Policies

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

Submission Instructions

- Please submit your report as a single .pdf file to Gradescope (entry code 9YJGEX), under "Problem Set 4". **In the report, include any images generated by your code along with your answers to the questions.** For instructions specifically pertaining to the Gradescope submission process, see https://www.gradescope.com/get_started#student-submission.
- Since we are only using Gradescope this year, we ask that you upload both your code and solution pdf together as a single pdf and mark each page with the problem number. There are various free online tools to combine pdfs. To download your code as a pdf from Jupyter notebook, you can navigate to File -> Download as -> "PDF via Latex" or alternatively download the Latex file and compile it yourself.

TA Office Hours

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

1 Deep Learning Principles [35 Points]

Relevant materials: lectures on deep learning

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

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

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

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

Solution A.:

During backpropagation, the gradient $\frac{\delta \mathcal{L}}{\delta W^\ell}$ is proportional to $\frac{\delta S^{\ell+1}}{\delta X^\ell} = W^{\ell+1}$ for hidden layers $\ell = 1, 2, ...L-1$ (all layers besides the output layer). Thus, if the weights of a neural network are initialized to zero, the initial gradients for all weights besides those entering the output layer will also be zero.

We now analyze the gradients for the weights W^L entering the output layer. Note that the ReLU activation function outputs 0 when its input is 0; we have $ReLU(0) = \max(0, 0) = 0$. We also know that $\frac{\delta \mathcal{L}}{\delta W^L}$ is proportional to $\frac{\delta S^L}{\delta W^L} = X^{L-1}$. When all the weights are initialized to 0, the input signal S^ℓ to the ReLU units is 0, so $X^{\ell-1} = 0$ for $\ell = 1, 2, ...L$. Thus, the gradient for W^L is also zero ($\frac{\delta \mathcal{L}}{\delta W^L} = 0$). Since all weights in the network have a gradient of zero, learning via gradient descent is not possible.

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.: *Sigmoid units have non-zero output when their input is zero, unlike ReLU units. Thus, $X^{L-1} \neq 0$ during the first iteration of gradient descent, allowing for a non-zero initial update to W^L. This allows for non-zero updates in preceding layers of the network during successive iterations of gradient descent, enabling learning.*

Using non-zero initial weights speeds convergence. We know $\frac{\delta \mathcal{L}}{\delta W^\ell}$ is proportional to $W^{\ell+1}$ for all layers besides the output layer. When the weights W^ℓ are initialized to zero, the multiplicative effect of this relationship makes initial updates to the network's first layers very small. Initializing with non-zero weights allows for larger

updates in the network's initial layers early on during training, hastening convergence.

Problem C: [10 Points]

When training any model using SGD, it's important to shuffle your data to avoid correlated samples. To illustrate one reason why this is particularly important for ReLU networks (i.e. it has ReLU activation functions between its hidden layers, but its output layer is still softmax/tanh/linear), consider a dataset of 1000 points, 500 of which have positive (+1) labels, and 500 of which have negative (-1) labels. What happens if we train a fully-connected network with ReLU activations using SGD, looping through all the negative examples before any of the positive examples? Do not assume that the derivative of ReLU at 0 is implemented as 0. *Hint: this is called the "dying ReLU" problem, although it is possible with other activation functions.*

Solution C: *The ReLU units "die" – their incoming weights are updated towards zero and become negative. For $S < 0$, the gradient of the rectifier function ReLU$(S) = \max(0, S)$ is 0, so the weights entering the ReLU units stop changing and the units simply output* 0.

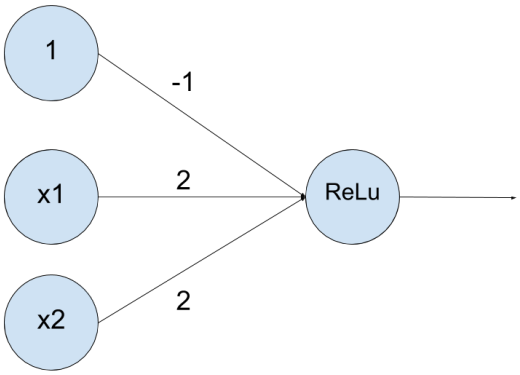
Problem D: Approximating Functions Part 1 **[7 Points]**

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.: *A network consisting of a single hidden layer can approximate the OR function, as shown below:*



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:

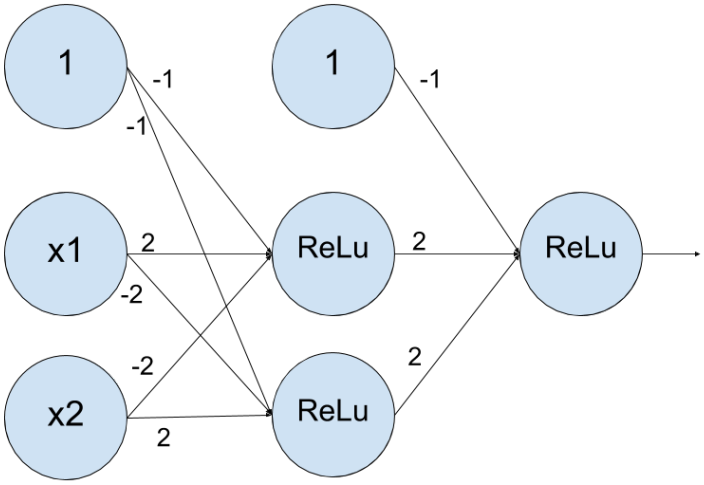
$$\text{XOR}(1,0) \geq 1$$
$$\text{XOR}(0,1) \geq 1$$
$$\text{XOR}(0,0) = \text{XOR}(1,1) = 0$$

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.: *Note that the output classes $\{0,1\}$ of XOR are not linearly separable with respect to x_1, x_2; in (x_1, x_2) coordinates, the points $(0,0), (1,1)$ (corresponding to an output of 0) and $(1,0), (1,1)$ (corresponding to an output of 1) are not linearly separable. Thus, we can't compute XOR using a single fully-connected ReLU layer.*

However, we can implement XOR using two fully connected layers, as shown below:



The first layer consists of two units; one that computes x_1 AND $(NOTx_2)$ and one that computes x_2 AND $(NOTx_1)$. The second (output) layer computes the OR of the outputs of the first layer.

2 Inside a Neural Network [17 Points]

Relevant Materials: Lectures on Deep Learning

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

Problem A: Installation [2 Points]

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

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

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

Solution A: *Should be some version numbers for both torch and torchvision. For example:*

torch: '1.4.0'

torchvision: '0.5.0'

Submission instructions:

For parts 2B-2D, submit one notebook. In your notebook, include the code you used to preprocess your data and train your model. Make sure your results are visible, including loss in each training epoch as well as testing loss and accuracy.

Problem B: The Data [5 Points]

Load and preprocess the tabular dataset, "COVID-19_Case_Surveillance_Public_Use_Data_Subset.csv." This is a small subset of the CDC's Covid-19 Case Surveillance data (https://data.cdc.gov/Case-Surveillance/COVID-19-Case-Surveillance-Public-Use-Data/vbim-akqf). Limit the number of input variables in your final dataset to be between 7 and 12, and use the "death_yn" column as the dependent variable.

The following are some considerations and tips for data preprocessing.

1. **The Dependent Variable:** You will notice that the dependent variable is sometimes missing or unusable. You must drop all the rows with missing/nan death_yn values: since that is the value the model is learning, having rows without a binary classification for it is not helpful in this case.

2. **Missing Values:** Other missing values cannot be input to a model without modification of some kind. Options to deal with them include dropping all rows with any missing values, dropping columns with a large proportion of missing values, various types of imputation, or making them into their own category when you factorize or one-hot.

3. **Categorical Variables:** Some columns are, or contain, string variables. Neural networks can only take numbers as input, so these columns must be modified. Options for this include factorizing, and one-hot encoding. Factorizing is the process of assigning each category an integer (see https://pandas.pydata.org/docs/reference/api/pandas.factorize.html), and one-hot encoding is making each category into its own binary column (see https://www.kaggle.com/code/dansbecker/using-categorical-data-with-one-hot-encoding).

4. **Normalization/Standardization:** This is important for neural networks. Columns that have very different magnitudes of data tend to throw off neural nets, so if you end up with cases like that, you should normalize (make the column have mean 0 and standard deviation of 1) or standardize (make all values range from 0 to 1). You might find sklearn's StandardScaler helpful for this https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.

5. **Data Imbalance:** You will notice that there are many more 0's in the death_yn column than ones. This will induce the model to learn to predict only zeros. The simplest way to make the model learn meaningfully is to undersample from the zero cases. You can make the ratio less extreme, or even balance them 1:1. In most cases, pruning will cause accuracy to go down (which can sometimes be remedied). Even so, it is extremely important to do so, because in real cases, it is almost never acceptable to train a model to predict only one class.

6. **Date Columns:** Fundamentally, we need a way to express date strings as a number. You could choose to do something like only keeping the year and converting to numeric form, or you could keep more fine grain detail by converting to a datetime object and then converting that to numeric. You could also drop if you think these columns are not useful.

7. **Correlated Columns:** You can drop highly correlated columns to get rid of this issue, especially if the data are expressing very similar things. Neural nets tend to not suffer from correlated data as much as other algorithms, however, making this not strictly necessary.

8. **Summary:** Since neural nets cannot take any input except numbers, all of the columns you intend to use need to go from something strange to plain numbers. Use as much of the above information as you need in order to make that happen. Lastly, **DO NOT** try to use Python base code to process dataframes. Use pandas functions, PyTorch utilities, numpy functions, or other library methods built to work with large amounts of data (even though this dataset is comparatively small, it is good practice to use packages for this kind of thing).

Keeping in mind the above, explain your preprocessing decisions.

Solution B:

Mention that neural nets can't take strings or categorical variables. Thus, explain process of factorizing or one-hot encoding. Mention any pruning and why. Mention any imputation and why. Mention normalizing/standardizing numerical data.

Problem C: Linear Neural Network [5 Points]

Now, use PyTorch's "Sequential" class to make a neural network with one hidden linear layer of size 5 and a binary output layer. Do not use any activation function in your linear layer, and do not use any other layer types. Train and test your model on your dataset from part B.

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

Solution C:

Should have a 5xinput_dim heatmap attached.

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

Create and train a 2-layer linear neural network and assess its performance on your dataset. It is expected that the 1-layer and 2-layer models have similar losses–why should this be the case? Is this observable in your code?

Solution D:

2 linear layers with no activation functions is still a linear function, and thus has no more modelling capacity than single layer.

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

Relevant Materials: Lectures on Deep Learning

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

Problem A: The Data [3 Points]

Load the MNIST dataset using torchvision; see the problem 3 sample code for how.

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

Solution A.: *Initial data: each image is 28x28, with pixel values ranging from 0-255 (larger numbers represent a darker shade). There are 60000 training samples and 10000 testing samples.*

Model submission instructions:

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

Problem B: Modeling Part 1 [8 Points]

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

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

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

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

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

Your task. Using at most 100 hidden units, build a network using only the allowed layers that achieves test accuracy of at least 0.975.

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

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

Solution B:

```
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 100),
    nn.ReLU(),
    nn.Linear(100, 10)
)
Test accuracy: 0.9783
```

Problem C: Modeling Part 2 [6 Points]

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

Solution C:

```
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 150),
```

```
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(150, 50),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(50, 10)
)
Test accuracy: 0.981
```

Problem D: Modeling Part 3 [6 Points]

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

Solution D:

```
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 700),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(700, 200),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(200, 100),
    nn.ReLU(),
    nn.Dropout(0.1),
    nn.Linear(100, 10)
)
Test accuracy: 0.9843
```

4 Convolutional Neural Networks [40 Points]

Relevant Materials: Lecture on CNNs

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

Consider a convolutional network in which we perform a convolution over each 8×8 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:

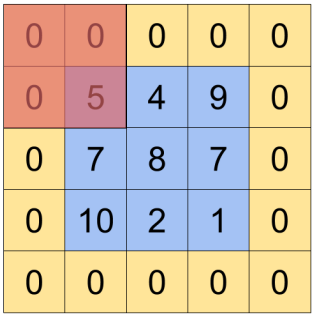


Figure: A convolution being applied to a 2×2 patch (the red square) of a 3×3 image that has been zero-padded to allow convolutions past the edges of the image.

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: *Let's consider the case in which we do not zero-pad our images - since pixels on the border of an image belong to fewer 8×8 image patches than those near the image center, they influence fewer of the convolutional layer's outputs and are therefore underrepresented.*

Zero-padding addresses this issue, resulting in new 8×8 image patches consisting of border pixels and padding pixels. However, since the padding pixels are zero-valued, the border pixels may overinfluence convolutions performed over these new image patches.

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? What is the shape of the output tensor?

Solution B.:

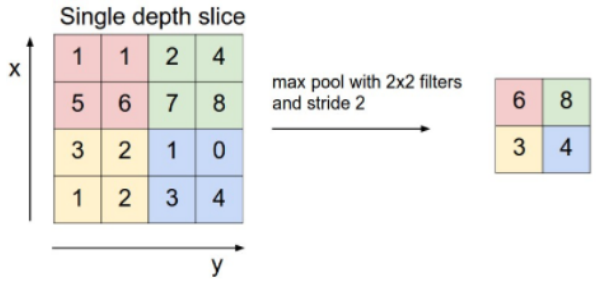
There are 8 x 5 x 5 x 3 + 8 = 608 weights. This is because for each input in each 5 x 5 x 3 window, we want to create 8 features. Also each of the 8 filters has a bias term.

Since there is no zero padding, the sliding window goes over the image (32 - 4) (32 - 4) times. For each filter a vector of 8 features is created. Thus the output has shape 28 x 28 x 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 C [3 points]:

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

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

Solution C.:

Average pooling:

$$\begin{bmatrix} 1 & .5 \\ .5 & .25 \end{bmatrix}, \begin{bmatrix} .5 & 1 \\ .25 & .5 \end{bmatrix}, \begin{bmatrix} .25 & .5 \\ .5 & 1 \end{bmatrix}, \begin{bmatrix} .5 & .25 \\ 1 & .5 \end{bmatrix}$$

Max pooling:

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

for all 4 matrices.

Problem D [4 points]:

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

Solution D.:

Pooling would be advantageous because 1) it will remove the noise such as missing pixels and 2) it will provide translational invariance which will allow for the model to work just as well whether the picture of the animal is in the top left or middle of the image.

Hyperparameter Tuning

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

1. **"Babysitting":** When tuning a model, you can manually choose a set of hyperparameters, monitor in- and out-of-sample performance, adjust according to intuition, and train again.
2. **Grid search:** For each hyperparameter (learning rate, dropout probability, etc), you define a *range* of values to search, and an *interval* at which to sample points. For example, you could try learning rates lr= $\{10^{-1}, 10^{-2}, \ldots, 10^{-5}\}$ and dropout probabilities $p = \{0, 0.1, \ldots, 0.5\}$, and then train your model for each (lr, p) pair, and keep the setting that yields best results.
3. **Random search:** You can again define a search range, but instead of testing points at pre-determined intervals, you could sample points at random and train your model with those hyperparameters. You could take it one step further and implement a "coarse-to-fine" approach: i.e. define a large search space, sample and test a few hyperparameters, and then repeat the process inside a finer search space around those hyperparameters that yielded best results in the first pass.

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

Solution E.: *This is an open-ended question with many correct answers. Some possible answers:*

- *Grid search: Good to explore the space almost exhaustively. It may be time-consuming, especially if training for long on hyperparameter settings that do not give good results.*
- *Random search: While it is not as structured, it still covers the hyperparameter space with enough trials. Can be a good way to understand which hyperparameters matter more in less time than grid search.*
- *"Babysitting": With experience, a person can understand the effect each hyperparameter has on a network, and thus with good intuition they can move the network in the right direction. In practice, this is hard to do precisely and a more structured approach like one of the other two approaches may work better.*

PyTorch implementation

Problem F [20 points]:

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

- **Linear:** A fully-connected layer
  - In convolutional networks, Linear (also called dense) layers are typically used to knit together higher-level feature representations.
  - Particularly useful to map the 2D features resulting from the last convolutional layer to categories for classification (like the 1000 categories of ImageNet or the 10 categories of MNIST).

  - Inefficient use of parameters and often overkill: for A input activations and B output activations, number of parameters needed scales as $O(AB)$.

- **Conv2d:** A 2-dimensional convolutional layer
  - The bread and butter of convolutional networks, conv layers impose a translational-invariance prior on a fully-connected network. By sliding filters across the image to form another image, conv layers perform "coarse-graining" of the image.
  - Networking several convolutional layers in succession helps the convolutional network knit together more abstract representations of the input. As you go higher in a convolutional network, activations represent pixels, then edges, colors, and finally objects.
  - More efficient use of parameters. For N filters of $K \times K$ size on an input of size $L \times L$, the number of parameters needed scales as $O(NK^2)$. When N, K are small, this can often beat the $O(L^4)$ scaling of a Linear layer applied to the L^2 pixels in the image.
- **MaxPool2d:** A 2-dimensional max-pooling layer
  - Another way of performing "coarse-graining" of images, max-pool layers are another way of ignoring finer-grained details by only considering maximum activations over small patches of the input.
  - Drastically reduces the input size. Useful for reducing the number of parameters in your model.
  - Typically used immediately following a series of convolutional-activation layers.
- **BatchNorm2d:** Performs batch normalization (Ioffe and Szegedy, 2014). Normalizes the activations of previous layer to standard normal (mean 0, standard deviation 1).
  - Accelerates convergence and improves performance of model, especially when saturating nonlinearities (sigmoid) are used.
  - Makes model less sensitive to higher learning rates and initialization, and also acts as a form of regularization.
  - Typically used immediately before nonlinearity (Activation) layers.
- **Dropout:** Takes some probability and at every iteration sets weights to zero at random with that probability
  - An effective form of regularization. During training, randomly selecting activations to shut off forces network to build in redundancies in the feature representation, so it does not rely on any single activation to perform classification.
- **ReLU (activation):** Sets negative inputs to 0
- **Softmax (activation):** Rescales input so that it can be interpreted as a (discrete) probability distribution.
- **Flatten:** Flattens any tensor into a single vector (required in order to pass a 2D tensor output from a convolutional layer as input into Linear layers)

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

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

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

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

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

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

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

Hints:

- You are provided with a sample network that achieves a high accuracy. Starting with this network, modify some of the regularization parameters (layerwise regularization strength, dropout probabilities) to see if you can maximize the test accuracy. You can also add layers or modify layers (e.g. changing the convolutional kernel sizes, number of filters, stride, dilation, etc.) so long as the total number of parameters remains under the cap of 1 million.
- You may want to read up on successful convolutional architectures, and emulate some of their design principles. Please cite any idea you use that is not your own.
- To better understand the function of each layer, check the PyTorch documentation.
- Linear layers take in single vector inputs (ex: *(784,)*) but Conv2D layers take in tensor inputs (ex: *(28, 28, 1)*): width, height, and channels. Using the transformation `transforms.ToTensor()` when loading the dataset will reshape the training/test X to a 4-dimensional tensor (ex: *(num_examples, width, height, channels)*) and normalize values. For the MNIST dataset, *channels=1*. Typical color images have 3 color channels, 1 for each color in RGB.

- If your model is running slowly on your CPU, try making each layer smaller and stacking more layers so you can leverage deeper representations.
- Other useful CNN design principles:
  - CNNs perform well with many stacked convolutional layers, which develop increasingly large-scale representations of the input image.
  - Dropout ensures that the learned representations are robust to some amount of noise.
  - Batch norm is done after a convolutional or dense layer and immediately prior to an activation/nonlinearity layer.
  - Max-pooling is typically done after a series of convolutions, in order to gradually reduce the size of the representation.
  - Finally, the learned representation is passed into a dense layer (or two), and then filtered down to the final softmax layer.

Solution F:

Please see the provided solution Jupyter notebook for this problem. Sample test accuracies for 10 dropout probabilities equally spaced in $[0, 1]$*:*

[0.9930999999999998, 0.9913999999999995, 0.9925000000000005, 0.9906000000000004, 0.9887000000000002, 0.9899999999999999, 0.9895000000000005, 0.9864000000000005, 0.9817000000000002, 0.9618999999999998]

Any justification for design strategies, so long as they seem well-justified, suffice. A strong answer should discuss higher-level feature representations, effects of regularization, etc.

Key point. *As stated in the problem, we validate our hyperparameters by looking at the final test accuracy of the model under consideration. This has the tendency to bleed in information about the test set into the design of the convolutional network. Typically, this problem is dealt with by maintaining a separate* validation *set that can either be held out from the training set, or cross-validated. In this scheme, the test set is* ***only used once we have finalized our model design and hyperparameters****.*