I am using one late hour on this assignment.

1a. We have the backpropagation algorithm defined as

$$\frac{\partial s^L}{\partial x^{L-1}} = W^L$$

We also know that the ReLU function is defined as

$$ReLU(s) = \max(s, 0)$$

Based on these functions, we can tell that the second network will not learn based on the backpropagation algorithm since the initial weights are set to zero. Additionally, the ReLU function will return zero for all inputs, which results in a zero gradient. Therefore, the weights are never updated, so there is no learning involved. The first network has the weights initialized to nonzero numbers, which results in a nonzero gradient. This allows learning, which shows in the output's test and training loss being more efficient.

1b. The backpropagation algorithm does not change, but we now use the sigmoid function:

$$f(s) = \frac{1}{1 + e^{-s}}$$

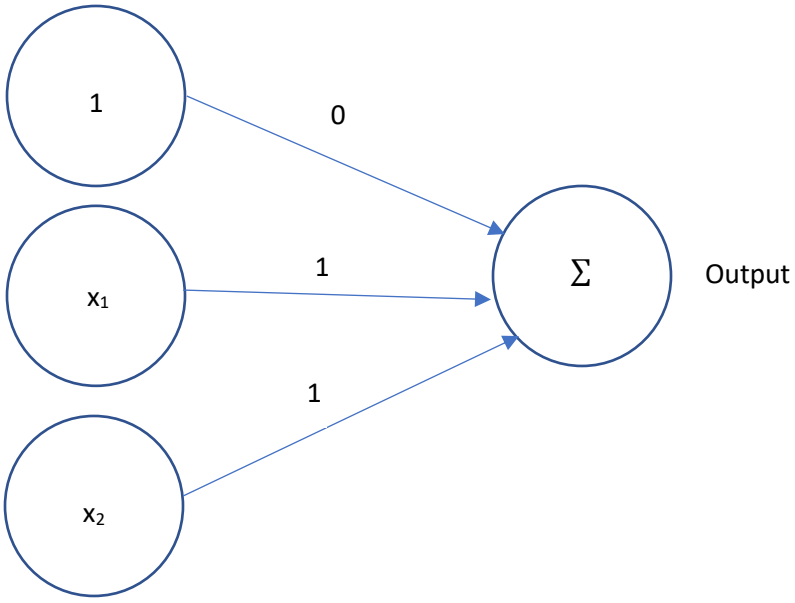
In this case, the sigmoid function will return a nonzero value even when the weights are initially set to zero, which solves one issue of the second network. In this case, we are faced with the vanishing gradient problem. Since the second network starts with the weights set to zero, it takes a lot longer for the backpropagation algorithm to give a significant enough change to the weights. This is why it took the second network around 3400 iterations before any change started happening while the first network took around 600 iterations. Since the first network had nonzero weights, the given algorithm took significantly fewer iterations since the initial weights were updated enough where the gradients become large enough for backpropagation and error loss. The second network took significantly longer since the gradient was practically zero for most of the iterations. It is also notable that the weights for the second network were updated in the same way, which resulted in the same final weights. This gave us a linear ending as shown in the visuals of the website.

1c. As stated in the hint, we have the "dying ReLU" problem that arises when we train a fully-connected network with ReLU activations using SGD by looping through all the negative examples first. Again, the ReLU function is given by:

$$ReLU(s) = \max(s, 0)$$

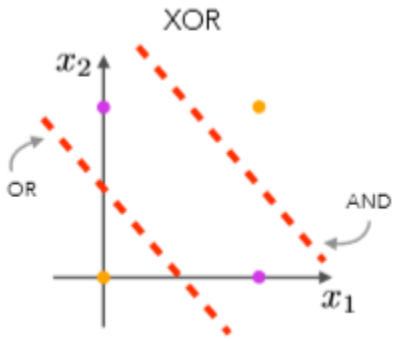
By feeding through all the negative examples, the model adjusts significantly in order to classify all the negative examples correctly, which creates a heavy negative bias. After we do this, the positive examples fed to the ReLU function will become negative and result with a ReLU output of zero. Thus, we say that all future nodes "die" by becoming zero and inhibiting any additional learning.

1d. Considering that we need a bias term:



By using this, we ignore the bias term (since the weight is zero), so we are given zero when both inputs are zero, one when one of the inputs is one, and two when both inputs are one.

1e. The minimum number of fully-connected layers needed to implement an XOR function is three. We know that in order to have only two layers, the XOR function would need to be linearly separable. However, this is not the case, and we need two lines to separate the dataset. XOR is a combination of OR and AND, which are both implemented through an additional layer. Thus, we would need an input layer, a hidden layer, and an output layer to fully establish XOR, which gives us three layers.



2a. Running the code gives:

```
import tensorflow as tf
import keras

print("tensorflow version: " + tf.__version__)
print("keras version: " + keras.__version__)
```

```
tensorflow version: 1.13.0-rc0
keras version: 2.2.4
```

2b. The input images have dimensions of 28 by 28. Each cell corresponds to a pixel on the image, and the value at each pixel ranges from 0 to 255. This corresponds to the RGB color value, where 0 results in black and 255 results in white. The input data is 60000 images with dimensions of 28 by 28, so we have a shape of (60000, 28, 28). The shape of the training output is (60000,). This is shown in my code on 2c.

2c. The new training input shape for X_train is (60000, 784), which can be done with numpy.reshape as shown in my code (or 28 x 28 = 784). The new training output shape for y_train is (60000, 10).

```
import numpy as np
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Flatten, Dropout

# Importing the MNIST dataset using Keras
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()

print("Size of X_train: " + str(np.shape(X_train)))
print("Size of y_train: " + str(np.shape(y_train)))

# One-hot encoding for y_train
y_train = keras.utils.np_utils.to_categorical(y_train)

# Reshape the image arrays
X_train = np.reshape(X_train, (60000, 784))

print("Size of X_train: " + str(np.shape(X_train)))
print("Size of y_train: " + str(np.shape(y_train)))
```

```
Size of X_train: (60000, 28, 28)
Size of y_train: (60000,)
Size of X_train: (60000, 784)
Size of y_train: (60000, 10)
```

2d. My code is attached in file CS155PS4Q2d.ipynb:

```
import numpy as np
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Flatten, Dropout

# Importing the MNIST dataset using Keras
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# One-hot encoding for output arrays
y_train = keras.utils.np_utils.to_categorical(y_train)
y_test = keras.utils.np_utils.to_categorical(y_test)

# Reshape the input arrays
X_train = np.reshape(X_train, (60000, 784))
X_test = np.reshape(X_test, (len(X_test), 784))

# Normalization
X_train = X_train.astype('float')
X_train /= 255
X_test = X_test.astype('float')
X_test /= 255

# Creation of model
model = Sequential()
model.add(Dense(100))
model.add(Activation('relu'))
model.add(Dropout(0.3))

# Last layer
model.add(Dense(10))
model.add(Activation('softmax'))

# compile and fit model
model.compile(loss='categorical_crossentropy', \
        optimizer='rmsprop', metrics=['accuracy'])

fit = model.fit(X_train, y_train, batch_size=128, nb_epoch=20,
   verbose=1)

# Printing the accuracy of our model
score = model.evaluate(X_test, y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

```
Test score: 0.08201487137448275
Test accuracy: 0.9784
```

2e. My code is attached in file CS155PS4Q2e.ipynb:

```
import numpy as np
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Flatten, Dropout

# Importing the MNIST dataset using Keras
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# One-hot encoding for output arrays
y_train = keras.utils.np_utils.to_categorical(y_train)
y_test = keras.utils.np_utils.to_categorical(y_test)

# Reshape the input arrays
X_train = np.reshape(X_train, (60000, 784))
X_test = np.reshape(X_test, (len(X_test), 784))

# Normalization
X_train = X_train.astype('float')
X_train /= 255
X_test = X_test.astype('float')
X_test /= 255

# Creation of model
model = Sequential()
model.add(Dense(100))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(100))
model.add(Activation('relu'))
model.add(Dropout(0.2))

# Last layer
model.add(Dense(10))
model.add(Activation('softmax'))

# compile and fit model
model.compile(loss='categorical_crossentropy', \
              optimizer='rmsprop', metrics=['accuracy'])

fit = model.fit(X_train, y_train, batch_size=256, nb_epoch=30,
    verbose=1)

# Printing the accuracy of our model
score = model.evaluate(X_test, y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

```
Test score: 0.08531460666186831
Test accuracy: 0.9806
```

2f. My code is attached in file CS155PS4Q2f.ipynb:

```
import numpy as np
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Flatten, Dropout

# Importing the MNIST dataset using Keras
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# One-hot encoding for output arrays
y_train = keras.utils.np_utils.to_categorical(y_train)
y_test = keras.utils.np_utils.to_categorical(y_test)

# Reshape the input arrays
X_train = np.reshape(X_train, (60000, 784))
X_test = np.reshape(X_test, (len(X_test), 784))

# Normalization
X_train = X_train.astype('float')
X_train /= 255
X_test = X_test.astype('float')
X_test /= 255

# Creation of model
model = Sequential()
model.add(Dense(333))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(333))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(334))
model.add(Activation('relu'))
model.add(Dropout(0.2))

# Last layer
model.add(Dense(10))
model.add(Activation('softmax'))

# compile and fit model
model.compile(loss='categorical_crossentropy', \
            optimizer='rmsprop', metrics=['accuracy'])

fit = model.fit(X_train, y_train, batch_size=256, nb_epoch=20,
    verbose=1)

# Printing the accuracy of our model
score = model.evaluate(X_test, y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

```
Test score: 0.08870012761469727
Test accuracy: 0.9832
```

3a. Using zero-padding allows us to preserve the spatial size of the original image. We want to preserve as much information about the original input volume, so we can extract as many features as possible. One drawback to zero-padding is that we can lose important features at the zero-padded locations. For example, the edges could contain important features to extract, but the zeros cause the feature to add no contribution in the filter.

3b. We have 8 filters that are 5 x 5 x 3. We also need to add the 8 biases for each filter. Thus,

$$8\ x\ 5\ x\ 5\ x\ 3 + 8 = 608\ paramters$$

3c. The shape of the output tensor will be 28 x 28 x 8, which is the result of only valid convolutions. This is based on the convolution size and stride.

3d. Taking the average of each patch:

$$\begin{bmatrix} 1 & 0.5 \\ 0.5 & 0.25 \end{bmatrix}, \begin{bmatrix} 0.5 & 1 \\ 0.25 & 0.5 \end{bmatrix}, \begin{bmatrix} 0.25 & 0.5 \\ 0.5 & 1 \end{bmatrix}, \begin{bmatrix} 0.5 & 0.25 \\ 1 & 0.5 \end{bmatrix}$$

3e. Taking the max from each patch:

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

3f. Pooling might be advantageous because it would allow the noise to be accounted for by the other pixels in the group. By using pooling, we can select regions and evaluate those areas, so small groups of missing pixels will not have a large scale affect on the overall results. Since the images are taken at different angles and locations, pooling would have a greater effect on the small amounts of noise.

2g. My code is attached in my Moodle submission.

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 109s 2ms/step - loss: 0.6346 - acc: 0.7892 - val_loss: 0.1381 - val_acc: 0.9559
Epoch 2/10
60000/60000 [==============================] - 109s 2ms/step - loss: 0.1839 - acc: 0.9417 - val_loss: 0.0901 - val_acc: 0.9700
Epoch 3/10
60000/60000 [==============================] - 111s 2ms/step - loss: 0.1482 - acc: 0.9537 - val_loss: 0.0630 - val_acc: 0.9795
Epoch 4/10
60000/60000 [==============================] - 109s 2ms/step - loss: 0.1309 - acc: 0.9603 - val_loss: 0.0590 - val_acc: 0.9813
Epoch 5/10
60000/60000 [==============================] - 107s 2ms/step - loss: 0.1228 - acc: 0.9631 - val_loss: 0.0551 - val_acc: 0.9830
Epoch 6/10
60000/60000 [==============================] - 109s 2ms/step - loss: 0.1118 - acc: 0.9665 - val_loss: 0.0546 - val_acc: 0.9836
Epoch 7/10
60000/60000 [==============================] - 108s 2ms/step - loss: 0.1047 - acc: 0.9681 - val_loss: 0.0476 - val_acc: 0.9848
Epoch 8/10
60000/60000 [==============================] - 109s 2ms/step - loss: 0.1014 - acc: 0.9703 - val_loss: 0.0550 - val_acc: 0.9827
Epoch 9/10
60000/60000 [==============================] - 114s 2ms/step - loss: 0.0999 - acc: 0.9705 - val_loss: 0.0468 - val_acc: 0.9852
Epoch 10/10
60000/60000 [==============================] - 111s 2ms/step - loss: 0.0953 - acc: 0.9722 - val_loss: 0.0404 - val_acc: 0.9880
```

The final test accuracy was 0.988. I found the most effective strategy was manipulating the Conv2D layers. Since this is the bread and butter of convolutional networks, by making this layer more accurate, we perform “coarse-graining” of the image. This improved the overall performance of my network, but the backbone of the network comes through the addition of activation and dropout layers. The dropout layers were very effective with regularization, but I needed to find an appropriate value to prevent under regularization or overregularization. The only problem I can see right now is the sigmoid activation layer. As we know, sigmoid plateaus at larger values, so this is being picky with small decimals. However, for complete optimization, this could cause some problems.