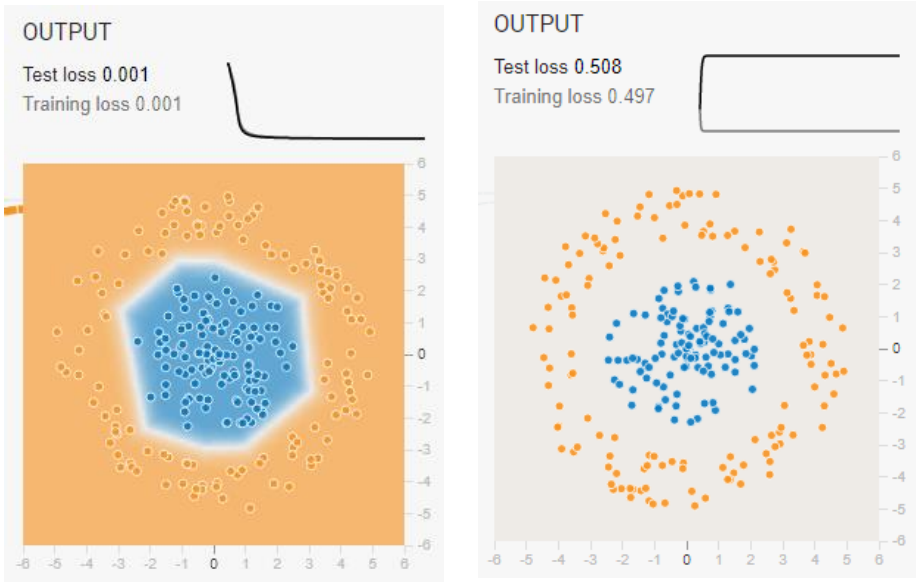
Jessica Wang
Machine Learning & Data Mining
Set 4

Deep Learning Principles

Problem A.

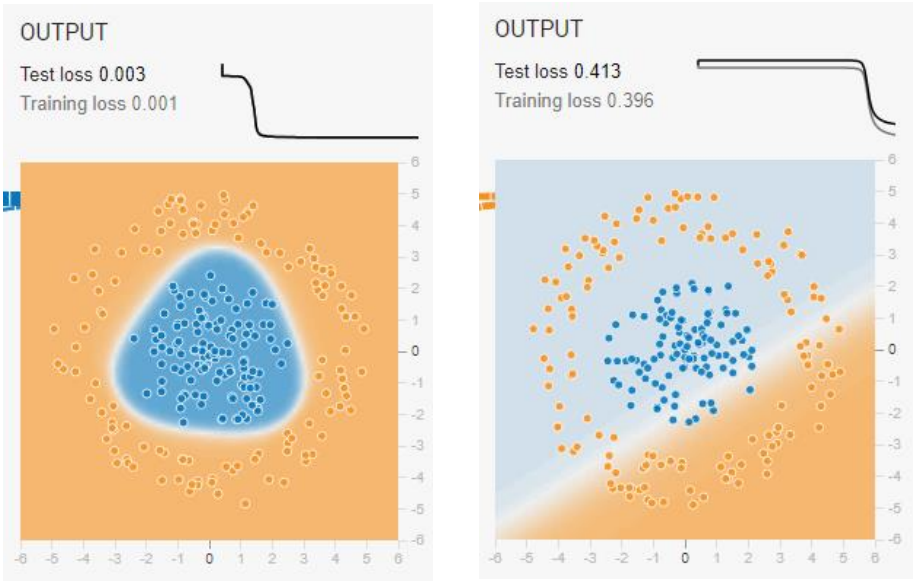
i. The performance of each network is depicted below—both networks have the same architecture and use ReLu activation but the neural network used on the left has weights initialized in the range (-0.5, 0.5) whereas the neural network used on the right has all weights initialized to 0.



Backpropagation calculates the loss gradients with respect to the internal weights. Therefore, if the weights are all initialized to 0, each iteration of backpropagation would have no effect on the weights that we are trying to optimize and each hidden neuron will receive the same signal. The gradient is propagated backward through the network and without the non-linearities, the network collapses to a linear function. The ReLU activation function outputs 0 when the input $x < 0$ and outputs x when the input $x > 0$. As such, we see that when our weights are initialized to 0, the network quickly stops learning and the test loss and training loss are quite high. When the weights are initialized to different values in the range (-0.5, 0.5), the ReLU activation function allows the network to converge quickly and drastically reduce the test loss and training loss.

Jessica Wang
Machine Learning & Data Mining
Set 4

ii. The performance of each network is depicted below—both networks have the same architecture and use sigmoid activation but the neural network used on the left has



weights initialized in the range (-0.5, 0.5) whereas the neural network used on the right has all weights initialized to 0. When the weights are initialized to 0, the network takes much longer to learn. The sigmoid activation function maps large negative values to 0 and large positive numbers to 1 which means the gradient approaches 0 near 0 and 1. Therefore, these saturated neurons do not significantly update. When the weights are initialized to different values in the range (-0.5, 0.5), the network converges more quickly and the test loss and training loss are drastically reduced in comparison.

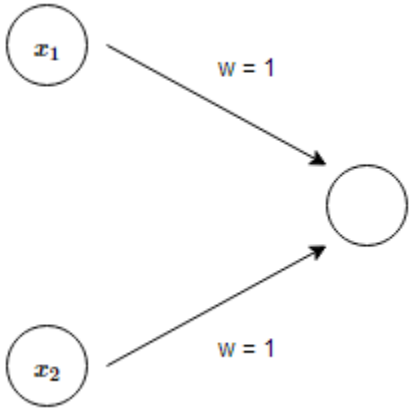
Problem B.

If we train a fully-connected network with ReLU activations using SGD, looping through all the negative examples before any of the positive examples, we experience the "dying ReLU" problem. The ReLU activation function outputs 0 when the input $x < 0$ and outputs x when the input $x > 0$. Since we loop through all the negative examples before any of the positive examples, the dead ReLU always outputs the same value, 0, and learns a large negative bias term so that the gradient flowing through the unit will be 0 and it is unlikely for the weights to recover.

Problem C.

i. The following fully-connected network with ReLU units implements the OR function on two 0/1 valued inputs, x_1 and x_2 with the least number of hidden units possible (sorry for not including the bias term in the diagram but there should be a bias term, set to 0 in my diagram). The ReLU activation function outputs 0 when the input $x < 0$ and outputs x when the input $x > 0$. In other words, ReLU($\sum(w_i x_i + b)$) outputs 0 when the argument inside is negative and vice versa. The diagram is below.

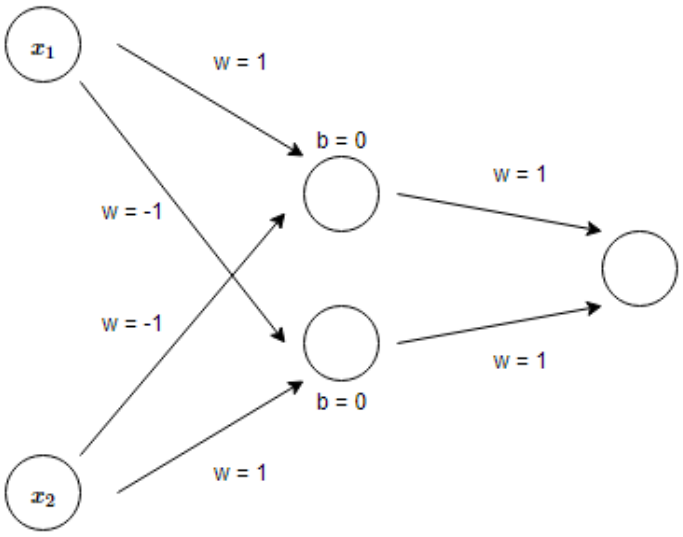
Jessica Wang
Machine Learning & Data Mining
Set 4



The ReLU activation function outputs 0 when the input $x < 0$ and outputs x when the input $x > 0$. Therefore, using the network we have defined, we would correctly output the following which implements the OR function:

OR(1, 0)	1 (≥ 1)
OR(0, 1)	1 (≥ 1)
OR(1, 1)	2 (≥ 1)
OR(0, 0)	0

ii. 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 two. A network with fewer layers cannot compute XOR because XOR is non-linear.



Jessica Wang
Machine Learning & Data Mining
Set 4

Depth vs Width on the MNIST Dataset

Problem A.

Tensorflow version number: 1.5.0
Keras version number: 2.0.9

Problem B.

i. The input data is a 3D array with dimensions (60000, 28, 28). There are 60000 data points or images, each represented by the array index, and the dimensions of each image is 28 by 28 where each value represents pixel intensity.
ii. The new shape of the training input is a 2D array with dimensions (60000, 784). There are still 60000 data points or images, but each 28 by 28 image has been reshaped to a single vector of length 784.

Problem C.

See Jupyter notebook for code. The test accuracy I achieved with 100 hidden units is 0.9776.

Problem D.

See Jupyter notebook for code. The test accuracy I achieved with 200 hidden units and two hidden layers is 0.9805.

Problem E.

See Jupyter notebook for code. The test accuracy I achieved with 1000 hidden units and three hidden layers is 0.9841.

Convolutional Neural Networks

Problem A.

When building a deep neural network, zero padding is beneficial because it prevents the spatial dimensions from decreasing and allows us to preserve a reasonably sized output rather than quickly reduce our input to a few pixels after a few convolutions. However, a drawback to zero padding is that it skews or changes our data since zero's are inserted to preserve the size.

Problem B.

i. 608 parameter (weights): each of the 8 filters has size 5 x 5 x 3 which results in 600 parameters but we need to add the bias term for each filter so we have a total of 608 parameters.
ii. 28 x 28 x 8: the shape of the output tensor is 28 x 28 x 8 since there are 8 filters and the 32 x 32 pixels become reshaped to 28 x 28 since $32 - 5 + 1 = 28$.

Jessica Wang
Machine Learning & Data Mining
Set 4

Problem C.

i.

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

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

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

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

ii.

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

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

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

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

iii. Pooling would be advantageous because it reduces variance and would reduce the effect of the small amounts of noise. Pooling layers reduce the dimensionality to avoid overfitting.

Jessica Wang
Machine Learning & Data Mining
Set 4

Problem D.

See Jupyter notebook for code.

The output of my final model is copied below. It achieves a test accuracy of 0.9889.

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 256s 4ms/step - loss: 0.1507 - acc: 0.9549 - val_loss: 0.0559 - val_acc: 0.9827
Epoch 2/10
60000/60000 [==============================] - 236s 4ms/step - loss: 0.0837 - acc: 0.9749 - val_loss: 0.0508 - val_acc: 0.9847
Epoch 3/10
60000/60000 [==============================] - 232s 4ms/step - loss: 0.0716 - acc: 0.9788 - val_loss: 0.0637 - val_acc: 0.9815
Epoch 4/10
60000/60000 [==============================] - 209s 3ms/step - loss: 0.0628 - acc: 0.9822 - val_loss: 0.0447 - val_acc: 0.9846
Epoch 5/10
60000/60000 [==============================] - 211s 4ms/step - loss: 0.0572 - acc: 0.9833 - val_loss: 0.0417 - val_acc: 0.9869
Epoch 6/10
60000/60000 [==============================] - 210s 4ms/step - loss: 0.0551 - acc: 0.9843 - val_loss: 0.0462 - val_acc: 0.9870
Epoch 7/10
60000/60000 [==============================] - 209s 3ms/step - loss: 0.0506 - acc: 0.9850 - val_loss: 0.0413 - val_acc: 0.9885
Epoch 8/10
60000/60000 [==============================] - 210s 3ms/step - loss: 0.0506 - acc: 0.9854 - val_loss: 0.0430 - val_acc: 0.9878
Epoch 9/10
60000/60000 [==============================] - 210s 3ms/step - loss: 0.0455 - acc: 0.9869 - val_loss: 0.0437 - val_acc: 0.9872
Epoch 10/10
60000/60000 [==============================] - 210s 4ms/step - loss: 0.0448 - acc: 0.9873 - val_loss: 0.0448 - val_acc: 0.9889
60000/60000 [==============================] - 47s 788us/step
[0.015551200077058881, 0.99571666666666669]
10000/10000 [==============================] - 8s 794us/step
[0.044757800438810773, 0.9889]
```

The output of manipulating the dropout probabilities is shown in the Jupyter notebook. I have copied the data into a table for readability.

Dropout probability, *p*	Training score	Training accuracy	Testing score	Testing accuracy
0	0.0392	0.9879	0.0556	0.9817
0.1111	0.0412	0.9875	0.0568	0.9807
0.2222	0.0429	0.9871	0.0602	0.9814
0.3333	0.0455	0.9865	0.05865	0.9824
0.4444	0.0536	0.9841	0.0681	0.9789
0.5556	0.0499	0.9852	0.0525	0.9836
0.6667	0.0655	0.9803	0.0682	0.9787
0.7778	0.0706	0.9788	0.0696	0.9783
0.8889	0.1162	0.9658	0.1131	0.9665
1.0	0.0390	0.9884	0.0544	0.9820

Jessica Wang
Machine Learning & Data Mining
Set 4

The most effective strategies in designing a convolutional network included using dropout to help combat overfitting since it took some probability and at every iteration, set the weights to zero at random with that probability. As such, increasing the probability of dropout was an effective form of regularization. Using Conv2D to network several convolutional layers in succession helped the convolutional network knit together more abstract representations of the input. I observed that increasing Dense greatly increased the number of parameters and as such, was an inefficient use of parameters and often overkill.

Validating our hyperparameters by varying them and training them for an epoch would be problematic if we had a lot of different hyperparameters we wanted to test since training for them for even an epoch takes a very long time. As such, validating our hyperparameters in a "brute force" sort of method by trying different values is quite time-inefficient.