1 Deep Learning Principles

Problem A

It appears that the first network has weights initialized between -1 and 1 while the second network has weights initialized to 0. Before the 250th iteration (approximately), we see that the first network has converged by this time such that the training and test losses are minimized greatly. On the other hand, we do not see this same level of success in the second network (up to approx. 250 iterations) such that no learning seems to occur, leaving the test and training losses to be relatively high. Considering the backpropagation algorithm and the ReLU function ($ReLU(x) = max(0, x)$), we see that this difference in performance is due to the weight initialization. With weights initialized as 0, the loss gradients calculated by the backpropagation algorithm have no impact on the weights we want to optimize. Thus, with ReLU(x) outputting zeros, the gradient is zero which explains the lack of learning that we see with the second network. This does not occur with the first network because the weights are initialized to nonzero values which ultimately results in a nonzero gradient which explains the successful learning behavior that we see such that training and test losses are minimized within 250 iterations.

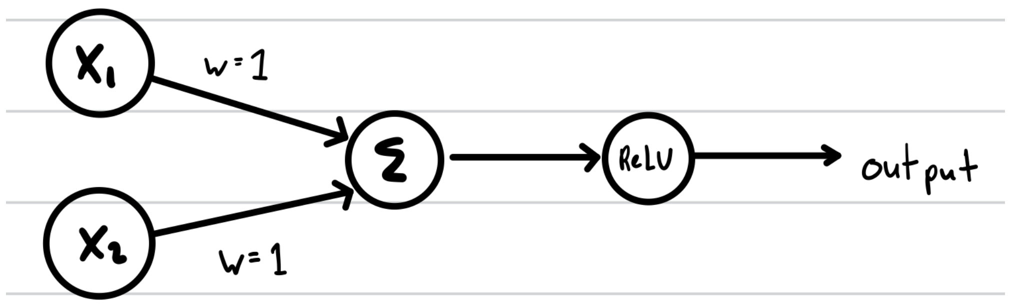
Problem B

With the activation function of sigmoid rather than ReLU, where $\sigma(x) = \frac{1}{1+e^{-x}}$, we examined the two networks up to 4000 iterations. For the first network (nonzero initial weights) we do see a successful learning behavior, but it happens a slower rate relative to problem A. The training and test losses are minimized to a similar level. For the second network (initial weights as zero), we don't see learning progress until after approximately 3000 iterations which was an overall improvement relative to problem A. Although, we should note that the learning process is not as successful as the first network (basically a linear boundary) since the training and test losses are around 0.4 (in comparison to approx. zero). This behavior with the second network is explained by the fact that sigmoid function can return nonzero outputs even with the weights being initially set to zero. It still takes a long time for the backpropagation to update the weights at a somewhat significant level due to their initialization. In other words, the second network gives us a case with saturating non-linearities. Furthermore, the performance of the first network is related to this logic because the sigmoid function gives us a case of saturating non-linearities with the smaller gradients. Thus, the weights are updated more slowly relative to problem A with the ReLU activation.

Problem C

We have a case of the "dying ReLU" problem if we train a fully-connected network with ReLU activations using SGD such that we loop through all of the 500 negative examples before any of the 500 positive examples. Considering the ReLU function defined by $ReLU(x) = max(0, x)$, we can clearly see that ReLU will output 0 for all of the negative examples, and the network needs to "adapt" in order to classify these points correctly. This adaption produces a significant negative bias in the weights such that by the time the positive examples are reached, they become negative themselves where ReLU still returns 0. Thus, further learning is likely to be restricted as the gradients become zero and the weights stop being updated by backpropagation. Therefore, the ReLU function can be thought of as "dead" in this case.

Problem D



$OR(x_1=1, x_2=0) = OR(0,1) = 1 \geq 1$ ✓

$OR(0,0) = 0$ ✓

$OR(1,1) = 2 \geq 1$ ✓

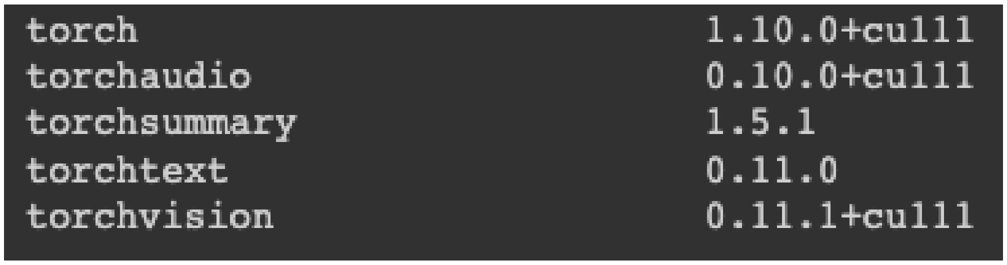
Problem E

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 is 2 (not including output layer). This is because XOR is not a linear function (we can imagine XOR not being linearly separable for $x_1 \in \{0,1\}, x_2 \in \{0,1\}$). In other words, since we would need two lines to separate $x_1 \in \{0,1\}, x_2 \in \{0,1\}$ with XOR, it is not possible to implement XOR with ReLU units with less than 2 layers.

2 Depth vs. Width on the MNIST Dataset

Question 2 Code

Problem A

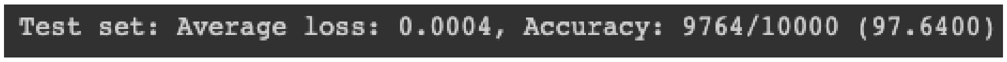


```
torch                         1.10.0+cu111
torchaudio                    0.10.0+cu111
torchsummary                  1.5.1
torchtext                     0.11.0
torchvision                   0.11.1+cu111
```

Problem B

The height and weight of the images are 28 by 28. The values at each array index correspond to pixels of the image such that the values represent intensity/color of the pixel. There are 60,000 images in the training set and 10,000 images in the testing set.

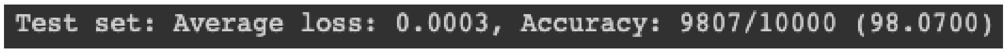
Problem C



```
Test set: Average loss: 0.0004, Accuracy: 9764/10000 (97.6400)
```

See Question 2 Code

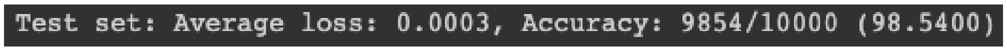
Problem D



```
Test set: Average loss: 0.0003, Accuracy: 9807/10000 (98.0700)
```

See Question 2 Code

Problem E



```
Test set: Average loss: 0.0003, Accuracy: 9854/10000 (98.5400)
```

See Question 2 Code

3 Convolutional Neural Networks

Question 3 Code

Problem A

One benefit to this zero-padding scheme is that we would be able to have output that has the same dimension as the input which can be advantageous in several ways, one being that it might be useful to preserve the size of our output as much as we can. One drawback is that there would be an increase in the level of computation needed to train and produce the model as this padding is processed, particularly with more complex dimensions.

Problem B

With 8 filters of size 5 x 5 x 3 we have 600 parameters, and then we much also consider a bias term for each filter which adds an additional 8 parameters.

$$\implies 8 \cdot (5 \cdot 5 \cdot 3) + 8 \cdot (1) = 608$$

Thus, the number of parameters in this layer is 608.

Problem C

The shape of the output tensor is 28 x 28 x 8 from the input 32 x 32 x 3 tensor.

Problem D

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0.5 \\ 0.5 & 0.25 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0.5 & 1 \\ 0.25 & 0.5 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0.25 & 0.5 \\ 0.5 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0.5 & 0.25 \\ 1 & 0.5 \end{bmatrix}$$

Problem E

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

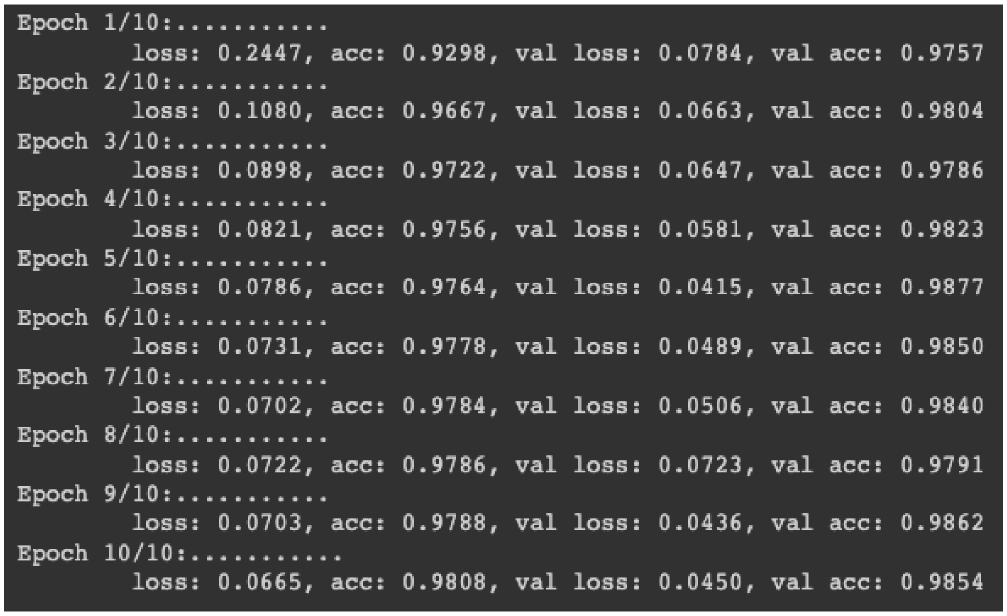
$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Problem F

Pooling might be advantageous given those properties because the model would not penalize the missing pixels too much would represent small amounts of noise. This is because pooling over the appropriately sized patches would help to effectively remove any drastic effects that these noises have while still maintaining the overall image. As seen in the previous two problems, we can imagine how the effects of noise can be minimized using average or max pool. Thus, the model's learning process becomes more efficient at classifying these images of animals even if they appear at various angles/locations with some noise.

Problem G

See Question 3 Code



```
Epoch 1/10:..........
              loss: 0.2447, acc: 0.9298, val loss: 0.0784, val acc: 0.9757
Epoch 2/10:..........
              loss: 0.1080, acc: 0.9667, val loss: 0.0663, val acc: 0.9804
Epoch 3/10:..........
              loss: 0.0898, acc: 0.9722, val loss: 0.0647, val acc: 0.9786
Epoch 4/10:..........
              loss: 0.0821, acc: 0.9756, val loss: 0.0581, val acc: 0.9823
Epoch 5/10:..........
              loss: 0.0786, acc: 0.9764, val loss: 0.0415, val acc: 0.9877
Epoch 6/10:..........
              loss: 0.0731, acc: 0.9778, val loss: 0.0489, val acc: 0.9850
Epoch 7/10:..........
              loss: 0.0702, acc: 0.9784, val loss: 0.0506, val acc: 0.9840
Epoch 8/10:..........
              loss: 0.0722, acc: 0.9786, val loss: 0.0723, val acc: 0.9791
Epoch 9/10:..........
              loss: 0.0703, acc: 0.9788, val loss: 0.0436, val acc: 0.9862
Epoch 10/10:..........
              loss: 0.0665, acc: 0.9808, val loss: 0.0450, val acc: 0.9854
```

*The final test accuracy for 10 epochs was **0.9854*** > 0.985.

Dropout prob. p	Test accuracy after 1 epoch
0	0.9753
0.11111111	0.9706
0.22222222	0.9710
0.33333333	0.9641
0.44444444	0.9687
0.55555556	0.9513
0.66666667	0.9434
0.77777778	0.9041
0.88888889	0.2696
1.0	0.0981

The code of my model can be found through the *Question 3 Code* link at the top of this problem. Overall, I kept the same foundational framework as the sample model, and I ultimately achieved my final model using these building blocks and iteratively changing parameters and modifying the layers slightly until I was satisfied with the outcome. As mentioned in the problem description, an effective strategy regarding time was testing one epoch at a time. I initially found that lowering the dropout probability increased the test accuracy, and from there I experimented with the in channels / out channels parameters with the convolution layer. As per the requirements (and effectiveness), I added a batch normalization layer further strengthened my model. Regarding the regularization methods, it definitely seemed like batch normalization was the most effective through my iterative process and considering that the test accuracy was highest when the dropout probability was zero. Furthermore, it seems like there could be a few issues regarding this way of validating our hyperparameters. For starters, if this model was more complex, this process would certainly not be as efficient with a larger number of hyperparameters. Also, considering more real world applications, it is probably worth considering that we may be overfitting the validation data by specifically picking the hyperparameters that have the best performance with the validation data. This might not always be something to consider depending on the circumstances of the application and validation dataset.