Caltech CS/CNS/EE 155
Jayden Nyamiaka

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

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

TA Office Hours

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

1 Deep Learning Principles [35 Points]

Relevant materials: lectures on deep learning

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

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

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

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

Solution A.: In about 250 iterations, the first network (the one at the first link) trains the dataset very well with 0.001 training loss and 0.001 test loss whereas the second network (the one at the second link) barely gets any better staying at a constant 0.508 training loss and 0.497 test loss after less than 10 iterations. This is because the weights of the second network are all initialized to 0 where the first network has nonzero weights, and the backpropagation algorithm doesn't work well with weights of 0. The backpropagation algorithm takes the loss calculated at the final output layer and uses the chain rule on each layer in reverse order to send the gradients of the loss with respect to each weight back through the network to update its weights. This way, the weights of nodes that predicted correctly can be increased and the weights of incorrect nodes can be decreased, resulting in lower loss for that specific run and, after enough epochs, weights that better fit the dataset. This means in our back propagation with ReLU, the weights of further layers will be present as factors in our computation of the loss with respect to the weights of earlier layers. So, if our weights are initialized to 0, the gradient of the loss with respect to earlier input layers will have 0 as a factor and thus equal 0. Therefore, the gradient descent will result in no change in weights for each point, and the weights will stay at 0. This is why the second network barely gets better: it can't update its weights. On the other hand with nonzero weights, the ReLU and weights will result in nonzero factors during backpropagation, so the gradient descent will update the weights and result in a better model overtime and the very well performing model we see at 250 iterations. The following analysis mathematically proves why initializing weights to 0 yields a model that doesn't improve.

Let loss L = loss evaluated for our NN

w_f, x_f, s_f be the weights, non-linearity (ReLU), and resulting output of the final layer

w_h, x_h, s_h be the weights, non-linearity (ReLU), and resulting output of the hidden layer before the final layer

Backpropagation takes L and gets $\frac{\partial L}{\partial w_f}$ and $\frac{\partial L}{\partial w_h}$ to update w_f and w_h using gradient descent.

First, $\frac{\partial L}{\partial w_f}$ is calculated very simply via: $\frac{\partial L}{\partial w_f} = \frac{\partial L}{\partial x_f} \cdot \frac{\partial x_f}{\partial s_f} \cdot \frac{\partial s_f}{\partial w_f}$

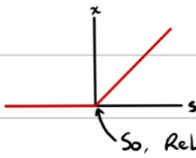
Now, we can consider these factors if all our weights are 0.

Recognize s_f is the sum at the final layer $s_f = w_f \cdot x_h$ where we are summing the weights w_f applied to the outputs of the previous layer x_h. So, since all our weights in w_f are 0, $s_f = w_f \cdot x_h = 0$ regardless of the value of x_h.

Thus, x_f, which is our activation ReLU applied to the sum s_f, will also equal 0

via $x_f = \text{ReLU}(s_f) = \text{ReLU}(0)$

Consider that ReLU = max(0, s) plotted is



So, ReLU(0) is here. And, now we can consider $\frac{\partial x_f}{\partial s_f}$, which for $s_f = 0$, $x_f = 0$ is here.

Technically, the derivative here does not exist, but for practical machine learning purposes, it's basically considered to be 0.

Thus, with the weights being 0 and ReLU as the activation function, $\frac{\partial x_f}{\partial s_f} = 0$, meaning

$$\frac{\partial L}{\partial w_f} = \frac{\partial L}{\partial x_f} \cdot \frac{\partial x_f}{\partial s_f} \cdot \frac{\partial s_f}{\partial w_f} = 0.$$

And, since all subsequent loss derivatives with respects to weights (i.e $\frac{\partial L}{\partial w_h}$, ...) rely on $\frac{\partial L}{\partial w_f}$ as factors in backpropagation, then all will equal 0, so there will be no change to the weights during the updates, explaining why the model isn't improving.

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 first model, for the first 400 iterations or so, the model remains poorly trained and relatively unchanged at around 50 / 50 accuracy before it starts improving. Then from 400, it gets better at a really fast rate, then it asymptotically approaches 0 loss. By 1000 iterations, it has a pretty good model with both the test loss and training loss at 0.009, and by 4000, both the test loss

and training loss are 0.001. Compared to ReLU, the sigmoid model results in smoother boundaries. With ReLU, boundaries were pretty rigid whereas with sigmoid, the boundaries are rounded off and the model space is closer to resembling a circular shape. Both the ReLU and sigmoid models are able to achieve a test and training loss of 0.001, but ReLU reaches it much faster.
For the second model, we are able to get a jump in accuracy at around 3300 iterations. For all the iterations before that the model stays at the constant 0.508 training loss and 0.497 test loss, same as its ReLU counterpart. But from around 3300 to 4000 iterations, the model improves to a test loss of 0.412 and training loss of 0.396. This is significant because the ReLU model never improved at all. This difference can be explained in the difference between their activation functions. With ReLU, the learning can happen faster because as the input to ReLU is > 0 and increases, the derivative of ReLU increases. Realize ReLU = max (0, s), so with s > 0, derivative of ReLU = 1, and ReLU increases proportionally with bigger values of s. However, in sigmoid, we can compute its gradient with respect to its input s as $e^{-s} / (1 + e^{-s})^2$. From this, we see that its derivative is at its largest of ¼ at 0, and smaller everywhere else. Both larger and smaller input values yield smaller and smaller gradients since there's asymptotic behavior at the ends, and the derivative is 0 at negative and positive infinity. This makes it so that ReLU outputs larger gradients to effect change during the backpropagation update whereas sigmoid can't do that. Unless input to sigmoid is 0, its gradient won't yield maximum change, whereas ReLU outputs its maximum value of 1 for every positive case, yielding larger updates and thus faster learning.
On the other hand, for the second model, sigmoid is able to produce a better model after 400 iterations because it is able to recover from weights initialized to 0. Since sigmoid has a nonzero gradient everywhere, although they are super small, nonzero updates are occurring during backpropagation even with 0 weights. Since the gradients of sigmoid are never 0, we are able to keep making updates and bring the weights away from 0 and to nonzero values. It takes a while since the updates are so small, but after about 3300 iterations, the model is able to revive its weights resulting in the jump in accuracy. As we proved before, however, ReLU is not able to do that because once its weights are 0, they can't be revived from 0.

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: By constantly feeding our model negative point after negative point for 500 points, we'll be telling our model that it should always output a negative label. For these 500 points, it won't be paying attention to which parameters equal what since it should always be predicting negative, so our model will develop a large negative bias. And, since we have a large negative bias term, we'll end up with zero or negative weights. Consider that the update during backpropagation is w_after = w_before - gradient of loss with respect to w * learning rate + bias. If the bias becomes too negative , then the weights will get smaller and smaller as they're subtracted by the large negative bias. Then, if we reach a point where the weight becomes zero or negative, then the gradient of ReLU will never again output anything but 0 (since we know ReLU has a derivative of 0 at all negative values). If all our weights reach this point, then we'll no longer be able to update our model, and our model will basically just be a constant function with nothing but this large bias term. So given any points, our model will continue to predict negative predictions, no matter the input, and won't be able to learn because all the update derivatives have become negative. This is the "dying ReLU" problem and the reason we have Leaky ReLU as an alternative.

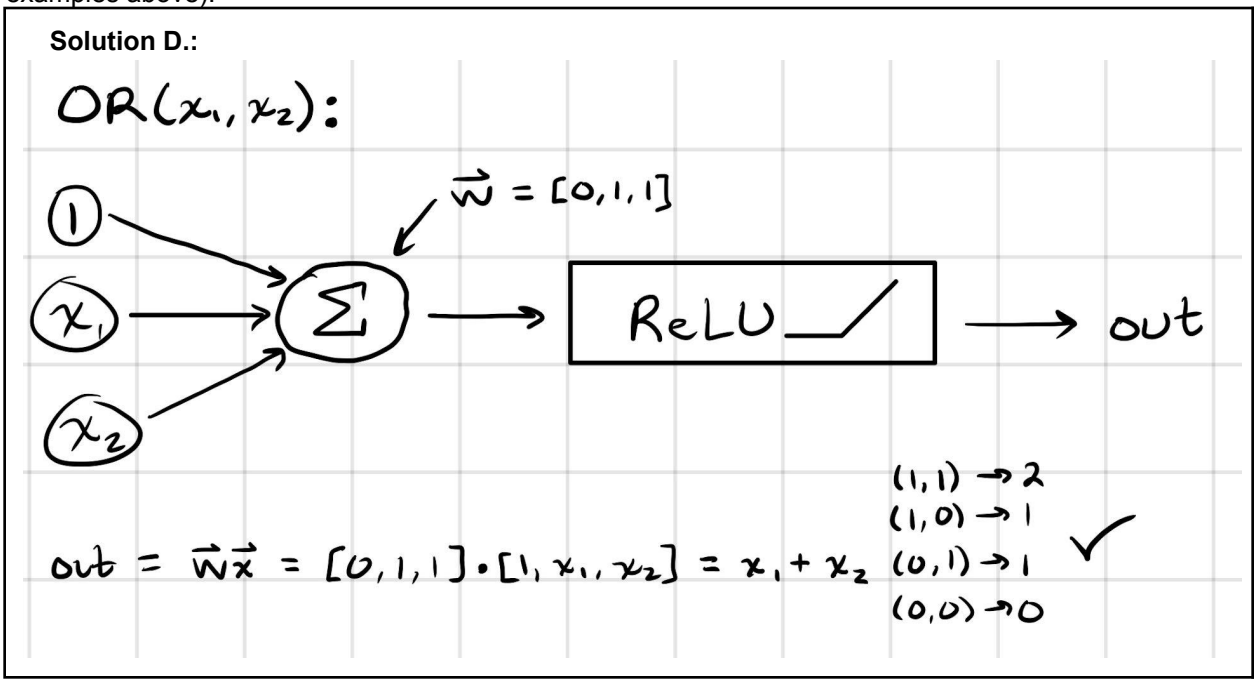
Problem D: Approximating Functions Part 1 **[7 Points]**

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]**

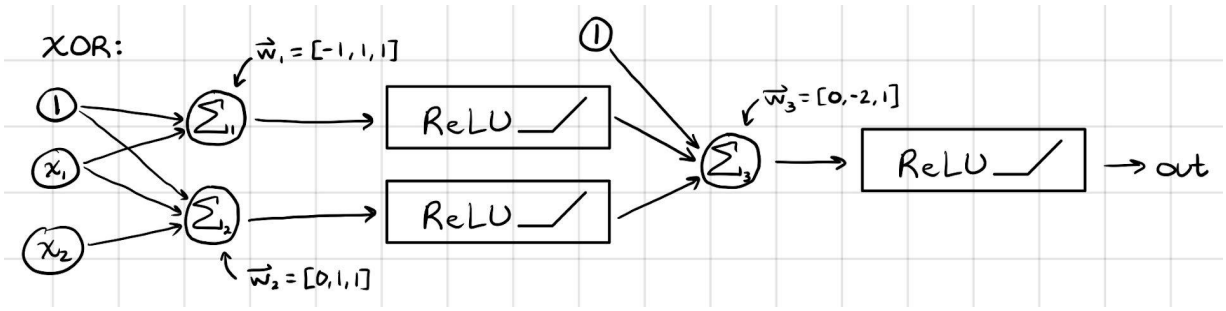
What is the minimum number of fully-connected layers (with ReLU units) needed to implement an XOR of two 0/1-valued inputs x_1, x_2? Recall that the XOR function is defined as:

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

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.:



You need at least 2 layers to compute XOR because its dataset is not linearly separable in 2 dimensions. It's the same way you cannot split XOR with 1 line on that graph from the lecture. Using only 1 layer is mathematically equivalent to using that one line. You need at least 2 layers to give it that multidimensionality which is similar to drawing two lines. This occurs because the activation function's nonlinearity brings another dimension. XOR is also logically equivalent to an inverse AND combined with an OR. In order to simulate this, we need 1 layer to do the inverse AND and OR and then another layer to combine them, so 1 layer simply isn't enough.

2 Inside a Neural Network [17 Points]

Relevant Materials: Lectures on Deep Learning

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

Problem A: Installation [2 Points]

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

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

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

Solution A: torch version: torch-1.13.1+cu117 torchvision version: torchvision-0.14.1+cu117

Problem B: The Data [5 Points]

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

Explain your preprocessing decisions.

Solution B:
https://colab.research.google.com/drive/1CZyUjvYWiKIIh20eaM7cwOjR0CBI7HMx?usp=sharing

- We can start by checking for and dropping any duplicates. There end up being none for this dataset, but it's still good to check.
- Next, we can drop the date columns/inputs for cdc_report_dt, pos_spec_dt, and onset_dt because they are missing values and are already approximately encapsulated via the cdc_case_earliest_dt column. According to the source at data.cdc.gov, cdc_case_earliest_dt shows the earliest valid date on file between the three dates cdc_report_dt, pos_spec_dt, and onset_dt. We need an approximate date for when the person contracted COVID as that may determine if they survived, so since this column has an approximately consistent date for each data point, we can use this. The earlier variants of COVID are known to be more deadly and less infectious, so knowing the approximate time the person had COVID may evince the lethality of the COVID variant they contracted and thus may prove to be an important input variable. So, we'll keep cdc_case_earliest_dt for sure. We can preprocess this data in two steps. First, we'll convert all the string times into time stamps so that we have a continuous numerical representation for our times. Then, we'll perform normalization by subtracting the mean and dividing by the standard deviation. Now, we can consider these points preprocessed.
- We can also drop all data points without death_yn values of "Missing" because death_yn is acting as our dependent variable, and it'd be pointless to train our data on a dependent variable we don't know. To preprocess these, we simply set all yes to 1 and all no to 0. Now, we have a

numerical boolean representation for our dependent variable and loss function.

- We remove race/ethnicity because morally, we shouldn't be using racial identity to predict something as important as someone's chances of dying from COVID. Someone's racial identity should not be taken into account when considering the implications of such a prediction like receiving treatment or access to medication. The neglect of this fact is a reason so many developed AIs have had racist tendencies in the past.
- We choose to include all the other columns in some way for the following respective reasons:
- current_status: having a laboratory confirmed case means someone definitely had COVID whereas a probable case may mean they could have had something else. Probable cases may mean the person had another disease with similar symptoms like a cold, flu, or pneumonia which may make those people more or less likely to survive compared to others. Similar to death_yn, we preprocess by setting Laboratory-confirmed case to 1 and Probable Case to 0.
- sex, age_group: people of different sexes and age groups are affected by illnesses differently. This holds true for almost all sicknesses in regard to resistance, recovery, etc, so this must be considered. Both sex and age group features have data points where there is no value set for those features; however, the proportion of these points in the data set (after we've removed points with missing or unknown death status) is so low (< 1%) that we can just remove these points. So, we remove all points that have missing or NA sex and age group. Now, we should have no problem. To preprocess sex, we simply set male to 1 and female to 0. This could also work vice versa. To preprocess age group, we have a couple options. I decided to label each category from "0 - 9 Years" to "80+ Years" with 0 to 8 respectively. This preserves the linear distance between the ages of the categories while giving us a numerical representation. Lastly, we can perform normalization on this numerical representation now using mean and standard deviation.
- hosp_yn, icu_yn, medcond_yn: Being hospitalized, being admitted to the icu, and/or having a pre-existing medical condition are probable indicators that COVID is affecting someone more than average, so these should be considered. However, all of 3 of these features have a high percentage of missing or unknown values. To mitigate this, we'll make a separate unknown category for these 3 features. Then, we can perform One-Hot encoding on each of these 3 features creating 9 features out of 3, one for yes, no, and unknown for each all set to either 1 or 0 for true or false. This allows us to capture the differences from the 3 categories easily without adding in a concept of uneven distance between any of the yes, no, unknown categories. We can then remove the unknown feature from all 3 of these because it is already captured by the other 2 yes and no features. Consider that if both no_hosp and hosp are false then that's equivalent to having hosp unknown, thus we don't separately need a separate hosp unknown feature. This arises because the yes, no, and unknown features are interconnected, and we can extend this to all 3 hosp_yn, icu_yn, medcond_yn. Lastly, we should consider that hosp, icu, and medcond are all probably interconnected, especially hosp and icu as if someone is hospitalized they are way more likely to be admitted to the ICU. Although these all may be interconnected slightly, they are not connected so much that they express the same thing, so we can recognize they bring more information to our data, so we should still consider them as different features. The performance of neural networks specifically doesn't decrease as much as other algorithms when provided with interconnected data, so we can evaluate that there is more to gain by including all 3 of them rather than removing one.
- Finally, we can isolate the death_yn column from the rest of the features as the labels and add in a column of all 1s at column 0 for the bias term.
- We will then end by normalizing all columns except the bias and death_yn columns since normalization helps networks perform better for all data, including categorical data. With that, we have preprocessed our data.

Problem C: Linear Neural Network [5 Points]

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

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

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

Solution C:
https://colab.research.google.com/drive/1CZyUjvYWiKIIh20eaM7cwOjR0CBI7HMx?usp=sharing
My testing and training loss for both neural nets are displayed. For ease of grading, I've also put them here.
The heat maps can be seen at the end of this pdf.

1-Layer Neural Net:
Train Epoch: 1 Loss: 0.3133
Train Epoch: 2 Loss: 0.3133
Train Epoch: 3 Loss: 0.3142
Train Epoch: 4 Loss: 0.3134
Train Epoch: 5 Loss: 0.3133
Train Epoch: 6 Loss: 0.3133
Train Epoch: 7 Loss: 0.3133
Train Epoch: 8 Loss: 0.3137
Test set: Average loss: 0.0107,
Accuracy: 63252/65159 (97.0733)

2-Layer Neural Net:
Train Epoch: 1 Loss: 0.3133
Train Epoch: 2 Loss: 0.3151
Train Epoch: 3 Loss: 0.3133
Train Epoch: 4 Loss: 0.3133
Train Epoch: 5 Loss: 0.3133
Train Epoch: 6 Loss: 0.3133
Train Epoch: 7 Loss: 0.3137
Train Epoch: 8 Loss: 0.3133
Test set: Average loss: 0.0107,
Accuracy: 63233/65159 (97.0442)

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

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

Solution D: We can expect both the 1-layer and 2-layer network to perform similarly because there isn't any nonlinear activation function applied between layers. By applying nonlinearity at each step of our neural network, the activation functions transform each weighted sum for our classification such that the data becomes more separated by our model with more layers. Without this nonlinearity, however, nothing is transformed and nothing is separated such that all the weighted sums for each layer preserve their relative linear distances to one another. So, without the nonlinear activation function, numerous layers are basically equivalent to just having one layer since all linear relationships are preserved between the layers. Thus, a 2-layer model with no activation function between the two layers is basically mathematically equivalent to the 1-layer model, which is why we can expect the performance to be the same between the models.

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.: Examining the input data, we can see (by taking the len of each dataset) that the training dataset has 60000 images while the testing dataset has 10000 images. Each dataset represents each row as a tuple of an input Tensor that holds the image representation and the int target for the number in the image. Each tuple can be accessed simply using [i] notation. Using len, we can examine this further. Each input Tensor holds a single data structure that stores the representation of the image. Each of these data structures then has 28 elements each with 28 floating point values, essentially constituting a 2D Tensor for the image of floating point values ranging from 0 to 1, where each floating point value represents the shading magnitude for each pixel in the image. Instead of using 3 floating point values to represent a color as usual, a single floating point value is enough here because we are doing grayscale coloring, not rgb.
You can check my code to see how I worked through this.

Model submission instructions:

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

Problem B: Modeling Part 1 [8 Points]

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

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

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

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

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

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

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

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

Solution B: Yields evaluation accuracy of 0.9754
I trained models with different parameters then compared them to select the final model. You can see the output graph on the last couple of pages.
https://colab.research.google.com/drive/1z4z7588zjzWV1TgpqV_pqMVnaTWOlXS0?usp=sharing
Using Adam, learning rate of 1e-3, back size of 32, and 15 epochs:

```
model = nn.Sequential(
  nn.Flatten(),
  nn.Linear(784, 95),
  nn.ReLU(),
  nn.Dropout(0.5),
  nn.Linear(95, 10)
)
```

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=95, bias=True)
  (2): ReLU()
  (3): Dropout(p=0.5, inplace=False)
  (4): Linear(in_features=95, out_features=10, bias=True)
)
Accuracy of ReLU with 100 hidden units: 9754/10000 (97.5400)
```

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: Yields accuracy of 0.9802
https://colab.research.google.com/drive/1b4Tv2uNNEqB-UoPj2L2piQKeqxUnYH6A?usp=sharing
Using Adam, learning rate of 1e-3, batch size of 128, and 40 epochs:

```
model = nn.Sequential(
  nn.Flatten(),
  nn.Linear(784, 100),
  nn.ReLU(),
  nn.Dropout(0.3),
  nn.Linear(100, 100),
  nn.ReLU(),
  nn.Dropout(0.2),
  nn.Linear(100, 10)
)
```

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=100, bias=True)
  (2): ReLU()
  (3): Dropout(p=0.3, inplace=False)
  (4): Linear(in_features=100, out_features=100, bias=True)
  (5): ReLU()
  (6): Dropout(p=0.2, inplace=False)
  (7): Linear(in_features=100, out_features=10, bias=True)
)
Accuracy of Final Model: Accuracy: 9802/10000 (98.0200)
```

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: Yields accuracy of 0.9835
https://colab.research.google.com/drive/1XQmQqUGokDMs_nRukrAJjNV6oPFtlfWh?usp=sharing
Using Adam, learning rate of 1e-3, batch size of 128, and 20 epochs:

```
model = nn.Sequential(
  nn.Flatten(),
  nn.Linear(784, 400),
  nn.ReLU(),
  nn.Dropout(0.3),
  nn.Linear(400, 250),
  nn.ReLU(),
  nn.Dropout(0.3),
  nn.Linear(250, 150),
  nn.ReLU(),
  nn.Dropout(0.3),
  nn.Linear(150, 10)
)
```

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=400, bias=True)
  (2): ReLU()
  (3): Dropout(p=0.3, inplace=False)
  (4): Linear(in_features=400, out_features=250, bias=True)
  (5): ReLU()
  (6): Dropout(p=0.3, inplace=False)
  (7): Linear(in_features=250, out_features=150, bias=True)
  (8): ReLU()
  (9): Dropout(p=0.3, inplace=False)
  (10): Linear(in_features=150, out_features=10, bias=True)
)
Accuracy of Final Model: Accuracy: 9835/10000 (98.3500)
```

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: One of the main benefits of zero padding is that we can capture and train on the data at the edges of our image. In an approach without zero-padding, the data on the edges is lost or not reflected as equally as all the other points because we don't apply the filter through the same construction. This especially makes a difference with small inputs (i.e small images) where there isn't as much data, so preserving the data at the edges is very beneficial. An obvious drawback to the zero padding scheme is the higher cost of computation. Using zero-padding, more work must be done in preprocessing to add the zeros, run the filters since we have more data, and compute those filters, resulting in more computation, time, and memory for training our 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 × 32 × 3 tensor. Your convolution has:

- Size: 5 × 5 × 3
- Filters: 8
- Stride: 1
- No zero-padding

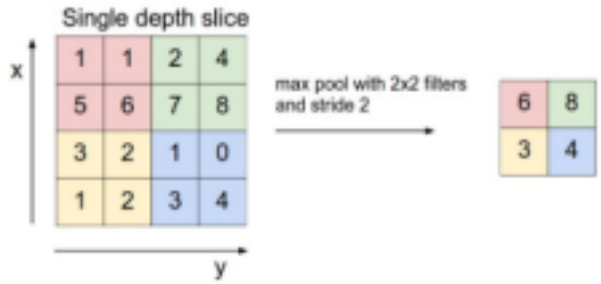
Problem B [2 points]: What is the number of parameters (weights) in this layer, including a bias term? What is the shape of the output tensor?

Solution B.: Having a bias term is equivalent to putting a layer of all ones into our input tensor, so our 32 × 32 × 3 input tensor becomes a 32 × 32 × 4 tensor where one of the 32 × 32 2D submatrices is all ones. Going off this, there will be 5 × 5 × 4 filters for every one of the 8 output channels where one of the 4 z dimensions corresponds to the bias term. No zero padding means we can't apply our filter to the image with each pixel as the center; the first and last two pixels off the edge of each 32 × 32 image can't be in the center when applying our filter due to the lack of padding. So, each filter can only be applied 28 times across a series of 32 pixels, or 28 × 28 times for our 32 × 32 pixel image, yielding a 28 by 28 output matrix for each filter. Thus, for our 8 output filters, this means we have 8 of these 28 × 28 matrices, producing an output tensor of shape 28 × 28 × 8.

Max/Average Pooling

Pooling is a downsampling technique for reducing the dimensionality of a layer's output. Pooling iterates across patches of an image similarly to a convolution, but pooling and convolutional layers compute their outputs differently: given a pooling layer *B* with preceding layer *A*, the output of *B* is some function (such as the max or average functions) applied to patches of *A*'s output.

Below is an example of max-pooling on a 2-D input space with a 2 × 2 filter (the max function is applied to 2 × 2 patches of the input) and a stride of 2 (so that the sampled patches do not overlap):



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: *See GitHub*

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

Solution C.: *1. Average pooling.* *2. Max pooling.*

Average: $\begin{bmatrix} 1 & 1/2 \\ 1/2 & 1/4 \end{bmatrix}, \begin{bmatrix} 1/2 & 1 \\ 1/4 & 1/2 \end{bmatrix}, \begin{bmatrix} 1/4 & 1/2 \\ 1/2 & 1 \end{bmatrix}, \begin{bmatrix} 1/2 & 1/4 \\ 1 & 1/2 \end{bmatrix}$

Max: $\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$

Problem D [4 points]:

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

Solution D.: We can benefit from pooling here because it weeds out the noise. Pooling works by reducing the number of data and parameters, and ideally, discards irrelevant data by consolidating each patch. This is exactly what we need here: by using max pooling, for example, the patch that contains say a missing pixel would take the value of the highest pixel in the patch (which wouldn't be the missing pixel), and thus the missing pixel would have no effect and be weeded out of our input. So, even though some individual pixels may be missing or distorted, pooling allows us to weed them out using the pixels surrounding them and assuming data around one area should be relatively the same.

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.: The pros and cons of "babysitting" are pretty much the same: it solely depends on intuition. The pro is that if your intuition is really good for a set problem and model, you may choose a set of parameters that models the data really well and is very close to the optimal model. However, if your intuition is wrong, you may be really off and make a bad model. Or, even worse: think you have a good model but miss the optimal model by a lot. Depending on intuition is risky, especially when training a model is so expensive because if your intuition based model is wrong, you have to start over. Another pro, however, is that it might not take as long as Grid Search and Random Search because you train one model at a time and can make as large as a change as you want from model to model whereas you have to train many models for one iteration of Grid Search and it can take a long time to get a good model from random search if you're not lucky. Due to this intuition based approach, "babysitting" may be less likely to work on problems that are complex or the designer is unfamiliar with, but really good for problems that are simple or have been well researched in terms of machine learning.

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 to gether 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 distribu tion.
- **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 im ages 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 activa tion/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:
https://colab.research.google.com/drive/1CAv84SGBeH3nKNEVGSjRbKtpP41qzB0V?usp=sharing

Graph is Below.

Final Model Validation Accuracy:

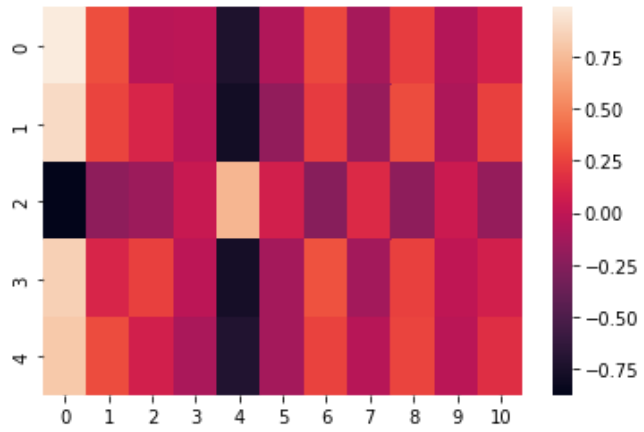
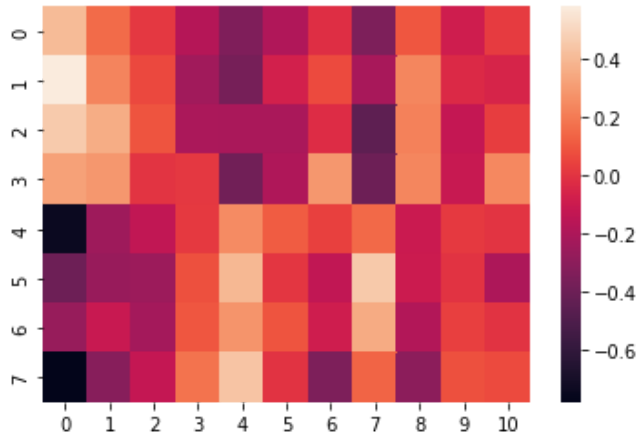
Testing Drop Out Probability = 0.0: val acc: 0.9665
Testing Drop Out Probability = 0.1: val acc: 0.9733
Testing Drop Out Probability = 0.2: val acc: 0.9634
Testing Drop Out Probability = 0.3: val acc: 0.9696
Testing Drop Out Probability = 0.4: val acc: 0.9592
Testing Drop Out Probability = 0.5: val acc: 0.9393
Testing Drop Out Probability = 0.6: val acc: 0.9468
Testing Drop Out Probability = 0.7: val acc: 0.9284
Testing Drop Out Probability = 0.8: val acc: 0.8750
Testing Drop Out Probability = 0.9: val acc: 0.5727

The most effective strategy I used was testing for every drop out probability with the default batch normalization epsilon then setting the one that yielded the greatest test accuracy as the optimal drop out probability and then testing 10 batch normalization epsilons in the evenly spaced range of [1e-5, 1e-10). I then chose the best one out of those as the optimal model. It didn't do that well with RMSprop as the optimizer, so I switched to Adam and it met the validation accuracy requirement of > 0.985. My final model had a drop out probability of 0.1 and a batch norm epsilon of 8e-5, and it outputted a validation accuracy of 0.9898. In my testing, drop out seems to have been most effective because changing the drop out within the range of 0 to 1 yielded the greatest change in validation accuracy. There wasn't much validation accuracy fluctuation when varying the batch normalization epsilon.
Yes, I do see a potential issue with this validation strategy. There may be combinations of parameters that yield the best results together, but don't do as well coupled with other parameters. For example, the optimal batch norm epsilon and dropout may be 2e-5 and 0.4 respectively, but when the dropout rate is tested with the default batch norm epsilon of 1e-5, we find the dropout rate of 0.1 to yield the best results. So, we set 0.1 as the dropout rate, and go on to find the optimal batch norm epsilon. Thus, we've passed up the best combination because we didn't consider that different combinations of parameters may compliment each other. This is why Grid Search trains models by

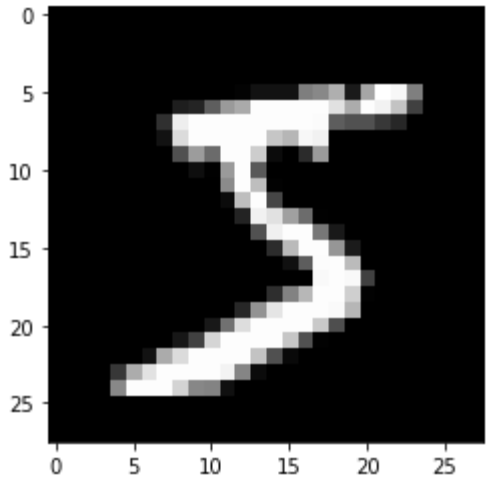
pairs, acknowledging the possibility that hyperparameters may compliment each other. However, for the purposes of this question, this strategy was good enough in finding good enough hyperparameters. It's also worth noting that I just got lucky when I switched from RMSprop to Adam at the end, and the model performed better. This is a problem because you shouldn't rely on luck when making design decisions because then you can't always trust the method to perform.

Generated Images

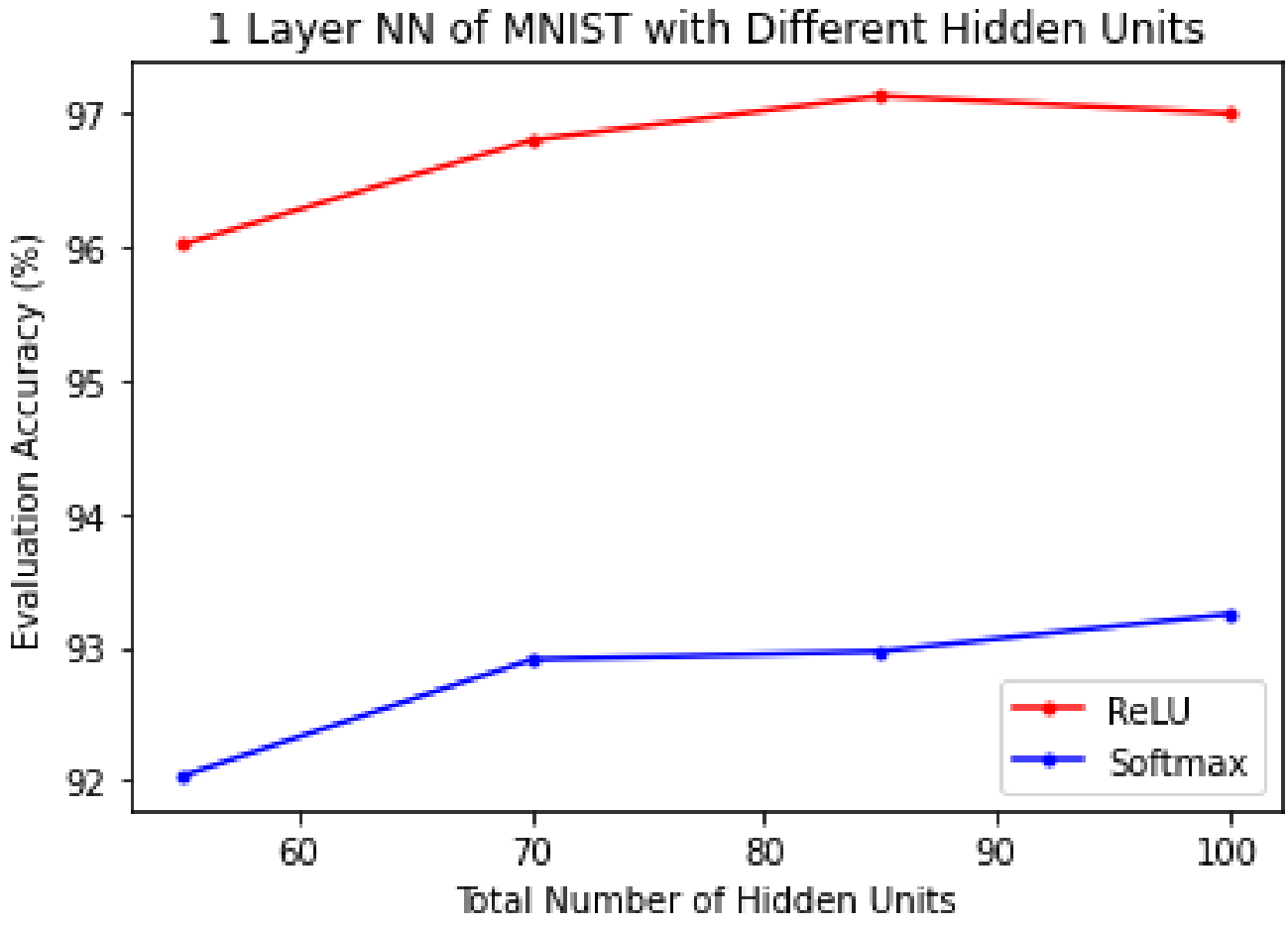
Part 2

1-Layer Net Weight Matrix Visualization	*2-Layer Net Weight Matrix Visualization*
	

Part 3



Graph used to find optimal model



Part 4