2017

TensorFlow

Thanks to

TENSORFLOW IN A NUTSHELL!

ADAPTED FROM UDACITY SELF-DRIVING CAR ENGINEERING NANODEGREE CLASS

CHRISTOS KYRKOU, PHD

Install

OS X or Linux

Prerequisites

Intro to TensorFlow requires **Python 3.4 or higher** and **Anaconda**. If you don't meet all of these requirements, please install the appropriate package(s).

Install TensorFlow

You're going to use an Anaconda environment. If you're unfamiliar with Anaconda environments, check out the **official documentation**.
Run the following commands to setup your environment:

```
conda create --name=IntroToTensorFlow python=3 anaconda
source activate IntroToTensorFlow
conda install -c conda-forge tensorflow
```

That's it! You have a working environment with TensorFlow. Test it out with the code in the *Hello, world!* section below.

Windows

Install Docker

Download and install Docker from the **official Docker website**.

Run the Docker Container

Run the command below to start a jupyter notebook server with TensorFlow:

```
docker run -it -p 8888:8888 gcr.io/tensorflow/tensorflow
```

Users in China should use the `b.gcr.io/tensorflow/tensorflow` instead of `gcr.io/tensorflow/tensorflow`

You can access the jupyter notebook at **localhost:8888**. The server includes 3 examples of TensorFlow notebooks, but you can create a new notebook to test all your code.

Hello, world!

Try running the following code in your Python console to make sure you have TensorFlow properly installed. The console will print "Hello, world!" if TensorFlow is installed. Don’t worry about understanding what it does. You’ll learn about it in the next section.

```
import tensorflow as tf

# Create TensorFlow object called tensor
hello_constant = tf.constant('Hello World!')

with tf.Session() as sess:
    # Run the tf.constant operation in the session
    output = sess.run(hello_constant)
    print(output)
```

Errors

If you're getting the error `tensorflow.python.framework.errors.InvalidArgumentError: Placeholder:0 is both fed and fetched`, you're running an older version of TensorFlow. Uninstall TensorFlow, and reinstall it using the instructions above. For more solutions, check out the **Common Problems** section.

Hello, Tensor World!

Let's analyze the Hello World script you ran. For reference, I've added the code below.

```
import tensorflow as tf

# Create TensorFlow object called hello_constant
hello_constant = tf.constant('Hello World!')

with tf.Session() as sess:
    # Run the tf.constant operation in the session
    output = sess.run(hello_constant)
    print(output)
```

Tensor

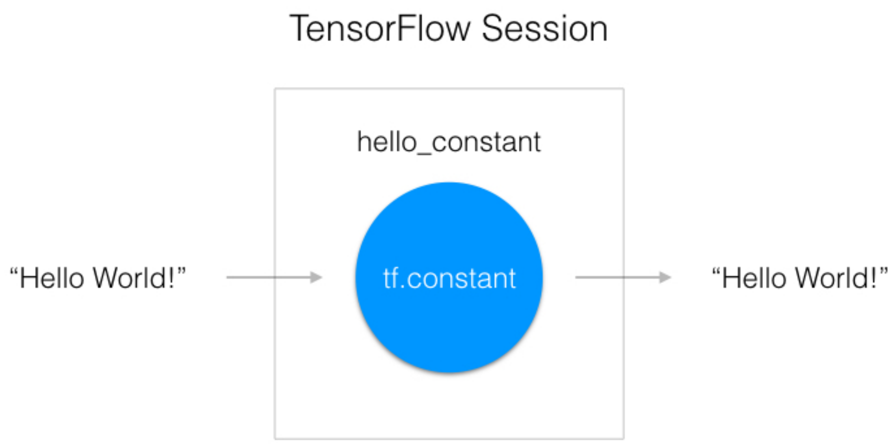
In TensorFlow, data isn't stored as integers, floats, or strings. These values are encapsulated in an object called a tensor. In the case of `hello_constant = tf.constant('Hello World!')`, `hello_constant` is a 0-dimensional string tensor, but tensors come in a variety of sizes as shown below:

```
# A is a 0-dimensional int32 tensor
A = tf.constant(1234)
# B is a 1-dimensional int32 tensor
B = tf.constant([ [123,456,789] ])
 # C is a 2-dimensional int32 tensor
C = tf.constant([ [123,456,789], [222,333,444] ])
```

`tf.constant()` is one of many TensorFlow operations you will use in this lesson. The tensor returned by `tf.constant()` is called a constant tensor, because the value of the tensor never changes.

Session

TensorFlow's api is built around the idea of a computational graph, a way of visualizing a mathematical process. Let's take the TensorFlow code you ran and turn that into a graph:



A "TensorFlow Session", as shown above, is an environment for running a graph. The session is in charge of allocating the operations to GPU(s) and/or CPU(s), including remote machines. Let's see how you use it.

```
with tf.Session() as sess:
    output = sess.run(hello_constant)
```

The code has already created the tensor, `hello_constant`, from the previous lines. The next step is to evaluate the tensor in a session.
The code creates a session instance, `sess`, using **`tf.Session`**. The **`sess.run()`** function then evaluates the tensor and returns the results.

Input

In the last section, you passed a tensor into a session and it returned the result. What if you want to use a non tensor? This is where **`tf.placeholder()`** and `feed_dict` come into place. In this section, you'll go over the basics of feeding data into TensorFlow.

tf.placeholder()

Sadly you can't just set `x` to your dataset and put it in TensorFlow, because over time you'll want your TensorFlow model to take in different datasets with different parameters. You need **`tf.placeholder()`**!
`tf.placeholder()` returns a tensor that gets its value from data passed to the **`tf.session.run()`**function, allowing you to set the input right before the session runs.

Session's feed_dict

```
x = tf.placeholder(tf.string)

with tf.Session() as sess:
    output = sess.run(x, feed_dict={x: 'Hello World'})
```

Use the `feed_dict` parameter in **`tf.session.run()`** to set the placeholder tensor. The above example shows the tensor `x` being set to the string `"Hello, world"`. It's also possible to set more than one tensor using `feed_dict` as shown below.

```
x = tf.placeholder(tf.string)
y = tf.placeholder(tf.int32)
z = tf.placeholder(tf.float32)

with tf.Session() as sess:
    output = sess.run(x, feed_dict={x: 'Test String', y: 123, z: 45.67})
```

Note: If the data passed to the `feed_dict` doesn't match the tensor type and can't be cast into the tensor type, you'll get the error "`ValueError: invalid literal for`...".

TensorFlow Math

Getting the input is great, but now you need to use it. You're going to use basic math functions that everyone knows and loves - add, subtract, multiply, and divide - with tensors. (There's many more math functions you can check out in the documentation.)

Addition

```
x = tf.add(5, 2)  # 7
```

You'll start with the add function. The `tf.add()` function does exactly what you expect it to do. It takes in two numbers, two tensors, or one of each, and returns their sum as a tensor.

Subtraction and Multiplication

Here's an example with subtraction and multiplication.

```
x = tf.sub(10, 4) # 6
y = tf.mul(2, 5)  # 10
```

The `x` tensor will evaluate to `6`, because `10 - 4 = 6`. The `y` tensor will evaluate to `10`, because `2 * 5 = 10`. That was easy!

Weights and Bias in TensorFlow

The goal of training a neural network is to modify weights and biases to best predict the labels. In order to use weights and bias, you'll need a Tensor that can be modified. This leaves out `tf.placeholder()` and `tf.constant()`, since those Tensors can't be modified. This is where `tf.Variable` class comes in.

tf.Variable()

```
x = tf.Variable(5)
```

The `tf.Variable` class creates a tensor with an initial value that can be modified, much like a normal Python variable. This tensor stores its state in the session, so you must initialize the state of the tensor manually. You'll use the `tf.global_variables_initializer()` function to initialize the state of all the Variable tensors.

Initialization

```
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
```

The `tf.global_variables_initializer()` call returns an operation that will initialize all TensorFlow variables from the graph. You call the operation using a session to initialize all the variables as shown above. Using the `tf.Variable` class allows us to change the weights and bias, but an initial value needs to be chosen.

Initializing the weights with random numbers from a normal distribution is good practice. Randomizing the weights helps the model from becoming stuck in the same place every time you train it. You'll learn more about this in the next lesson, when you study gradient descent.

Similarly, choosing weights from a normal distribution prevents any one weight from overwhelming other weights. You'll use the `tf.truncated_normal()` function to generate random numbers from a normal distribution.

tf.truncated_normal()

```
n_features = 120
n_labels = 5
weights = tf.Variable(tf.truncated_normal((n_features, n_labels)))
```

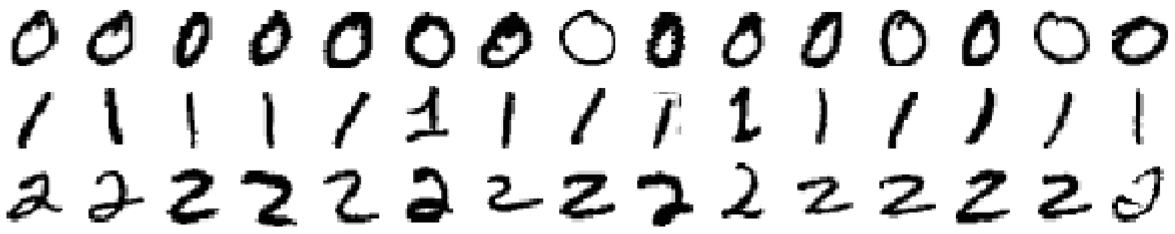
The `tf.truncated_normal()` function returns a tensor with random values from a normal distribution whose magnitude is no more than 2 standard deviations from the mean.
Since the weights are already helping prevent the model from getting stuck, you don't need to randomize the bias. Let's use the simplest solution, setting the bias to 0.

tf.zeros()

```
n_labels = 5
bias = tf.Variable(tf.zeros(n_labels))
```

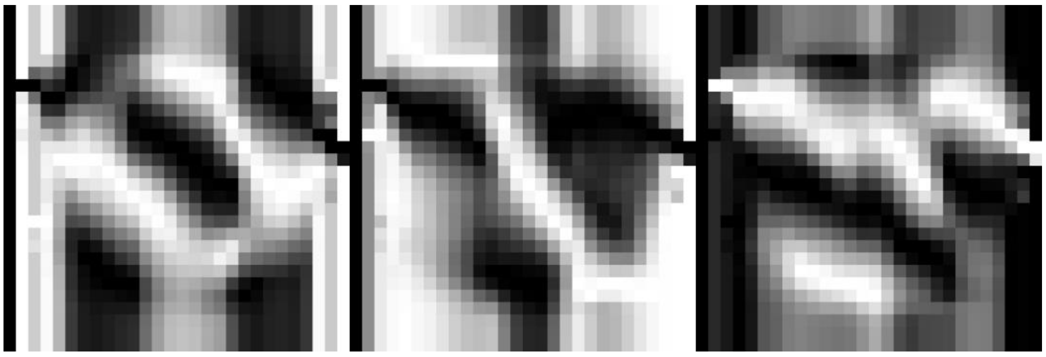
The `tf.zeros()` function returns a tensor with all zeros.

Linear Classifier Quiz



Subset of MNIST dataset.

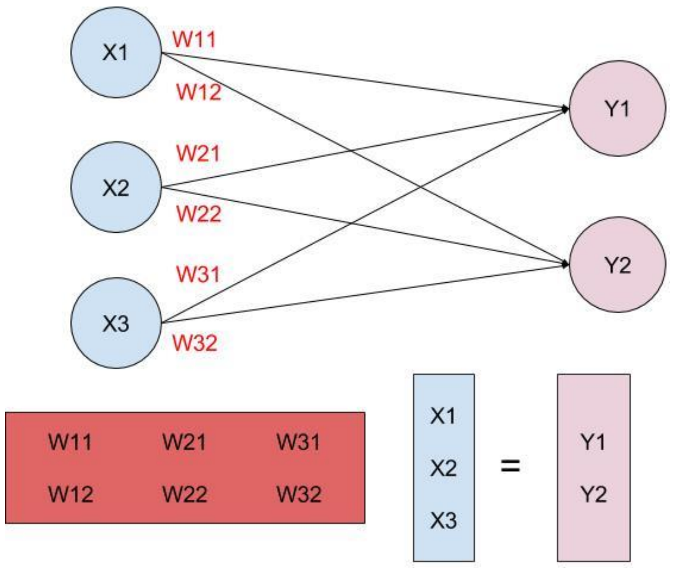
You'll can classify the handwritten numbers `0`, `1`, and `2` from the MNIST dataset using TensorFlow. The above is a small sample of the data you'll be training on. Notice how some of the `1`s are written with a serif at the top and at different angles. The similarities and differences will play a part in shaping the weights of the model.



Left: Weights for labeling 0. Middle: Weights for labeling 1. Right: Weights for labeling 2.

The images above are trained weights for each label (`0`, `1`, and `2`). The weights display the unique properties of each digit they have found.

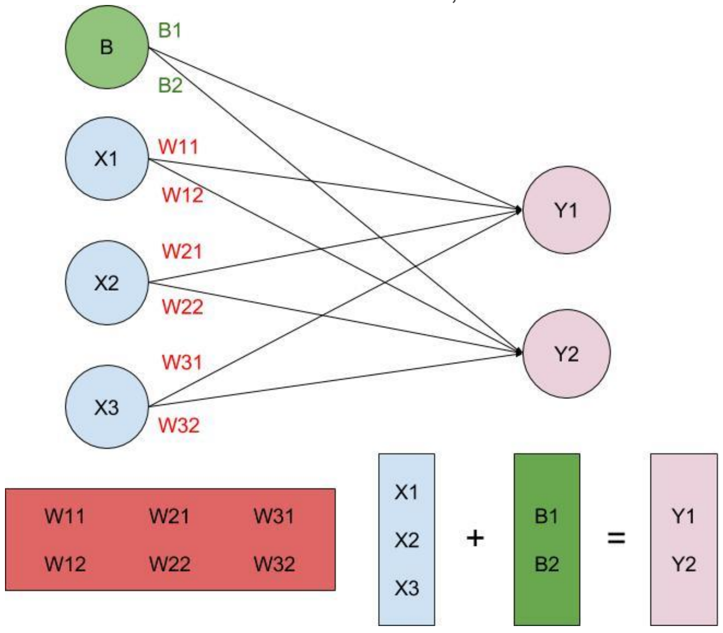
Linear Function



Function y = Wx

Let’s derive the function `y = Wx + b`. We want to translate our input, `x`, to labels, `y`. For example, imagine we want to classify images as digits.

`x` would be our list of pixel values, and `y` would be the logits, one for each digit. Let's take a look at `y = Wx`, where the weights, `W`, determine the influence of `x` at predicting each `y`.
`y = Wx` allows us to segment the data into their respective labels using a line. However, this line has to pass through the origin, because whenever `x` equals 0, then `y` is also going to equal 0.
We want the ability to shift the line away from the origin to fit more complex data. The simplest solution is to add a number to the function, which we call “bias”.



Function y = Wx + b

Our new function becomes `Wx + b`, allowing us to create predictions on linearly separable data. Let’s use a concrete example and calculate the logits.

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Softmax Function

Softmax

The next step is to assign a probability to each label, which you can then use to classify the data. Use the softmax function to turn your logits into probabilities.

In the one dimensional case, the array is just a single set of logits. In the two dimensional case, each column in the array is a set of logits. The `softmax(x)` function should return a NumPy array of the same shape as `x`.

For example, given a one-dimensional array:

```
# logits is a one-dimensional array with 3 elements
logits = [1.0, 2.0, 3.0]
# softmax will return a one-dimensional array with 3 elements
print softmax(logits)
$ [ 0.09003057  0.24472847  0.66524096]
```

Given a two-dimensional array where each column represents a set of logits:

```
# logits is a two-dimensional array
logits = np.array([
    [1, 2, 3, 6],
    [2, 4, 5, 6],
    [3, 8, 7, 6]])
# softmax will return a two-dimensional array with the same shape
print softmax(logits)
$ [
    [ 0.09003057  0.00242826  0.01587624  0.33333333]
    [ 0.24472847  0.01794253  0.11731043  0.33333333]
    [ 0.66524096  0.97962921  0.86681333  0.33333333]
  ]
```

TensorFlow Mini-batching

In order to use mini-batching, you must first divide your data into batches.

Unfortunately, it's sometimes impossible to divide the data into batches of exactly equal size. For example, imagine you'd like to create batches of 128 samples each from a dataset of 1000 samples. Since 128 does not evenly divide into 1000, you'd wind up with 7 batches of 128 samples, and 1 batch of 104 samples. (7*128 + 1*104 = 1000)

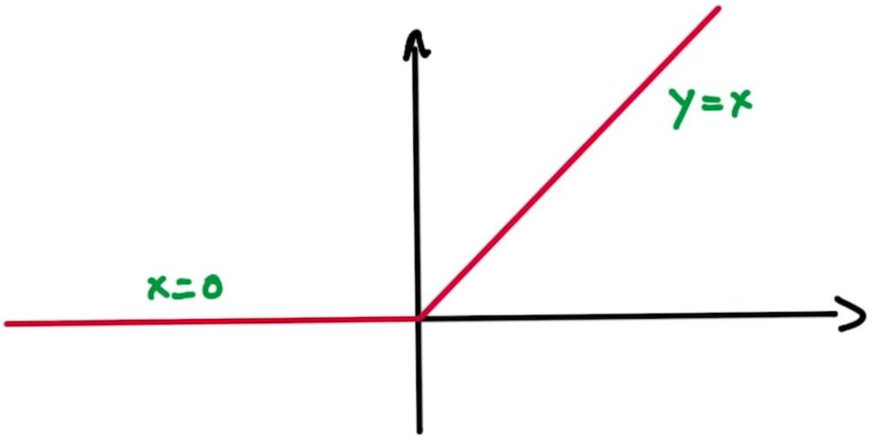
In that case, the size of the batches would vary, so you need to take advantage of TensorFlow's **`tf.placeholder()`** function to receive the varying batch sizes. Continuing the example, if each sample had `n_input = 784` features and `n_classes = 10` possible labels, the dimensions for `features` would be `[None, n_input]` and `labels` would be `[None, n_classes]`.

```
# Features and Labels
features = tf.placeholder(tf.float32, [None, n_input])
labels = tf.placeholder(tf.float32, [None, n_classes])
```

What does `None` do here?

The `None` dimension is a placeholder for the batch size. At runtime, TensorFlow will accept any batch size greater than 0.

TensorFlow ReLUs

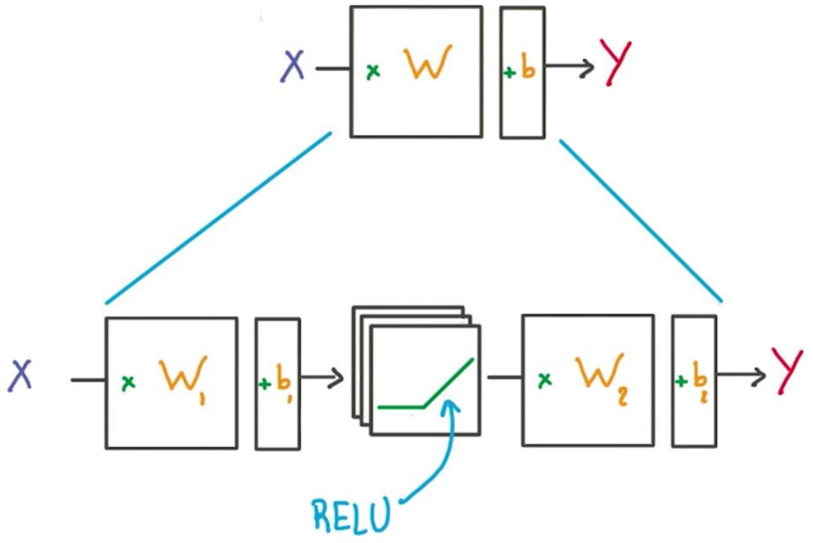


Rectified linear unit (ReLU) graph

A Rectified linear unit (ReLU) is type of 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()`**, 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()`** 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.



Converting a single-layer linear network to a two-layer network with ReLU activation.

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. You can find this and many more examples of TensorFlow at **Aymeric Damien's GitHub repository**.

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]))
}
```

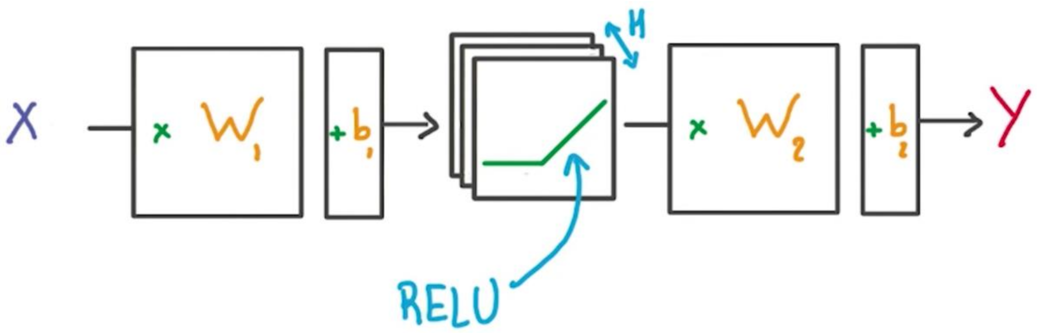
Deep neural networks use multiple layers with each layer requiring it's own weight and bias. The `'hidden_layer'` weight and bias is for the hidden layer. The `'out'` weight and bias is for the output layer. If the neural network were deeper, there would be weights and biases for each additional layer.

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.
The `tf.reshape()` function above reshapes the 28px by 28px matrices in `x` into vectors of 784px by 1px.

Multilayer Perceptron



```
# 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

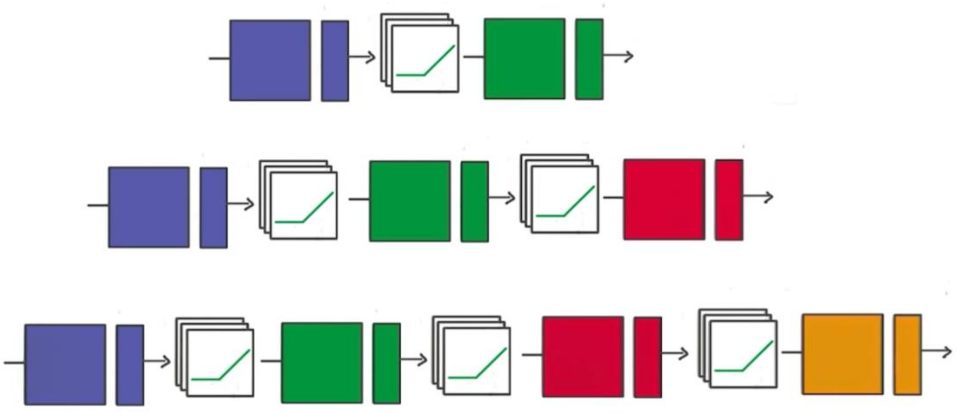
```
# Define loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits, y))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
.minimize(cost)
```

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



That's it! Going from one layer to two is easy. Adding more layers to the network allows you to solve more complicated problems. In the next video, you'll see how changing the number of layers can affect your network.

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`. This class provides the functionality to save any `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()` function. The values are then saved to the `save_file` location, "model.ckpt", using the `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()` function loads the saved data into `weights` and `bias`.
Since `tf.train.Saver.restore()` sets all the TensorFlow Variables, you don't need to call `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, labe
ls))
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_si
ze)
            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.

TensorFlow Dropout

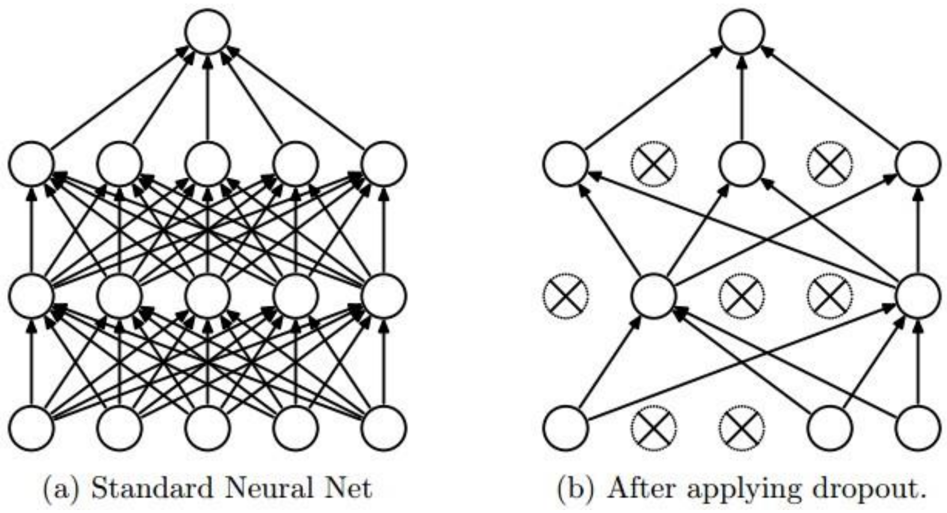


Figure 1: Taken from the paper "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" (https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf)

Dropout is a regularization technique for reducing overfitting. The technique temporarily drops units (artificial neurons) 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()` function, which you can use to implement dropout.

Let's look at an example of how to use `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()` 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()` 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.

```
...

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: 1.0})
```

Dimensionality

From what we've learned so far, how can we calculate the number of neurons of each layer in our CNN?

Given our input layer has a volume of `W`, our filter has a volume (`height * width * depth`) of `F`, we have a stride of `S`, and a padding of `P`, the following formula gives us the volume of the next layer: `(W−F+2P)/S+1`.
Knowing the dimensionality of each additional layer helps us understand how large our model is and how our decisions around filter size and stride affect the size of our network.

```
new_height = (input_height - filter_height + 2 * P)/S + 1
new_width = (input_width - filter_width + 2 * P)/S + 1
```

TensorFlow Convolution Layer

Let's examine how to implement a CNN in TensorFlow.

TensorFlow provides the `tf.nn.conv2d()` and `tf.nn.bias_add()` functions to create your own convolutional layers.

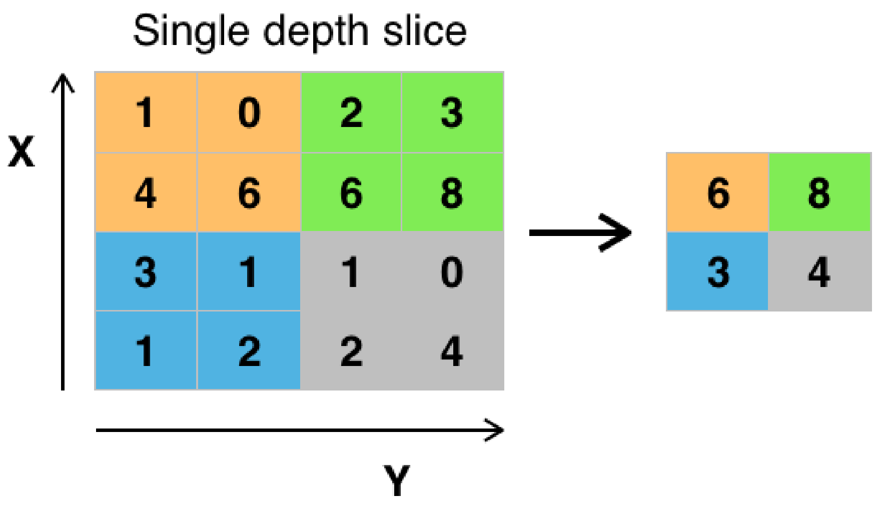
```
# Output depth
k_output = 64

# Image Properties
image_width = 10
image_height = 10
color_channels = 3

# Convolution filter
filter_size_width = 5
filter_size_height = 5

# Input/Image
input = tf.placeholder(
    tf.float32,
    shape=[None, image_width, image_height, color_channels])

# Weight and bias
weight = tf.Variable(tf.truncated_normal(
    [filter_size_width, filter_size_height, color_channels, k_output]))
bias = tf.Variable(tf.zeros(k_output))

# Apply Convolution
conv_layer = tf.nn.conv2d(input, weight, strides=[1, 2, 2, 1], padding='SAM
E')
# Add bias
conv_layer = tf.nn.bias_add(conv_layer, bias)
# Apply activation function
conv_layer = tf.nn.relu(conv_layer)
```

The code above uses the `tf.nn.conv2d()` function to compute the convolution with `weight` as the filter and `[1, 2, 2, 1]` for the strides. TensorFlow uses a stride for each `input` dimension, `[batch, input_height, input_width, input_channels]`. We are generally always going to set the stride for `batch` and `input_channels` (i.e. the first and fourth element in the `strides` array) to be `1`.

You'll focus on changing `input_height` and `input_width` while setting `batch` and `input_channels` to 1. The `input_height` and `input_width` strides are for striding the filter over `input`. This example code uses a stride of 2 with 5x5 filter over `input`.

The `tf.nn.bias_add()` function adds a 1-d bias to the last dimension in a matrix.

TensorFlow Max Pooling



By Aphex34 (Own work) [CC BY-SA 4.0 (http://creativecommons.org/licenses/by-sa/4.0)], via Wikimedia Commons

The image above is an example of max pooling with a 2x2 filter and stride of 2. The four 2x2 colors represent each time the filter was applied to find the maximum value.

For example, `[[1, 0], [4, 6]]` becomes `6`, because `6` is the maximum value in this set. Similarly, `[[2, 3], [6, 8]]` becomes `8`.

Conceptually, the benefit of the max pooling operation is to reduce the size of the input, and allow the neural network to focus on only the most important elements. Max pooling does this by only retaining the maximum value for each filtered area, and removing the remaining values.

TensorFlow provides the `tf.nn.max_pool()` function to apply max pooling to your convolutional layers.

```
...
conv_layer = tf.nn.conv2d(input, weight, strides=[1, 2, 2, 1], padding='SAM
E')
conv_layer = tf.nn.bias_add(conv_layer, bias)
conv_layer = tf.nn.relu(conv_layer)
# Apply Max Pooling
conv_layer = tf.nn.max_pool(
    conv_layer,
    ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1],
    padding='SAME')
```

The `tf.nn.max_pool()` function performs max pooling with the `ksize` parameter as the size of the filter and the `strides` parameter as the length of the stride. 2x2 filters with a stride of 2x2 are common in practice.
The `ksize` and `strides` parameters are structured as 4-element lists, with each element corresponding to a dimension of the input tensor (`[batch, height, width, channels]`). For both `ksize` and `strides`, the batch and channel dimensions are typically set to `1`.

Convolutional Network in TensorFlow

It's time to walk through an example Convolutional Neural Network (CNN) in TensorFlow.

The structure of this network follows the classic structure of CNNs, which is a mix of convolutional layers and max pooling, followed by fully-connected layers.

The code you'll be looking at is similar to what you saw in the segment on **Deep Neural Network in TensorFlow**, except we restructured the architecture of this network as a CNN. Just like in that segment, here you'll study the line-by-line breakdown of the code. If you want, you can even **download the code** and run it yourself.
Thanks to **Aymeric Damien** for providing the original TensorFlow model on which this segment is based.
Time to dive in!

Dataset

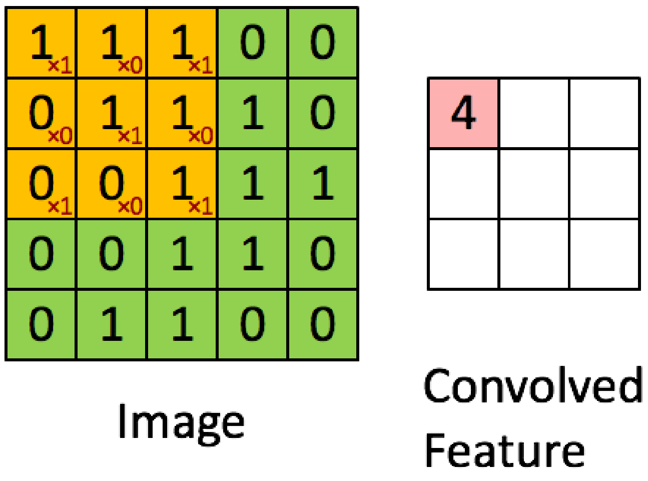
You've seen this section of code from previous lessons. Here we're importing the MNIST dataset and using a convenient TensorFlow function to batch, scale, and One-Hot encode the data.

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets(".", one_hot=True, reshape=False)

import tensorflow as tf

# Parameters
learning_rate = 0.00001
epochs = 10
batch_size = 128

# Number of samples to calculate validation and accuracy
# Decrease this if you're running out of memory to calculate accuracy
test_valid_size = 256

# Network Parameters
n_classes = 10  # MNIST total classes (0-9 digits)
dropout = 0.75  # Dropout, probability to keep units
```

Weights and Biases

```
# Store layers weight & bias
weights = {
    'wc1': tf.Variable(tf.random_normal([5, 5, 1, 32])),
    'wc2': tf.Variable(tf.random_normal([5, 5, 32, 64])),
    'wd1': tf.Variable(tf.random_normal([7*7*64, 1024])),
    'out': tf.Variable(tf.random_normal([1024, n_classes]))}

biases = {
    'bc1': tf.Variable(tf.random_normal([32])),
    'bc2': tf.Variable(tf.random_normal([64])),
    'bd1': tf.Variable(tf.random_normal([1024])),
    'out': tf.Variable(tf.random_normal([n_classes]))}
```

Convolutions



Convolution with 3×3 Filter.
Source: http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

The above is an example of a convolution with a 3x3 filter and a stride of 1 being applied to data with a range of 0 to 1. The convolution for each 3x3 section is calculated against the weight, `[[1, 0, 1], [0, 1, 0], [1, 0, 1]]`, then a bias is added to create the convolved feature on the right. In this case, the bias is zero. In TensorFlow, this is all done using `tf.nn.conv2d()` and `tf.nn.bias_add()`.

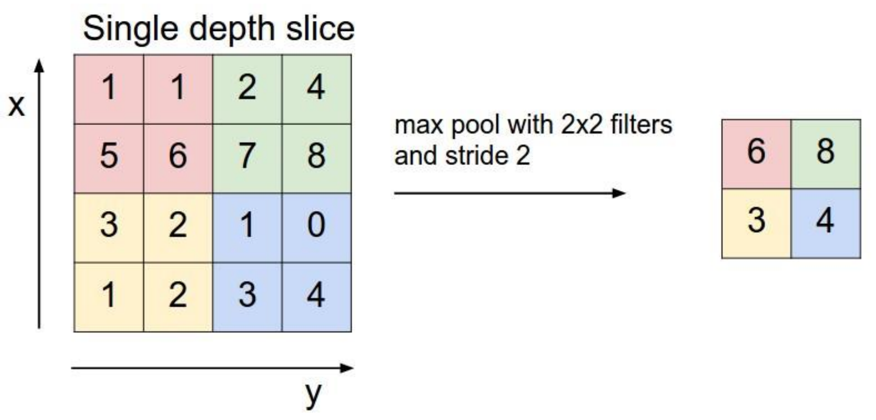
```
def conv2d(x, W, b, strides=1):
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], padding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)
```

The `tf.nn.conv2d()` function computes the convolution against weight `W` as shown above. In TensorFlow, stride is an array of 4 elements; the first element in the stride array indicates the stride for batch and last element indicates stride for features. It's good practice to remove the batches or features you want to skip from the dataset than to use stride. You can always set the first and last element to 1 in order to use all batches and features.

The middle two elements are the strides for height and width respectively. I've mentioned stride as one number because you usually have a square stride where `height = width`. When someone says they are using a stride of 3, they usually mean `tf.nn.conv2d(x, W, strides=[1, 3, 3, 1])`.

To make life easier, the code is using `tf.nn.bias_add()` to add the bias.
Using `tf.add()` doesn't work when the tensors aren't the same shape.

Max Pooling



Max Pooling with 2x2 filter and stride of 2. Source: http://cs231n.github.io/convolutional-networks/

The above is an example of max pooling with a 2x2 filter and stride of 2. The left square is the input and the right square is the output. The four 2x2 colors in input represents each time the filter was applied to create the max on the right side. For example, `[[1, 1], [5, 6]]` becomes 6 and `[[3, 2], [1, 2]]` becomes 3.

```
def maxpool2d(x, k=2):
    return tf.nn.max_pool(
        x,
        ksize=[1, k, k, 1],
        strides=[1, k, k, 1],
        padding='SAME')
```

The `tf.nn.max_pool()` function does exactly what you would expect, it performs max pooling with the `ksize` parameter as the size of the filter.

Model

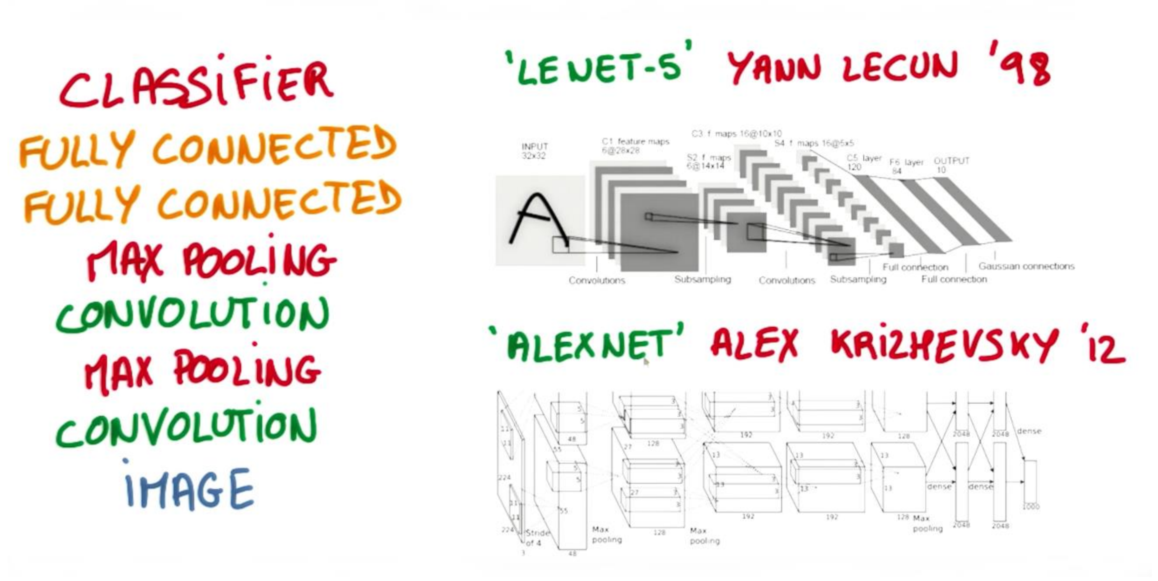


Image from Explore The Design Space video

In the code below, we're creating 3 layers alternating between convolutions and max pooling followed by a fully connected and output layer. The transformation of each layer to new dimensions are shown in the comments. For example, the first layer shapes the images from 28x28x1 to 28x28x32 in the convolution step. Then next step applies max pooling, turning each sample into 14x14x32. All the layers are applied from `conv1`to `output`, producing 10 class predictions.

```
def conv_net(x, weights, biases, dropout):
    # Layer 1 - 28*28*1 to 14*14*32
    conv1 = conv2d(x, weights['wc1'], biases['bc1'])
    conv1 = maxpool2d(conv1, k=2)

    # Layer 2 - 14*14*32 to 7*7*64
    conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])
    conv2 = maxpool2d(conv2, k=2)

    # Fully connected layer - 7*7*64 to 1024
    fc1 = tf.reshape(conv2, [-1, weights['wd1'].get_shape().as_list()[0]])
    fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])
    fc1 = tf.nn.relu(fc1)
    fc1 = tf.nn.dropout(fc1, dropout)

    # Output Layer - class prediction - 1024 to 10
    out = tf.add(tf.matmul(fc1, weights['out']), biases['out'])
    return out
```

Session

Now let's run it!

```
# tf Graph input
x = tf.placeholder(tf.float32, [None, 28, 28, 1])
y = tf.placeholder(tf.float32, [None, n_classes])
keep_prob = tf.placeholder(tf.float32)

# Model
logits = conv_net(x, weights, biases, keep_prob)

# Define loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits, y))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
\
    .minimize(cost)

# Accuracy
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initializing the variables
init = tf. global_variables_initializer()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    for epoch in range(epochs):
        for batch in range(mnist.train.num_examples//batch_size):
            batch_x, batch_y = mnist.train.next_batch(batch_size)
            sess.run(optimizer, feed_dict={
                x: batch_x,
                y: batch_y,
                keep_prob: dropout})
            # Calculate batch loss and accuracy
            loss = sess.run(cost, feed_dict={
                x: batch_x,
                y: batch_y,
                keep_prob: 1.})
            valid_acc = sess.run(accuracy, feed_dict={
                x: mnist.validation.images[:test_valid_size],
                y: mnist.validation.labels[:test_valid_size],
                keep_prob: 1.})

            print('Epoch {:>2}, Batch {:>3} -'
                  'Loss: {:>10.4f} Validation Accuracy: {:.6f}'.format(
                epoch + 1,
                batch + 1,
                loss,
                valid_acc))

    # Calculate Test Accuracy
    test_acc = sess.run(accuracy, feed_dict={
        x: mnist.test.images[:test_valid_size],
        y: mnist.test.labels[:test_valid_size],
        keep_prob: 1.})
    print('Testing Accuracy: {}'.format(test_acc))
```

That's it! You are on your way to mastering TensorFlow! Now that you've seen how to implement a CNN in TensorFlow, try and apply it on your own!