Machine Learning & Data Mining
Set 4

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

Policies

- **Due 9 PM, February 2nd**, via Gradescope.
- You are free to collaborate on all of the problems, subject to the collaboration policy stated in the syllabus.
- In this course, we will be using Google Colab for code submissions. You will need a Google account.
- This set uses PyTorch, a Python package for neural networks. We recommend using Google Colab, which comes with PyTorch already installed. There will be a PyTorch recitation to help you get started.

Submission Instructions

- Submit your report as a single .pdf file to Gradescope (entry code 7426YK), under "Set 4 Report".
- 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.
- For instructions specifically pertaining to the Gradescope submission process, see https://www.gradescope.com/get_started#student-submission.

Google Colab Instructions

For each notebook, you need to save a copy to your drive.

1. Open the github preview of the notebook, and click the icon to open the colab preview.
2. On the colab preview, go to File → Save a copy in Drive.
3. Edit your file name to "lastname_firstname_set_problem", e.g."yue_yisong_set4_prob2.ipynb"

TA Office Hours

- **Megan Tjandrasuwita [Question 1 and Question 2]**
  - Monday 1/31: 3:00 pm - 4:00 pm (Annenberg North Lawn Tent), Tuesday, 2/1: 5:00 pm - 6:00 pm (Zoom)
- **Pantelis Vafidis [Question 3]**
  - Monday, 1/31: 6:00 pm - 8:00 pm (Zoom)

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.: *For a two layer neural network, we can solve the gradient $\nabla_w \mathcal{L}$ for each layer:*

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} = \frac{\partial L}{\partial x^{(2)}} \frac{\partial x^{(2)}}{\partial s^{(2)}} \frac{\partial s^{(2)}}{\partial x^{(1)}} \frac{\partial x^{(1)}}{\partial s^{(1)}} \frac{\partial s^{(1)}}{\partial W^{(1)}}$$
$$\frac{\partial \mathcal{L}}{\partial W^{(2)}} = \frac{\partial L}{\partial x^{(2)}} \frac{\partial x^{(2)}}{\partial s^{(2)}} \frac{\partial s^{(2)}}{\partial W^{(2)}}$$

We know for the ReLU activation function that $\frac{\partial x^{(\ell)}}{\partial s^{(\ell)}}$ goes to zero for $s^{(\ell)} \to -\infty$. That said, the main difference between the two neural networks are initial layer weights. Namely, the second network is initialized with all weights equal to zero. As such the outputs vanish because they were initialized in the region of the ReLU function that saturates the outputs, i.e. the derivatives are always equal to a small number. Because of this, the loss change is very small and the weights are barely updated.

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.: *For the network that was initialized with non-zero weights, the network took longer to converge on a solution when using the sigmoid activation function than the one that used the ReLU function. This is primarily due to the fact that the sigmoid function is saturating for both $s^{(\ell)} \to +\infty$ and $s^{(\ell)} \to -\infty$, which means the gradients and corresponding outputs can vanish at both extremes. Additionally, the decision boundary of the network using the sigmoid function is much "smoother" compared to the network using ReLU function, and you can see the relative contribution of each neuron has a less defined boundary (the ReLU network makes a distinct 6 sided boundary around the blue points in the middle whereas the sigmoid network is more smooth). In*

the end, both of the networks arrive at about same test and training error.
For the second network with zero weights, it actually benefits from performing more iterations and using the sigmoid function: in the case of using the ReLU function for 250 iterations, the weights are all still zero and there is no decision boundary at all (errors are still about 0.5), but for the sigmoid function for 4000 iterations the weights are non-zero and the decision boundary is defined as a straight line and errors are closer to 0.4.

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: *When using the ReLU function, the gradient is zero for negative points (that is, when $s^{(\ell)} \leq 0$). As such, by only training on negative points, you are biasing your model to learn where there are no gradients, with very little to not updates of your loss function and weights. In fact, your network will only learn the negative weights, and may be biased to the negative side of the ReLU function so deeply that it might not recover, such that the outputs and network "dies."*

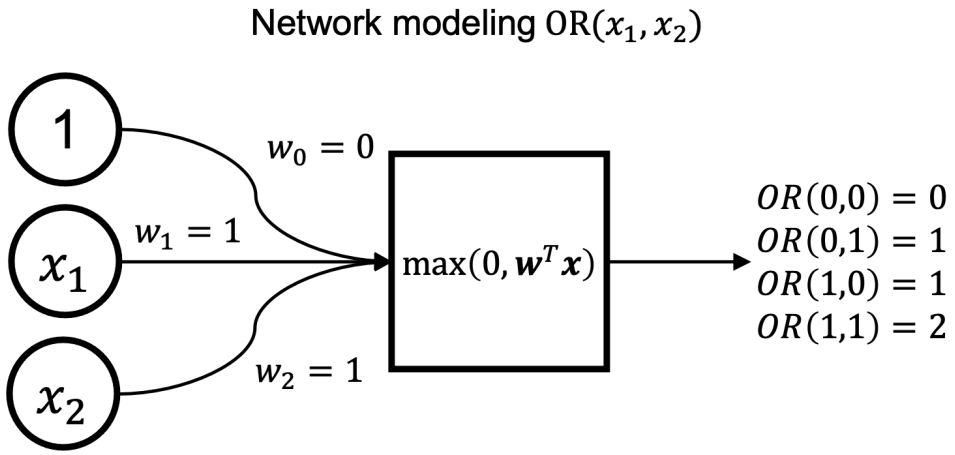
Problem D: Approximating Functions Part 1 **[7 Points]**

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

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

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

Solution D:



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

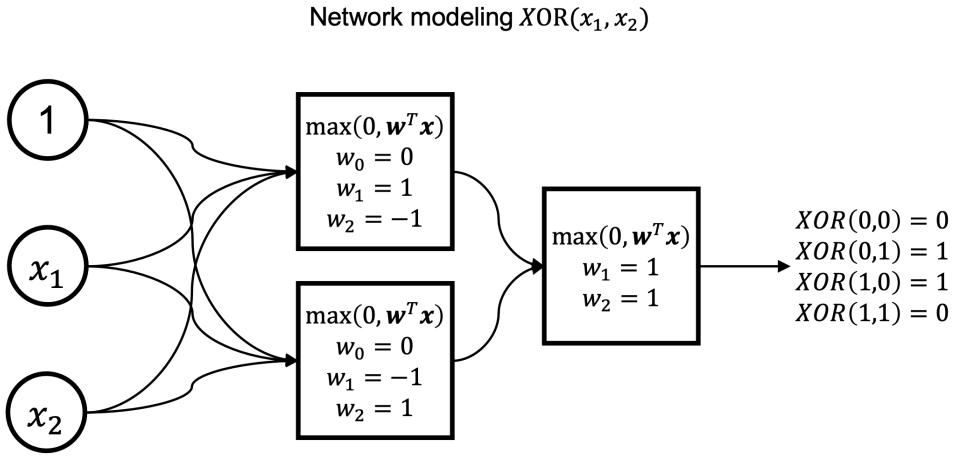
What is the minimum number of fully-connected layers (with ReLU units) needed to implement an XOR of two 0/1-valued inputs x_1, x_2? Recall that the XOR function is defined as:

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

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

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

Solution E: *The minimum number of fully-connected layers is two:*



Logically, to write a network that approximates the XOR function, you need to satisfy two conditions: give a value greater than or equal to 1 when $x_1 \neq x_2$ and give a value less than or equal to zero when $x_1 = x_2$. This is satisfied by the general equation $a - b$, and since x_1 and x_2 are distinct and cannot permute, we need to approximate two equations to satisfy the conditions (i.e. $x_1 - x_2$ and $x_2 - x_1$). We must then take the sum of these two functions, as they give "opposite" outputs, and as such that second layer is required for the summation of the two functions.

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

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: *Each image is 28x28 pixels. The first index selects a tuple of size 2 that contains the the input tensor (i.e. the image) and the label of the target number that the image is supposed to replicate. As such, the second index selects whether to return the torch.tensor object or the target number label. The torch.tensor object itself has 3 dimensions representing the color channel, the height channel, and the width channel (CxHxW). In this case, there is only one color channel. There are 60,000 images in the training set and 10,000 images in the test 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: *Link to Google Colab notebook for problem 2.*
The best test accuracy achieved was 0.9786.

```
Epoch 1/10: loss: 0.2904, acc: 0.9138, val loss: 0.00009134, val acc: 0.9596
Epoch 2/10: loss: 0.1696, acc: 0.9494, val loss: 0.00040934, val acc: 0.9700
Epoch 3/10: loss: 0.1432, acc: 0.9576, val loss: 0.00018152, val acc: 0.9719
Epoch 4/10: loss: 0.1287, acc: 0.9609, val loss: 0.00002176, val acc: 0.9730
Epoch 5/10: loss: 0.1214, acc: 0.9633, val loss: 0.00004694, val acc: 0.9744
Epoch 6/10: loss: 0.0831, acc: 0.9737, val loss: 0.00000891, val acc: 0.9774
Epoch 7/10: loss: 0.0789, acc: 0.9762, val loss: 0.00000283, val acc: 0.9772
Epoch 8/10: loss: 0.0733, acc: 0.9775, val loss: 0.00000779, val acc: 0.9775
Epoch 9/10: loss: 0.0716, acc: 0.9775, val loss: 0.00000802, val acc: 0.9786
Epoch 10/10: loss: 0.0681, acc: 0.9789, val loss: 0.00480728, val acc: 0.9776
```

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: *The best test accuracy achieved was 0.9810.*

```
Epoch 1/10: loss: 0.4155, acc: 0.8739, val loss: 0.00078923, val acc: 0.9550
Epoch 2/10: loss: 0.1709, acc: 0.9507, val loss: 0.00085497, val acc: 0.9660
Epoch 3/10: loss: 0.1253, acc: 0.9634, val loss: 0.00000399, val acc: 0.9729
Epoch 4/10: loss: 0.1018, acc: 0.9701, val loss: 0.00001178, val acc: 0.9758
Epoch 5/10: loss: 0.0876, acc: 0.9748, val loss: 0.00009233, val acc: 0.9764
Epoch 6/10: loss: 0.0607, acc: 0.9821, val loss: 0.00092222, val acc: 0.9810
Epoch 7/10: loss: 0.0527, acc: 0.9850, val loss: 0.00016449, val acc: 0.9805
Epoch 8/10: loss: 0.0494, acc: 0.9852, val loss: 0.00000648, val acc: 0.9805
Epoch 9/10: loss: 0.0468, acc: 0.9863, val loss: 0.00087275, val acc: 0.9807
Epoch 10/10: loss: 0.0428, acc: 0.9869, val loss: 0.00000766, val acc: 0.9809
```

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: *The best test accuracy achieved was 0.9854.*

```
Epoch 1/10: loss: 0.2575, acc: 0.9206, val loss: 0.00002250, val acc: 0.9595
Epoch 2/10: loss: 0.1017, acc: 0.9689, val loss: 0.00013258, val acc: 0.9718
Epoch 3/10: loss: 0.0727, acc: 0.9779, val loss: 0.00015880, val acc: 0.9757
Epoch 4/10: loss: 0.0569, acc: 0.9827, val loss: 0.00001425, val acc: 0.9785
Epoch 5/10: loss: 0.0434, acc: 0.9864, val loss: 0.00011532, val acc: 0.9729
Epoch 6/10: loss: 0.0164, acc: 0.9957, val loss: 0.00000559, val acc: 0.9851
Epoch 7/10: loss: 0.0087, acc: 0.9977, val loss: 0.00002169, val acc: 0.9848
Epoch 8/10: loss: 0.0061, acc: 0.9983, val loss: 0.00000074, val acc: 0.9849
Epoch 9/10: loss: 0.0040, acc: 0.9988, val loss: 0.00000006, val acc: 0.9854
Epoch 10/10: loss: 0.0026, acc: 0.9994, val loss: 0.00000006, val acc: 0.9854
```

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:

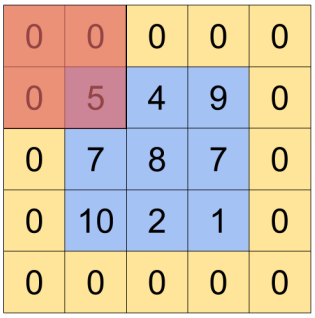


Figure: A convolution being applied to a 2×2 patch (the red square) of a 3×3 image that has been zero-padded to allow convolutions past the edges of the image.

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 of padding is that you can maintain the same size as the actual image as you perform your convolutions and build your layers, which means you can build more layers in your network and presumably some of the network building is trivialized because you can theoretically maintain the same size features throughout your network. However, introducing padding can also have drawbacks if you decide to use a large filter relative to the size of your image. In this case, the padding values (i.e. 0) may be disproportionately represented in the resulting convolved image if the filter size is large, which means you may be losing information and introducing noise into your model.*

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.: *Each filter has* $5 \times 5 \times 3$ *parameters plus 1 bias term for each filter, or 76 total parameters per filter. Since there are 8 filters, we have 608 total parameters for this layer.*

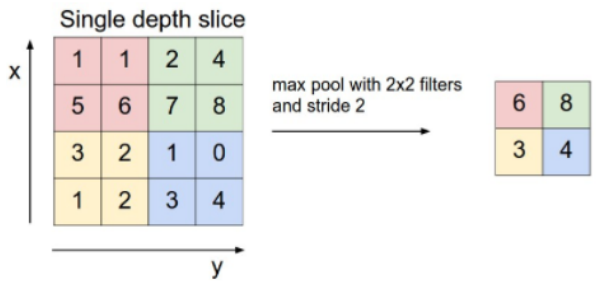
Problem C [3 points]: What is the shape of the output tensor? Remember that convolution is performed over the first two dimensions of the input only, and that a filter is applied to all channels.

Solution C.: *The shape of the output tensor is just the shape of the resulting convolution (in 2D you can just say a* 5×5 *filter on a* 32×32 *image), or* 28×28*, with each filter adding an extra dimension. So the final shape of the output tensor is* $28 \times 28 \times 8$*.*

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, 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.: *It uses information from neighboring pixels to fill in the gaps or eliminate the noise. Put another way, you use the surrounding pixels to take a "vote" of what is the most likely pixel in the filter window, which is especially important if data is missing or noisy.*

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?

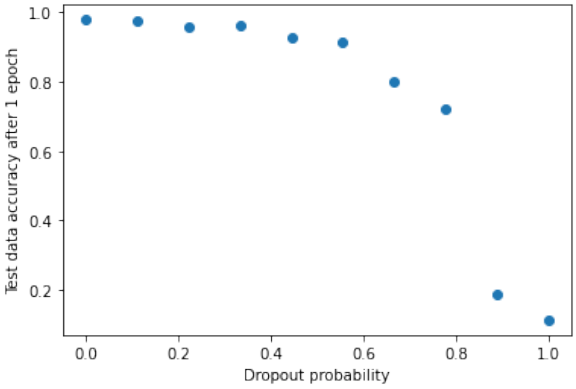
Hints:

- 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 probabil-

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

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

Solution G: *Link to Google Colab notebook for problem 3.*




```
Epoch 1/10:..........
        loss: 0.1959, acc: 0.9420, val loss: 0.0675, val acc: 0.9782
Epoch 2/10:..........
        loss: 0.0928, acc: 0.9718, val loss: 0.0611, val acc: 0.9818
Epoch 3/10:..........
        loss: 0.0772, acc: 0.9768, val loss: 0.0445, val acc: 0.9864
Epoch 4/10:..........
        loss: 0.0669, acc: 0.9791, val loss: 0.0351, val acc: 0.9888
Epoch 5/10:..........
        loss: 0.0599, acc: 0.9815, val loss: 0.0364, val acc: 0.9886
Epoch 6/10:..........
        loss: 0.0443, acc: 0.9866, val loss: 0.0292, val acc: 0.9921
Epoch 7/10:..........
        loss: 0.0436, acc: 0.9869, val loss: 0.0282, val acc: 0.9916
Epoch 8/10:..........
        loss: 0.0434, acc: 0.9871, val loss: 0.0270, val acc: 0.9920
Epoch 9/10:..........
        loss: 0.0420, acc: 0.9876, val loss: 0.0263, val acc: 0.9926
Epoch 10/10:..........
        loss: 0.0409, acc: 0.9872, val loss: 0.0261, val acc: 0.9925
```

The final accuracy of my model when trained for 10 epochs was 0.9925

When designing my network, I found several key points that were important. First, the number of convolutional layers was important for extracting features, and my model improved when I added more. Second, using a 3×3 *kernel size was sufficient to extract features, and we didn't need to use more. Additionally, my model benefited from several forms of regularization, as I used batch normalization and dropout at every convolutional layer, and also included L2 regularization in my loss function. Empirically, it seemed that L2 regularization actually got me a long way compared to other forms of regularization, and as my hyperparameter tuning exercise for dropout probability shows, my model doesn't actually require too much dropout because of the other regularization strategies that I am implementing. For the most part, it seemed the batch normalization an the L2 regularization seemed to be the most effective at improving test accuracy.*

One potential problem with validating hyperparameters in the method that we used is that one epoch is not representative of the entire model, and relies heavily on stochastic/random processes (e.g. initialization and dropout). As such, it may be more necesseary to complete a small number of epochs or to repeat this exercise to ensure that the hyperparameter choice is robust. However, this comes at the cost of computational time, and in general the more hyperparameters you try to tune the longer it will take to find the ideal model parameters. The ideal scenario, however, would be to make a model that is insulated from hyperparameter choice, such it performs well at a wide range of hyperparameters.

Other notes about designing a convolutional network: limiting the number of parameters in the final fully-connected linear layer(s) is important to reduce network runtime. As such, using pooling is useful for dimensionality reduction, while still maintaining information. Furthermore, the number of filters is also important for maintaining a reasonable runtime, as that increases the number of weights to be calculated in that final layer. Additionally, one feature of both the DNN and CNN models that I implemented is changing the learning rate of the optimizer about half-way through the the SGD. I found that this was very important to consistently getting accuracies that were under the cutoffs.