2 day free extension granted by Professor Yue

CS 155 PS 4

1 Deep Learning Principles

Problem A

In the first network, the weights are randomly initialized, which allows the backpropagation algorithm to gradually reduce the loss as the ReLU function doesn't saturate as long as some of the weights are positive such that not all sums are negative or zero. In the second network, all the weights are initialized as zero, meaning all the sums will be zero and thus ReLU saturates at zero (the gradient of ReLU and $\partial \mathbf{s}^{L} / \partial \mathbf{x}^{L-1}$ both vanish). This causes the gradients to vanish and the backpropagation algorithm is unable to update the loss using the gradients.

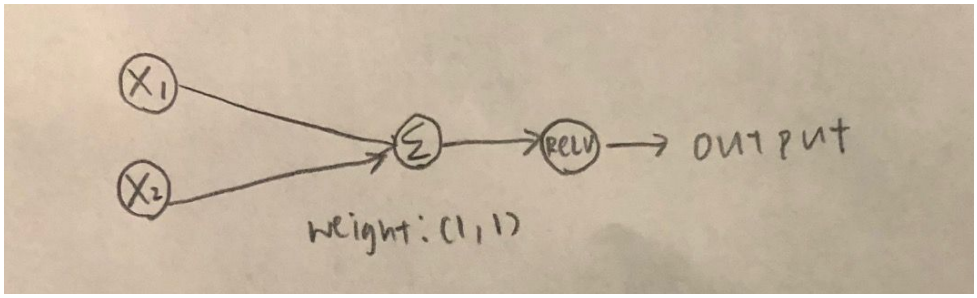
Problem B

For the first network, the decision boundary for the model trained using sigmoid looks smoother compared to the one trained using ReLU, as the sigmoid function uses soft thresholds (everywhere differentiable unlike ReLU). The loss function plateaus at around 200 iterations when the ReLU function is used, while it plateaus past 1000 iterations when the sigmoid function is used. The model is learned a lot slower with the sigmoid function, because it is a saturating non-linearity function meaning the gradient is very small almost everywhere (goes to zero at positive and negative infinities). In backpropagation, the product of many small gradients is even smaller (goes to zero as depth increases), and the small gradients cause the weights to be updated very slowly. For the second network, the model trained using sigmoid eventually learns a little bit (a linear boundary that classifies better than random guessing, with test loss ~ 0.4) after about 3000 iterations, in contrast to when the model learns nothing using ReLU. This is because the sigmoid gradient is very small but nonzero even when the weights are initialized to zero, so after many iterations it is able to learn using the backpropagation algorithm.

Problem C

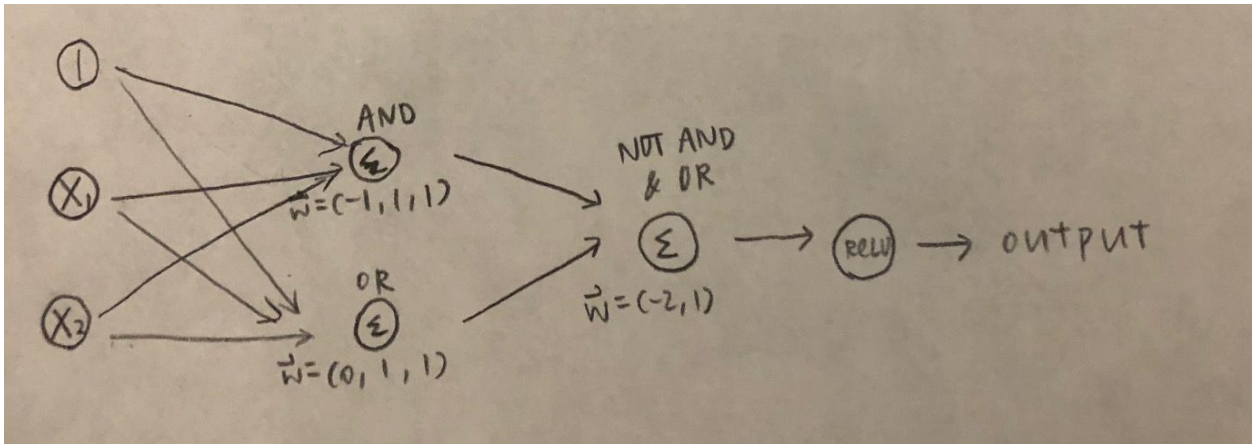
SGD updates the weights using gradients calculated on a single example. Looping through all the negative examples first will introduce a large negative bias term in the weights, so even when we start training on the positive examples, they will be classified as negative. Since all the sum values will be negative, ReLU will saturate at zero, resulting in "dead" neurons that can't recover. The weights will never be updated in the backpropagation algorithm once all the gradients become zero, so introducing positive examples later won't do anything to the dead neurons, and the model can't learn anything further.

Problem D



Using a weight (1, 1) or any values greater than 1 for each parameter would work.

Problem E



The minimum number of layers is 2 (1 hidden layer and 1 output layer (including ReLU activation)), because XOR is not a linear function while each layer with ReLU unit can only compute a linear function. We can compute the XOR function using two linear functions in the first layer, AND and OR, then take the difference of the two functions (OR - AND is equivalent to taking the intersection of NOT AND and OR) in the second hidden layer.

2 Depth vs Width on the MNIST Dataset

Problem A
Tensorflow 1.12.0
Keras 2.2.4

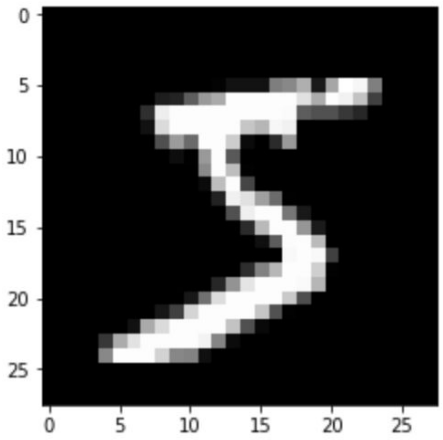
Problem B

```
# Problem B

from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()

print('Shape of training input matrix: ' + str(X_train.shape))
plt.imshow(X_train[0], cmap=plt.get_cmap('gray'))
plt.show()
```

```
Shape of training input matrix: (60000, 28, 28)
```



Each example from the input data is a 28 by 28 image. The values in each array index represent the intensity of each pixel in the image.

Problem C

```
# Problem C

y_train = keras.utils.np_utils.to_categorical(y_train)
y_test = keras.utils.np_utils.to_categorical(y_test)
X_train = np.reshape(X_train, (len(X_train), 28 * 28, ))
X_test = np.reshape(X_test, (len(X_test), 28 * 28, ))
```

The new shape of the training input matrix is (60000, 784), and the new shape of each example in the training input matrix is (784,), which is a vector of length 784.

Problem D

```
# Problem D

# Create a model
model = Sequential()
model.add(Dense(60, activation='relu', input_shape=(784, )))
model.add(Dropout(0.1))
model.add(Dense(40, activation='relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
fit = model.fit(X_train, y_train, epochs=20, batch_size=256, verbose=1)

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

```
Test score: 0.07730352643054211
Test accuracy: 0.977
```

Problem E

```
# Problem E

# Create a model
model = Sequential()
model.add(Dense(120, activation='relu', input_shape=(784, )))
model.add(Dropout(0.2))
model.add(Dense(80, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
fit = model.fit(X_train, y_train, epochs=20, batch_size=256, verbose=1)

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

```
Test score: 0.06497625442128774
Test accuracy: 0.982
```

Problem F

```
# Problem F

# Create a model
model = Sequential()
model.add(Dense(400, activation='relu', input_shape=(784, )))
model.add(Dropout(0.5))
model.add(Dense(400, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(100, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
fit = model.fit(X_train, y_train, epochs=20, batch_size=256, verbose=1)

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

```
Test score: 0.06377804316383917
Test accuracy: 0.9843
```

3 Convolutional Neural Networks

Problem A

A benefit of the zero-padding scheme is that we preserve the input spatial size in the output feature map, which allows for a deep neural network without the spatial size decreasing too quickly. A drawback would be that we would be wasting computational resources by processing the padding elements.

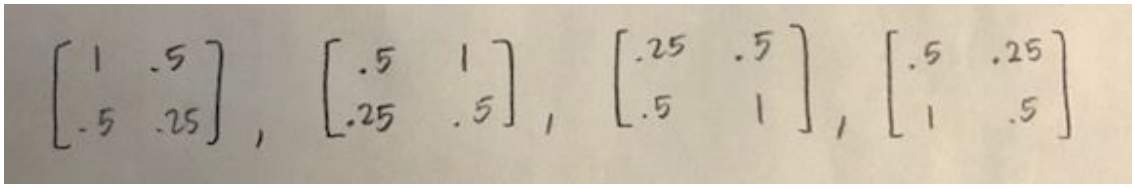
Problem B

The number of parameters in this layer is 608 (number of weights 5 * 5 * 3 + 1 bias term for each of the 8 filters).

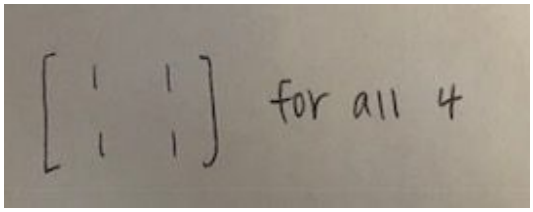
Problem C

The shape of the output tensor is 28 x 28 x 8.

Problem D


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

Problem E



Problem F

Using pooling, the outcome of learning will not be too sensitive on individual pixels, so some of the pixels containing noise will have a minimal impact on learning. The model will learn based on patches when pooling is used, and as long as most of the pixels in the patch are not distorted (such that max stays the same, average is only slightly affected), fitting the noise could be prevented or reduced.

Problem G
Code is submitted on Moodle.

Dropout probability	Test accuracy after training for 1 epoch
0.0	0.9806
0.1	0.9758
0.2	0.9719
0.3	0.9679
0.4	0.9674
0.5	0.9592
0.6	0.9468
0.7	0.9059
0.8	0.8795
0.9	0.1135

Using dropout probability = 0.1, the test accuracy for the final model after training for 10 epochs was 0.9914.

The most effect strategy in designing this convolutional network was following the general CNN design principles (batch norm after a convolutional layer and immediately prior to an activation layer) and using max-pooling to reduce the size of the representation). Testing the parameters for dropout probabilities using validation accuracy after 1 epoch was a very quick and intuitive way to choose the best dropout probability. After the model was finalized, I trained for 10 epochs and observed overfitting (validation error went up and training error went down), so I solved it by adding an extra batch norm layer. Batch norm was the most effective regularization method, as it immediately solved the overfitting problem that I observed. Dropout was not that effective, as we can see that the model performs the best with 0 dropout probability. Layerwise regularization also did not help, most likely due to the MNIST dataset being simple.

The problem with designing our model by validating our hyperparameters this way is that we may overfit the validation data, because we're purposely picking the hyperparameters that perform the best on the specific set of validation data.