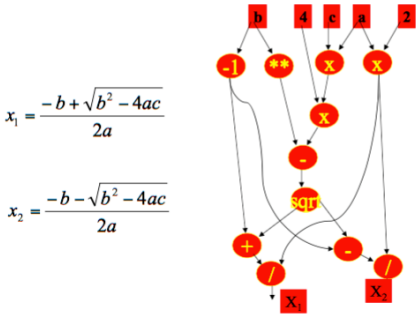
Tensor Flow Introduction

https://www.datacamp.com/community/tutorials/tensorflow-tutorial
https://www.tensorflow.org/tutorials/recurrent

You can use the TensorFlow library do to **numerical computations**, which in itself doesn't seem all too special, but these computations **are done with data flow graphs** (*A directed graph that shows the data dependencies between a number of functions[1]).* In these graphs, **nodes represent mathematical operations**, while the **edges represent** the data, which usually are **multidimensional data arrays** or tensors, that are communicated between these edges.



1: Data flow graphs for Bhaskara formula

Mathematics Review

Vectors

Array of numbers having just one column and a certain number of rows. You could also consider vectors **as scalar magnitudes** that **have been given a direction.**

Plane Vectors

They are much like regular vectors as you have seen above, with the sole difference **that they find themselves in a vector space.**

If you have a vector that is 2 X 1 you can represent vectors on the coordinate (x,y) plane with arrows or rays.

Tensor

A tensor, then, is the mathematical representation of a physical entity that may be characterized by **magnitude and multiple directions**.

And, just like you represent a scalar with a single number and a vector with sequence of three numbers in a 3-dimensional space, for example, a tensor can be represented by an array of 3R numbers in a 3-dimensional space.

In an N-dimensional space, **scalars** will still require only **one number**, while **vectors** will require **N numbers**, and **tensors** will require **N^R numbers**. This explains why you often hear that scalars are tensors of rank 0: since they have no direction, you can represent them with one number.

Installing TensorFlow

Virtualenv is a virtual Python environment isolated from other Python development, incapable of interfering with or being affected by other Python programs on the same machine.

Install and Virtualenv:

```
pip3 install --upgrade --no-binary :all:  virtualenv
```

Create a Virtualenv environment:

```
virtualenv --system-site-packages -p python3 ~/tensorflow
cd tensorflow
source ./bin/activate
```

Ensure pip ≥8.1 is installed:

```
(tensorflow)$ easy_install -U pip
```

Issue one of the following commands to install TensorFlow and **all the packages** that TensorFlow requires into the active Virtualenv environment:

```
(tensorflow)$ pip3 install --upgrade --no-binary :all:  tensorflow (macOS)
(tensorflow)$ pip3 install --upgrade tensorflow   (Ubuntu)
```

```
" --no-binary :all:  "  option avoid a error on macOS
```

When you are done using TensorFlow, you may deactivate the environment by issuing the following command:

```
(tensorflow)$ deactivate
```

Validate your installation

Activate your Virtualenv container.

```
source ./bin/activate
```

Invoke python from your shell as follows:

```
python3
```

Enter the following short program inside the python interactive shell:

```
# Python
import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print(sess.run(hello))
```

Getting Started With TensorFlow: Basics

First, import the tensorflow library under the alias tf, as you have seen in the previous section. Then initialize two variables that are actually constants. Pass an array of four numbers to the constant() function.

Next, you can use multiply() to multiply your two variables. Store the result in the result variable.

Lastly, print out the result with the help of the print() function.

```
# Import `tensorflow`
import tensorflow as tf
# Initialize two constants
x1 = tf.constant([1,2,3,4])
x2 = tf.constant([5,6,7,8])
# Multiply
result = tf.multiply(x1, x2)
# Print the result
print(result)
```

```
>>> Tensor("Mul:0", shape=(4,), dtype=int32)
```

However, contrary to what you might expect, the result doesn't actually get calculated; It just defined the model but no process ran to calculate the result. You can see this in the print-out: there's not really a result that you want to see (namely, 30). This means that **TensorFlow has a lazy evaluation**!

```
# Import `tensorflow`
import tensorflow as tf
# Initialize two constants
```

```
x1 = tf.constant([1,2,3,4])
x2 = tf.constant([5,6,7,8])
# Multiply
result = tf.multiply(x1, x2)
# Intialize the Session
sess = tf.Session()

# Print the result
print(sess.run(result))
# Close the session
sess.close()

>>>> [ 5 12 21 32]
```

In the code chunks above you have just defined a default Session, but it's also good to know that you can pass in options as well. You can, for example, specify the config argument and then use the ConfigProto protocol buffer to add **configuration options for your session**.

Getting Started With TensorFlow: traffic_signs

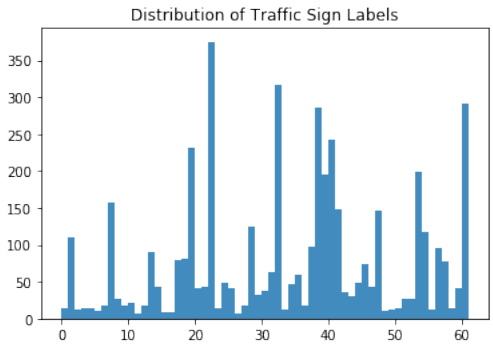
Loading data:

```
pip3 install scikit-image
python3 -m pip install numpy
python3 -m pip install matplotlib
```

Observation: On macOS Create a file ~/.matplotlib/matplotlibrc there and add the following code: backend: TkAgg

Traffic Sign Statistics:

With your data loaded in, it's time for some data inspection!



You clearly see that **not all types of traffic signs are equally represented** in the dataset. This is something that you'll deal with later when you're **manipulating the data before you start modeling your neural network**.

Visualizing the Traffic Sign:

The previous, small analyses or checks have already given you some idea of the data that you're working with, but when your data mostly consists of images, the step that you should take to explore your data is by visualizing it.

These four images are not of the same size!

Feature Extraction

To tackle the differing image sizes, you're going to rescale the images and you'll also go through the trouble of converting the images to grayscale:

```
# Resize images
images32 = [transform.resize(image, (28, 28)) for image in images]
images32 = np.array(images32)

# Image Conversion to Grayscale
images32 = rgb2gray(np.array(images32))
```

Modeling The Neural Network

```
# Initialize placeholders
# Remember that placeholders are values that are unassigned and that will be
initialized by the session when you run it.
x = tf.placeholder(dtype = tf.float32, shape = [None, 28, 28])
```

```
y = tf.placeholder(dtype = tf.int32, shape = [None])

# Flatten the input data
#Then, you build up the network.
#You first start off by flattening (convert a matrix to an arrray) the input with the
help of the flatten() function,
#which will give you an array of shape [None, 784] instead of the [None, 28, 28],
which is the shape of your grayscale images.
images_flat = tf.contrib.layers.flatten(x)

# Fully connected layer
#After you have flattened the input, you construct a fully connected layer that
generates logits of size [None, 62]
#Logits is the function operates on the unscaled output of earlier layers and that
uses the relative scale to understand the units is linear.
logits = tf.contrib.layers.fully_connected(images_flat, 62, tf.nn.relu)


# Define a loss function
# With the multi-layer perceptron built out, you can define the loss function.
loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labels = y,
logits = logits))

# Define an optimizer
# You also want to define a training optimizer;
# Some of the most popular optimization algorithms used are the Stochastic Gradient
Descent (SGD), ADAM and RMSprop.
# Depending on whichever algorithm you choose, you'll need to tune certain parameters,
such as learning rate or momentum.
# In this case, you pick the ADAM optimizer, for which you define the learning rate at
0.001.
train_op = tf.train.AdamOptimizer(learning_rate=0.001).minimize(loss)

# Convert logits to label indexes
correct_pred = tf.argmax(logits, 1)

# Define an accuracy metric
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

Running The Neural Network

Now that you have built up your model layer by layer, it's time to actually run it! To do this, you first need to initialize a session with the help of Session() to which you can pass your graph graph that you defined in the previous section. Next, you can run the session with run(), to which you pass the initialized operations in the form of the init variable that you also defined in the previous section.

Next, you can use this initialized session to start epochs or training loops. In this case, you pick 201 because you want to be able to register the last loss_value; In the loop, you run the session

with the training optimizer and the loss (or accuracy) metric that you defined in the previous section.

```
tf.set_random_seed(1234)
sess = tf.Session()

sess.run( tf.global_variables_initializer() )

for i in range(201):
        print('EPOCH', i)
        _, accuracy_val = sess.run([train_op, accuracy], feed_dict={x: images32, y:
labels})
        if i % 10 == 0:
            print("Loss: ", loss)
        print('DONE WITH EPOCH')

# Pick 10 random images
sample_indexes = random.sample(range(len(images32)), 10)
sample_images = [images32[i] for i in sample_indexes]
sample_labels = [labels[i] for i in sample_indexes]

# Run the "predicted_labels" op.
predicted = sess.run([correct_pred], feed_dict={x: sample_images})[0]

# Print the real and predicted labels
print(sample_labels)
print(predicted)

# Display the predictions and the ground truth visually.
fig = plt.figure(figsize=(10, 10))
for i in range(len(sample_images)):
    truth = sample_labels[i]
    prediction = predicted[i]
    plt.subplot(5, 2,1+i)
    plt.axis('off')
    color='green' if truth == prediction else 'red'
    plt.text(40, 10, "Truth:        {0}\nPrediction: {1}".format(truth, prediction),
             fontsize=12, color=color)
    plt.imshow(sample_images[i])

plt.show()
```