Open in Colab

T81-558: Applications of Deep Neural Networks

Module 3: Introduction to TensorFlow

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 3 Material

- Part 3.1: Deep Learning and Neural Network Introduction [Video] [Notebook]
- Part 3.2: Introduction to Tensorflow and Keras [Video] [Notebook]
- Part 3.3: Saving and Loading a Keras Neural Network [Video] [Notebook]
- Part 3.4: Early Stopping in Keras to Prevent Overfitting [Video] [Notebook]
- **Part 3.5: Extracting Weights and Manual Calculation** [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
try:
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

Part 3.5: Extracting Weights and Manual Network Calculation

Weight Initialization

The weights of a neural network determine the output for the neural network. The training process can adjust these weights, so the neural network produces

useful output. Most neural network training algorithms begin by initializing the weights to a random state. Training then progresses through iterations that continuously improve the weights to produce better output.

The random weights of a neural network impact how well that neural network can be trained. If a neural network fails to train, you can remedy the problem by simply restarting with a new set of random weights. However, this solution can be frustrating when you are experimenting with the architecture of a neural network and trying different combinations of hidden layers and neurons. If you add a new layer, and the network's performance improves, you must ask yourself if this improvement resulted from the new layer or from a new set of weights. Because of this uncertainty, we look for two key attributes in a weight initialization algorithm:

- How consistently does this algorithm provide good weights?
- How much of an advantage do the weights of the algorithm provide?

One of the most common yet least practical approaches to weight initialization is to set the weights to random values within a specific range. Numbers between -1 and +1 or -5 and +5 are often the choice. If you want to ensure that you get the same set of random weights each time, you should use a seed. The seed specifies a set of predefined random weights to use. For example, a seed of 1000 might produce random weights of 0.5, 0.75, and 0.2. These values are still random; you cannot predict them, yet you will always get these values when you choose a seed of 1000. Not all seeds are created equal. One problem with random weight initialization is that the random weights created by some seeds are much more difficult to train than others. The weights can be so bad that training is impossible. If you cannot train a neural network with a particular weight set, you should generate a new set of weights using a different seed.

Because weight initialization is a problem, considerable research has been around it. By default, Keras uses the Xavier weight initialization algorithm, introduced in 2006 by Glorot & Bengio[Cite:glorot2010understanding], produces good weights with reasonable consistency. This relatively simple algorithm uses normally distributed random numbers.

To use the Xavier weight initialization, it is necessary to understand that normally distributed random numbers are not the typical random numbers between 0 and 1 that most programming languages generate. Normally distributed random numbers are centered on a mean (μ, mu) that is typically 0. If 0 is the center (mean), then you will get an equal number of random numbers above and below 0. The next question is how far these random numbers will venture from 0. In theory, you could end up with both positive and negative numbers close to the maximum positive and negative ranges supported by your computer. However,

the reality is that you will more likely see random numbers that are between 0 and three standard deviations from the center.

The standard deviation (σ, sigma) parameter specifies the size of this standard deviation. For example, if you specified a standard deviation of 10, you would mainly see random numbers between -30 and +30, and the numbers nearer to 0 have a much higher probability of being selected.

The above figure illustrates that the center, which in this case is 0, will be generated with a 0.4 (40%) probability. Additionally, the probability decreases very quickly beyond -2 or +2 standard deviations. By defining the center and how large the standard deviations are, you can control the range of random numbers that you will receive.

The Xavier weight initialization sets all weights to normally distributed random numbers. These weights are always centered at 0; however, their standard deviation varies depending on how many connections are present for the current layer of weights. Specifically, Equation 4.2 can determine the standard deviation:

$$Var(W) = \frac{2}{n_{in} + n_{out}}$$

The above equation shows how to obtain the variance for all weights. The square root of the variance is the standard deviation. Most random number generators accept a standard deviation rather than a variance. As a result, you usually need to take the square root of the above equation. Figure 3.XAVIER shows how this algorithm might initialize one layer.

Figure 3.XAVIER: Xavier Weight Initialization  Xavier Weight Initialization

We complete this process for each layer in the neural network.

Manual Neural Network Calculation

This section will build a neural network and analyze it down the individual weights. We will train a simple neural network that learns the XOR function. It is not hard to hand-code the neurons to provide an XOR function; however, we will allow Keras for simplicity to train this network for us. The neural network is small, with two inputs, two hidden neurons, and a single output. We will use 100K epochs on the ADAM optimizer. This approach is overkill, but it gets the result, and our focus here is not on tuning.

In [2]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
import numpy as np

# Create a dataset for the XOR function
```

```
x = np.array([
    [0,0],
    [1,0],
    [0,1],
    [1,1]
])

y = np.array([
    0,
    1,
    1,
    0
])

# Build the network
# sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

done = False
cycle = 1

while not done:
    print("Cycle #{}".format(cycle))
    cycle+=1
    model = Sequential()
    model.add(Dense(2, input_dim=2, activation='relu'))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='adam')
    model.fit(x,y,verbose=0,epochs=10000)

    # Predict
    pred = model.predict(x)

    # Check if successful.  It takes several runs with this
    # small of a network
    done = pred[0]<0.01 and pred[3]<0.01 and pred[1] > 0.9 \
        and pred[2] > 0.9
    print(pred)
```

```
Cycle #1
[[0.49999997]
 [0.49999997]
 [0.49999997]
 [0.49999997]]
Cycle #2
[[0.33333334]
 [1.        ]
 [0.33333334]
 [0.33333334]]
Cycle #3
[[0.33333334]
 [1.        ]
 [0.33333334]
 [0.33333334]]
Cycle #4
[[0.]
 [1.]
 [1.]
 [0.]]
```

In [3]:
```
pred[3]
```

Out[3]:
```
array([0.], dtype=float32)
```

The output above should have two numbers near 0.0 for the first and fourth spots (input [0,0] and [1,1]). The middle two numbers should be near 1.0 (input [1,0] and [0,1]). These numbers are in scientific notation. Due to random starting weights, it is sometimes necessary to run the above through several cycles to get a good result.

Now that we've trained the neural network, we can dump the weights.

In [4]:
```
# Dump weights
for layerNum, layer in enumerate(model.layers):
    weights = layer.get_weights()[0]
    biases = layer.get_weights()[1]

    for toNeuronNum, bias in enumerate(biases):
        print(f'{layerNum}B -> L{layerNum+1}N{toNeuronNum}: {bias}')

    for fromNeuronNum, wgt in enumerate(weights):
        for toNeuronNum, wgt2 in enumerate(wgt):
            print(f'L{layerNum}N{fromNeuronNum} \
                  -> L{layerNum+1}N{toNeuronNum} = {wgt2}')
```

```
0B -> L1N0: 1.3025760914331386e-08
0B -> L1N1: -1.4192625741316078e-08
L0N0                    -> L1N0 = 0.659289538860321
L0N0                    -> L1N1 = -0.9533336758613586
L0N1                    -> L1N0 = -0.659289538860321
L0N1                    -> L1N1 = 0.9533336758613586
1B -> L2N0: -1.9757269598130733e-08
L1N0                    -> L2N0 = 1.5167843103408813
L1N1                    -> L2N0 = 1.0489506721496582
```

If you rerun this, you probably get different weights. There are many ways to solve the XOR function.

In the next section, we copy/paste the weights from above and recreate the calculations done by the neural network. Because weights can change with each training, the weights used for the below code came from this:

```
0B -> L1N0: -1.2913415431976318
0B -> L1N1: -3.021530048386012e-08
L0N0 -> L1N0 = 1.2913416624069214
L0N0 -> L1N1 = 1.1912699937820435
L0N1 -> L1N0 = 1.2913411855697632
L0N1 -> L1N1 = 1.1912697553634644
1B -> L2N0: 7.626241297587034e-36
L1N0 -> L2N0 = -1.548777461051941
L1N1 -> L2N0 = 0.8394404649734497
```

In [5]:
```python
input0 = 0
input1 = 1

hidden0Sum = (input0*1.3)+(input1*1.3)+(-1.3)
hidden1Sum = (input0*1.2)+(input1*1.2)+(0)

print(hidden0Sum) # 0
print(hidden1Sum) # 1.2

hidden0 = max(0,hidden0Sum)
hidden1 = max(0,hidden1Sum)

print(hidden0) # 0
print(hidden1) # 1.2

outputSum = (hidden0*-1.6)+(hidden1*0.8)+(0)
print(outputSum) # 0.96

output = max(0,outputSum)

print(output) # 0.96
```

```
0.0
1.2
0
1.2
0.96
0.96
```

In [ ]: