Icon

Description automatically generated

**Multilayer Neural Networks**

In this lesson, you'll learn how to build multilayer neural networks with TensorFlow. Adding a hidden layer to a network allows it to model more complex functions. Also, using a non-linear activation function on the hidden layer lets it model non-linear functions.

Next, you'll see how a ReLU hidden layer is implemented in TensorFlow.

**Note**: Depicted above is a "2-layer" neural network:

1. The first layer effectively consists of the set of weights and biases applied to X and passed through ReLUs. The output of this layer is fed to the next one, but is not observable outside the network, hence it is known as a *hidden layer*.
2. The second layer consists of the weights and biases applied to these intermediate outputs, followed by the softmax function to generate probabilities.

A Rectified linear unit (ReLU) is type of [**activation function**](https://en.wikipedia.org/wiki/Activation_function) that is defined as f(x) = max(0, x). The function returns 0 if x is negative, otherwise it returns x. TensorFlow provides the ReLU function as [**tf.nn.relu()**](https://www.tensorflow.org/api_docs/python/tf/nn/relu), as shown below.

*# Hidden Layer with ReLU activation function*

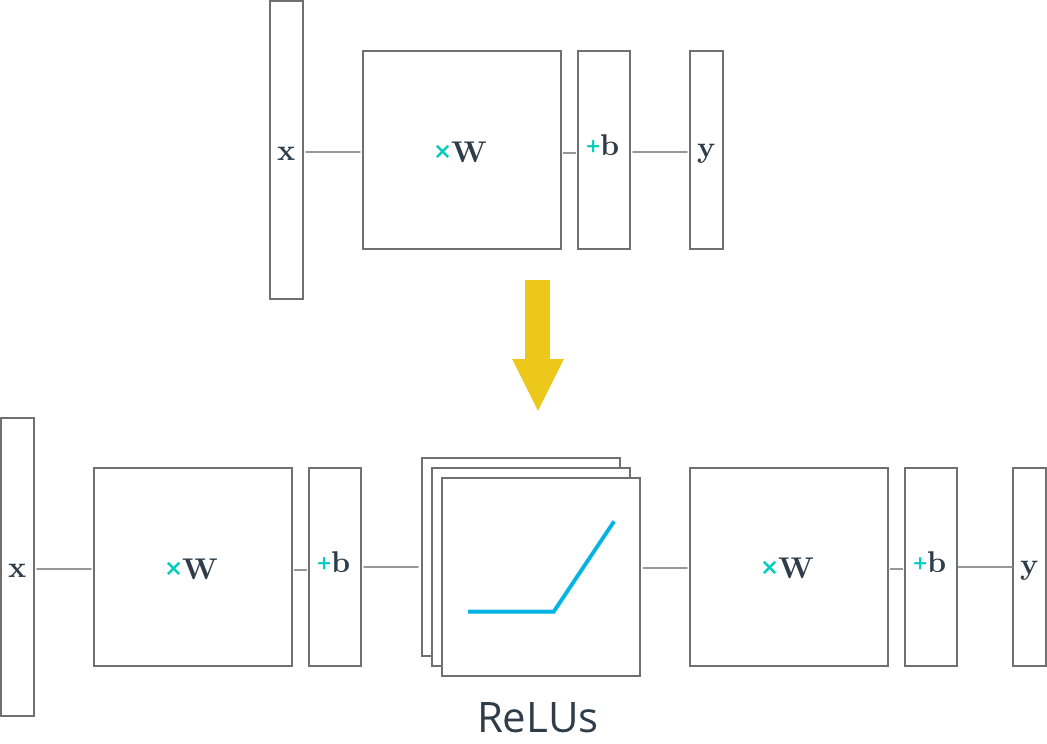
hidden\_layer = tf.add(tf.matmul(features, hidden\_weights), hidden\_biases)

hidden\_layer = tf.nn.relu(hidden\_layer)

output = tf.add(tf.matmul(hidden\_layer, output\_weights), output\_biases)

The above code applies the [**tf.nn.relu()**](https://www.tensorflow.org/api_docs/python/tf/nn/relu) function to the hidden\_layer, effectively turning off any negative weights and acting like an on/off switch. Adding additional layers, like the output layer, after an activation function turns the model into a nonlinear function. This nonlinearity allows the network to solve more complex problems.

## Quiz



In this quiz, you'll use TensorFlow's ReLU function to turn the linear model below into a nonlinear model.

# Deep Neural Network in TensorFlow

You've seen how to build a logistic classifier using TensorFlow. Now you're going to see how to use the logistic classifier to build a deep neural network.

## Step by Step

In the following walkthrough, we'll step through TensorFlow code written to classify the letters in the MNIST database. If you would like to run the network on your computer, the file is provided [**here**](https://d17h27t6h515a5.cloudfront.net/topher/2017/February/58a61a3a_multilayer-perceptron/multilayer-perceptron.zip). You can find this and many more examples of TensorFlow at [**Aymeric Damien's GitHub repository**](https://github.com/aymericdamien/TensorFlow-Examples).

## Code

### TensorFlow MNIST

**from** tensorflow.examples.tutorials.mnist **import** input\_data

mnist = input\_data.read\_data\_sets(".", one\_hot=**True**, reshape=**False**)

You'll use the MNIST dataset provided by TensorFlow, which batches and One-Hot encodes the data for you.

### Learning Parameters

**import** tensorflow **as** tf

*# Parameters*

learning\_rate = 0.001

training\_epochs = 20

batch\_size = 128 *# Decrease batch size if you don't have enough memory*

display\_step = 1

n\_input = 784 *# MNIST data input (img shape: 28\*28)*

n\_classes = 10 *# MNIST total classes (0-9 digits)*

The focus here is on the architecture of multilayer neural networks, not parameter tuning, so here we'll just give you the learning parameters.

### Hidden Layer Parameters

n\_hidden\_layer = 256 *# layer number of features*

The variable n\_hidden\_layer determines the size of the hidden layer in the neural network. This is also known as the width of a layer.

### Weights and Biases

*# Store layers weight & bias*

weights = {

'hidden\_layer': tf.Variable(tf.random\_normal([n\_input, n\_hidden\_layer])),

'out': tf.Variable(tf.random\_normal([n\_hidden\_layer, n\_classes]))

}

biases = {

'hidden\_layer': tf.Variable(tf.random\_normal([n\_hidden\_layer])),

'out': tf.Variable(tf.random\_normal([n\_classes]))

}

Deep neural networks use multiple layers with each layer requiring it's own weight and bias. The 'hidden\_layer' weight and bias is for the hidden layer. The 'out' weight and bias is for the output layer. If the neural network were deeper, there would be weights and biases for each additional layer.

### Input

*# tf Graph input*

x = tf.placeholder("float", [**None**, 28, 28, 1])

y = tf.placeholder("float", [**None**, n\_classes])

x\_flat = tf.reshape(x, [-1, n\_input])

The MNIST data is made up of 28px by 28px images with a single [**channel**](https://en.wikipedia.org/wiki/Channel_(digital_image%29). The [**tf.reshape()**](https://www.tensorflow.org/versions/master/api_docs/python/tf/reshape) function above reshapes the 28px by 28px matrices in x into row vectors of 784px.

### Multilayer Perceptron

Diagram

Description automatically generated

*# Hidden layer with RELU activation*

layer\_1 = tf.add(tf.matmul(x\_flat, weights['hidden\_layer']),\

biases['hidden\_layer'])

layer\_1 = tf.nn.relu(layer\_1)

*# Output layer with linear activation*

logits = tf.add(tf.matmul(layer\_1, weights['out']), biases['out'])

You've seen the linear function tf.add(tf.matmul(x\_flat, weights['hidden\_layer']), biases['hidden\_layer']) before, also known as xw + b. Combining linear functions together using a ReLU will give you a two layer network.

### Optimizer

*# Define loss and optimizer*

cost = tf.reduce\_mean(\

tf.nn.softmax\_cross\_entropy\_with\_logits(logits=logits, labels=y))

optimizer = tf.train.GradientDescentOptimizer(learning\_rate=learning\_rate)\

.minimize(cost)

This is the same optimization technique used in the Intro to TensorFLow lab.

### Session

*# Initializing the variables*

init = tf.global\_variables\_initializer()

*# Launch the graph*

**with** tf.Session() **as** sess:

sess.run(init)

*# Training cycle*

**for** epoch **in** range(training\_epochs):

total\_batch = int(mnist.train.num\_examples/batch\_size)

*# Loop over all batches*

**for** i **in** range(total\_batch):

batch\_x, batch\_y = mnist.train.next\_batch(batch\_size)

*# Run optimization op (backprop) and cost op (to get loss value)*

sess.run(optimizer, feed\_dict={x: batch\_x, y: batch\_y})

The MNIST library in TensorFlow provides the ability to receive the dataset in batches. Calling the mnist.train.next\_batch() function returns a subset of the training data.

## Deeper Neural Network

A picture containing clock, drawing

Description automatically generated

That's it! Going from one layer to two is easy. Adding more layers to the network allows you to solve more complicated problems.

# Save and Restore TensorFlow Models

Training a model can take hours. But once you close your TensorFlow session, you lose all the trained weights and biases. If you were to reuse the model in the future, you would have to train it all over again!

Fortunately, TensorFlow gives you the ability to save your progress using a class called [**tf.train.Saver**](https://www.tensorflow.org/api_docs/python/tf/train/Saver). This class provides the functionality to save any [**tf.Variable**](https://www.tensorflow.org/api_docs/python/tf/Variable) to your file system.

## Saving Variables

Let's start with a simple example of saving weights and bias Tensors. For the first example you'll just save two variables. Later examples will save all the weights in a practical model.

**import** tensorflow **as** tf

*# The file path to save the data*

save\_file = './model.ckpt'

*# Two Tensor Variables: weights and bias*

weights = tf.Variable(tf.truncated\_normal([2, 3]))

bias = tf.Variable(tf.truncated\_normal([3]))

*# Class used to save and/or restore Tensor Variables*

saver = tf.train.Saver()

**with** tf.Session() **as** sess:

*# Initialize all the Variables*

sess.run(tf.global\_variables\_initializer())

*# Show the values of weights and bias*

print('Weights:')

print(sess.run(weights))

print('Bias:')

print(sess.run(bias))

*# Save the model*

saver.save(sess, save\_file)

Weights:

[[-0.97990924 1.03016174 0.74119264]

[-0.82581609 -0.07361362 -0.86653847]]

Bias:

[ 1.62978125 -0.37812829 0.64723819]

The Tensors weights and bias are set to random values using the [**tf.truncated\_normal()**](https://www.tensorflow.org/api_docs/python/tf/truncated_normal) function. The values are then saved to the save\_file location, "model.ckpt", using the [**tf.train.Saver.save()**](https://www.tensorflow.org/api_docs/python/tf/train/Saver#save) function. (The ".ckpt" extension stands for "checkpoint".)

If you're using TensorFlow 0.11.0RC1 or newer, a file called "model.ckpt.meta" will also be created. This file contains the TensorFlow graph.

## Loading Variables

Now that the Tensor Variables are saved, let's load them back into a new model.

*# Remove the previous weights and bias*

tf.reset\_default\_graph()

*# Two Variables: weights and bias*

weights = tf.Variable(tf.truncated\_normal([2, 3]))

bias = tf.Variable(tf.truncated\_normal([3]))

*# Class used to save and/or restore Tensor Variables*

saver = tf.train.Saver()

**with** tf.Session() **as** sess:

*# Load the weights and bias*

saver.restore(sess, save\_file)

*# Show the values of weights and bias*

print('Weight:')

print(sess.run(weights))

print('Bias:')

print(sess.run(bias))

Weights:

[[-0.97990924 1.03016174 0.74119264]

[-0.82581609 -0.07361362 -0.86653847]]

Bias:

[ 1.62978125 -0.37812829 0.64723819]

You'll notice you still need to create the weights and bias Tensors in Python. The [**tf.train.Saver.restore()**](https://www.tensorflow.org/api_docs/python/tf/train/Saver#restore) function loads the saved data into weights and bias.

Since [**tf.train.Saver.restore()**](https://www.tensorflow.org/api_docs/python/tf/train/Saver#restore) sets all the TensorFlow Variables, you don't need to call [**tf.global\_variables\_initializer()**](https://www.tensorflow.org/api_docs/python/tf/global_variables_initializer).

## Save a Trained Model

Let's see how to train a model and save its weights.

First start with a model:

*# Remove previous Tensors and Operations*

tf.reset\_default\_graph()

**from** tensorflow.examples.tutorials.mnist **import** input\_data

**import** numpy **as** np

learning\_rate = 0.001

n\_input = 784 *# MNIST data input (img shape: 28\*28)*

n\_classes = 10 *# MNIST total classes (0-9 digits)*

*# Import MNIST data*

mnist = input\_data.read\_data\_sets('.', one\_hot=**True**)

*# Features and Labels*

features = tf.placeholder(tf.float32, [**None**, n\_input])

labels = tf.placeholder(tf.float32, [**None**, n\_classes])

*# Weights & bias*

weights = tf.Variable(tf.random\_normal([n\_input, n\_classes]))

bias = tf.Variable(tf.random\_normal([n\_classes]))

*# Logits - xW + b*

logits = tf.add(tf.matmul(features, weights), bias)

*# Define loss and optimizer*

cost = tf.reduce\_mean(\

tf.nn.softmax\_cross\_entropy\_with\_logits(logits=logits, labels=labels))

optimizer = tf.train.GradientDescentOptimizer(learning\_rate=learning\_rate)\

.minimize(cost)

*# Calculate accuracy*

correct\_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(labels, 1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

Let's train that model, then save the weights:

**import** math

save\_file = './train\_model.ckpt'

batch\_size = 128

n\_epochs = 100

saver = tf.train.Saver()

*# Launch the graph*

**with** tf.Session() **as** sess:

sess.run(tf.global\_variables\_initializer())

*# Training cycle*

**for** epoch **in** range(n\_epochs):

total\_batch = math.ceil(mnist.train.num\_examples / batch\_size)

*# Loop over all batches*

**for** i **in** range(total\_batch):

batch\_features, batch\_labels = mnist.train.next\_batch(batch\_size)

sess.run(

optimizer,

feed\_dict={features: batch\_features, labels: batch\_labels})

*# Print status for every 10 epochs*

**if** epoch % 10 == 0:

valid\_accuracy = sess.run(

accuracy,

feed\_dict={

features: mnist.validation.images,

labels: mnist.validation.labels})

print('Epoch {:<3} - Validation Accuracy: {}'.format(

epoch,

valid\_accuracy))

*# Save the model*

saver.save(sess, save\_file)

print('Trained Model Saved.')

Epoch 0 - Validation Accuracy: 0.06859999895095825

Epoch 10 - Validation Accuracy: 0.20239999890327454

Epoch 20 - Validation Accuracy: 0.36980000138282776

Epoch 30 - Validation Accuracy: 0.48820000886917114

Epoch 40 - Validation Accuracy: 0.5601999759674072

Epoch 50 - Validation Accuracy: 0.6097999811172485

Epoch 60 - Validation Accuracy: 0.6425999999046326

Epoch 70 - Validation Accuracy: 0.6733999848365784

Epoch 80 - Validation Accuracy: 0.6916000247001648

Epoch 90 - Validation Accuracy: 0.7113999724388123

Trained Model Saved.

## Load a Trained Model

Let's load the weights and bias from memory, then check the test accuracy.

saver = tf.train.Saver()

*# Launch the graph*

**with** tf.Session() **as** sess:

saver.restore(sess, save\_file)

test\_accuracy = sess.run(

accuracy,

feed\_dict={features: mnist.test.images, labels: mnist.test.labels})

print('Test Accuracy: {}'.format(test\_accuracy))

Test Accuracy: 0.7229999899864197

That's it! You now know how to save and load a trained model in TensorFlow. Let's look at loading weights and biases into modified models in the next section.

# Loading the Weights and Biases into a New Model

Sometimes you might want to adjust, or "finetune" a model that you have already trained and saved.

However, loading saved Variables directly into a modified model can generate errors. Let's go over how to avoid these problems.

## Naming Error

TensorFlow uses a string identifier for Tensors and Operations called name. If a name is not given, TensorFlow will create one automatically. TensorFlow will give the first node the name <Type>, and then give the name <Type>\_<number> for the subsequent nodes. Let's see how this can affect loading a model with a different order of weights and bias:

**import** tensorflow **as** tf

*# Remove the previous weights and bias*

tf.reset\_default\_graph()

save\_file = 'model.ckpt'

*# Two Tensor Variables: weights and bias*

weights = tf.Variable(tf.truncated\_normal([2, 3]))

bias = tf.Variable(tf.truncated\_normal([3]))

saver = tf.train.Saver()

*# Print the name of Weights and Bias*

print('Save Weights: {}'.format(weights.name))

print('Save Bias: {}'.format(bias.name))

**with** tf.Session() **as** sess:

sess.run(tf.global\_variables\_initializer())

saver.save(sess, save\_file)

*# Remove the previous weights and bias*

tf.reset\_default\_graph()

*# Two Variables: weights and bias*

bias = tf.Variable(tf.truncated\_normal([3]))

weights = tf.Variable(tf.truncated\_normal([2, 3]))

saver = tf.train.Saver()

*# Print the name of Weights and Bias*

print('Load Weights: {}'.format(weights.name))

print('Load Bias: {}'.format(bias.name))

**with** tf.Session() **as** sess:

*# Load the weights and bias - ERROR*

saver.restore(sess, save\_file)

The code above prints out the following:

Save Weights: Variable:0

Save Bias: Variable\_1:0

Load Weights: Variable\_1:0

Load Bias: Variable:0

...

InvalidArgumentError (see above for traceback): Assign requires shapes of both tensors to match.

...

You'll notice that the name properties for weights and bias are different than when you saved the model. This is why the code produces the "Assign requires shapes of both tensors to match" error. The code saver.restore(sess, save\_file) is trying to load weight data into bias and bias data into weights.

Instead of letting TensorFlow set the name property, let's set it manually:

**import** tensorflow **as** tf

tf.reset\_default\_graph()

save\_file = 'model.ckpt'

*# Two Tensor Variables: weights and bias*

weights = tf.Variable(tf.truncated\_normal([2, 3]), name='weights\_0')

bias = tf.Variable(tf.truncated\_normal([3]), name='bias\_0')

saver = tf.train.Saver()

*# Print the name of Weights and Bias*

print('Save Weights: {}'.format(weights.name))

print('Save Bias: {}'.format(bias.name))

**with** tf.Session() **as** sess:

sess.run(tf.global\_variables\_initializer())

saver.save(sess, save\_file)

*# Remove the previous weights and bias*

tf.reset\_default\_graph()

*# Two Variables: weights and bias*

bias = tf.Variable(tf.truncated\_normal([3]), name='bias\_0')

weights = tf.Variable(tf.truncated\_normal([2, 3]) ,name='weights\_0')

saver = tf.train.Saver()

*# Print the name of Weights and Bias*

print('Load Weights: {}'.format(weights.name))

print('Load Bias: {}'.format(bias.name))

**with** tf.Session() **as** sess:

*# Load the weights and bias - No Error*

saver.restore(sess, save\_file)

print('Loaded Weights and Bias successfully.')

Save Weights: weights\_0:0

Save Bias: bias\_0:0

Load Weights: weights\_0:0

Load Bias: bias\_0:0

Loaded Weights and Bias successfully.

That worked! The Tensor names match and the data loaded correctly.

Graphical user interface, text

Description automatically generated

# TensorFlow Dropout

Diagram

Description automatically generated

Figure 1: Taken from the paper "Dropout: A Simple Way to Prevent Neural Networks from

Dropout is a regularization technique for reducing overfitting. The technique temporarily drops units ([**artificial neurons**](https://en.wikipedia.org/wiki/Artificial_neuron)) from the network, along with all of those units' incoming and outgoing connections. Figure 1 illustrates how dropout works.

TensorFlow provides the [**tf.nn.dropout()**](https://www.tensorflow.org/api_docs/python/tf/nn/dropout) function, which you can use to implement dropout.

Let's look at an example of how to use [**tf.nn.dropout()**](https://www.tensorflow.org/api_docs/python/tf/nn/dropout).

keep\_prob = tf.placeholder(tf.float32) *# probability to keep units*

hidden\_layer = tf.add(tf.matmul(features, weights[0]), biases[0])

hidden\_layer = tf.nn.relu(hidden\_layer)

hidden\_layer = tf.nn.dropout(hidden\_layer, keep\_prob)

logits = tf.add(tf.matmul(hidden\_layer, weights[1]), biases[1])

The code above illustrates how to apply dropout to a neural network.

The [**tf.nn.dropout()**](https://www.tensorflow.org/api_docs/python/tf/nn/dropout) function takes in two parameters:

1. hidden\_layer: the tensor to which you would like to apply dropout
2. keep\_prob: the probability of keeping (i.e. not dropping) any given unit

keep\_prob allows you to adjust the number of units to drop. In order to compensate for dropped units, [**tf.nn.dropout()**](https://www.tensorflow.org/api_docs/python/tf/nn/dropout) multiplies all units that are kept (i.e. not dropped) by 1/keep\_prob.

During training, a good starting value for keep\_prob is 0.5.

During testing, use a keep\_prob value of 1.0 to keep all units and maximize the power of the model.

## Quiz 1

Take a look at the code snippet below. Do you see what's wrong?

There's nothing wrong with the syntax, however the test accuracy is extremely low.

...

keep\_prob = tf.placeholder(tf.float32) *# probability to keep units*

hidden\_layer = tf.add(tf.matmul(features, weights[0]), biases[0])

hidden\_layer = tf.nn.relu(hidden\_layer)

hidden\_layer = tf.nn.dropout(hidden\_layer, keep\_prob)

logits = tf.add(tf.matmul(hidden\_layer, weights[1]), biases[1])

...

**with** tf.Session() **as** sess:

sess.run(tf.global\_variables\_initializer())

**for** epoch\_i **in** range(epochs):

**for** batch\_i **in** range(batches):

....

sess.run(optimizer, feed\_dict={

features: batch\_features,

labels: batch\_labels,

keep\_prob: 0.5})

validation\_accuracy = sess.run(accuracy, feed\_dict={

features: test\_features,

labels: test\_labels,

keep\_prob: 0.5})

Graphical user interface, text, application, email

Description automatically generated