuracy, so I kept it at 0.1 for all layers.