

TensorFlow Basics - Tutorial

Learn how to build a neural network and how to train, evaluate and optimize it with TensorFlow

Deep learning is a subfield of machine learning that is a set of algorithms that is inspired by the structure and function of the brain.

TensorFlow is the second machine learning framework that Google created and used to design, build, and train deep learning models. You can use the TensorFlow library to do numerical computations, which in itself doesn't seem all too special, but these computations are done with data flow graphs. 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.

You see? The name "TensorFlow" is derived from the operations which neural networks perform on multidimensional data arrays or tensors! It's literally a flow of tensors. For now, this is all you need to know about tensors, but you'll go deeper into this in the next sections!

Today's TensorFlow tutorial for beginners will introduce you to performing deep learning in an interactive way:

- You'll first learn more about [tensors](#);
- Then, the tutorial you'll briefly go over some of the ways that you can [install TensorFlow](#) on your system so that you're able to get started and load data in your workspace;
- After this, you'll go over some of the [TensorFlow basics](#): you'll see how you can easily get started with simple computations.
- After this, you get started on the real work: you'll load in data on Belgian traffic signs and [exploring](#) it with simple statistics and plotting.
- In your exploration, you'll see that there is a need to [manipulate your data](#) in such a way that you can feed it to your model. That's why you'll take the time to rescale your images and convert them to grayscale.
- Next, you can finally get started on [your neural network model](#)! You'll build up your model layer per layer;
- Once the architecture is set up, you can use it to [train your model interactively](#) and to eventually also [evaluate](#) it by feeding some test data to it.
- Lastly, you'll get some pointers for [further](#) improvements that you can do to the model you just constructed and how you can continue your learning with TensorFlow.

Also, you could be interested in a course on [Deep Learning in Python](#), DataCamp's [Keras tutorial](#) or the [keras with R tutorial](#).

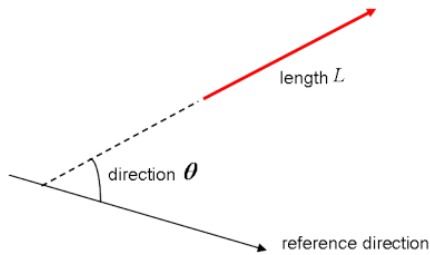
Introducing Tensors

To understand tensors well, it's good to have some working knowledge of linear algebra and vector calculus. You already read in the introduction that tensors are implemented in TensorFlow as multidimensional data arrays, but some more introduction is maybe needed in order to completely grasp tensors and their use in machine learning.

Plane Vectors

Before you go into plane vectors, it's a good idea to shortly revise the concept of "vectors"; Vectors are special types of matrices, which are rectangular arrays of numbers. Because vectors are ordered collections of numbers, they are often seen as column matrices: they have just one column and a certain number of rows. In other terms, you could also consider vectors as scalar magnitudes that have been given a direction.

Remember: an example of a scalar is "5 meters" or "60 m/sec", while a vector is, for example, "5 meters north" or "60 m/sec East". The difference between these two is obviously that the vector has a direction. Nevertheless, these examples that you have seen up until now might seem far off from the vectors that you might encounter when you're working with machine learning problems. This is normal; The length of a mathematical vector is a pure number: it is absolute. The direction, on the other hand, is relative: it is measured relative to some reference direction and has units of radians or degrees. You usually assume that the direction is positive and in counterclockwise rotation from the reference direction.



Visually, of course, you represent vectors as arrows, as you can see in the picture above. This means that you can consider vectors also as arrows that have direction and length. The direction is indicated by the arrow's head, while the length is indicated by the length of the arrow.

So what about plane vectors then?

Plane vectors are the most straightforward setup of tensors. They are much like regular vectors as you have seen above, with the sole difference that they find themselves in a vector space. To understand this better, let's start with an example: you have a vector that is 2×1 . This means that the vector belongs to the set of real numbers that come paired two at a time. Or, stated differently, they are part of two-space. In such cases, you can represent vectors on the coordinate (x,y) plane with arrows or rays.

Working from this coordinate plane in a standard position where vectors have their endpoint at the origin $(0,0)$, you can derive the x coordinate by looking at the first row of the vector, while you'll find the y coordinate in the second row. Of course, this standard position doesn't always need to be maintained: vectors can move parallel to themselves in the plane without experiencing changes.

Note that similarly, for vectors that are of size 3×1 , you talk about the three-space. You can represent the vector as a three-dimensional figure with arrows pointing to positions in the vectors pace: they are drawn on the standard x , y and z axes.

It's nice to have these vectors and to represent them on the coordinate plane, but in essence, you have these vectors so that you can perform operations on them and one thing that can help you in doing this is by expressing your vectors as bases or unit vectors.

Unit vectors are vectors with a magnitude of one. You'll often recognize the unit vector by a lowercase letter with a circumflex, or "hat". Unit vectors will come in convenient if you want to express a 2-D or 3-D vector as a sum of two or three orthogonal components, such as the x - and y -axes, or the z -axis.

And when you are talking about expressing one vector, for example, as sums of components, you'll see that you're talking about component vectors, which are two or more vectors whose sum is that given vector.

Tip: watch [this video](#), which explains what tensors are with the help of simple household objects!

Tensors

Next to plane vectors, also covectors and linear operators are two other cases that all three together have one thing in common: they are specific cases of tensors. You still remember how a vector was characterized in the previous section as scalar magnitudes that have been given a direction. 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 a sequence of three numbers in a 3-dimensional space, for example, a tensor can be represented by an array of 3^R numbers in a 3-dimensional space. The " R " in this notation represents the rank of the tensor: this means that in a 3-dimensional space, a second-rank tensor can be represented by 3 to the power of 2 or 9 numbers. 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. With this in mind, it's relatively easy to recognize scalars, vectors, and tensors and to set them apart: scalars can be represented by a single number, vectors by an ordered set of numbers, and tensors by an array of numbers.

What makes tensors so unique is the combination of components and basis vectors: basis vectors transform one way between reference frames and the components transform in just such a way as to keep the combination between components and basis vectors the same.

Installing TensorFlow

Now that you know more about TensorFlow, it's time to get started and install the library. Here, it's good to know that TensorFlow provides APIs for Python, C++, Haskell, Java, Go, Rust, and there's also a third-party package for R called `tensorflow`.

Tip: if you want to know more about deep learning packages in R, consider checking out DataCamp's [keras: Deep Learning in R Tutorial](#).

In this tutorial, you will download a version of TensorFlow that will enable you to write the code for your deep learning project in Python. On the [TensorFlow installation webpage](#), you'll see some of the most common ways and latest instructions to install TensorFlow using `virtualenv`, `pip`, Docker and lastly, there are also some of the other ways of installing TensorFlow on your personal computer.

Note You can also install TensorFlow with Conda if you're working on Windows. However, since the installation of TensorFlow is community supported, it's best to check the [official installation instructions](#).

Now that you have gone through the installation process, it's time to double check that you have installed TensorFlow correctly by importing it into your workspace under the alias `tf`:

```
import tensorflow as tf
```

Note that the alias that you used in the line of code above is sort of a convention - It's used to ensure that you remain consistent with other developers that are using TensorFlow in data science projects on the one hand, and with open-source TensorFlow projects on the other hand.

Getting Started With TensorFlow: Basics

You'll generally write TensorFlow programs, which you run as a chunk; This is at first sight kind of contradictory when you're working with Python. However, if you would like, you can also use TensorFlow's Interactive Session, which you can use to work more interactively with the library. This is especially handy when you're used to working with IPython.

For this tutorial, you'll focus on the second option: this will help you to get kickstarted with deep learning in TensorFlow. But before you go any further into this, let's first try out some minor stuff before you start with the heavy lifting.

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.

Note that you could potentially also pass in an integer, but that more often than not, you'll find yourself working with arrays. As you saw in the introduction, tensors are all about arrays! So make sure that you pass in an array :) 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)
```

Note that you have defined constants in the DataCamp Light code chunk above. However, there are two other types of values that you can potentially use, namely [placeholders](#), which are values that are unassigned and that will be initialized by the session when you run it. Like the name already gave away, it's just a placeholder for a tensor that will always be fed when the session is run; There are also [Variables](#), which are values that can change. The constants, as you might have already gathered, are values that don't change.

The result of the lines of code is an abstract tensor in the computation graph. 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!

However, if you do want to see the result, you have to run this code in an interactive session. You can do this in a few ways, as is demonstrated in the DataCamp Light code chunks below:

```
# 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)
# Initialize the Session
sess = tf.Session()
# Print the result
print(sess.run(result))
# Close the session
sess.close()
```

Note that you can also use the following lines of code to start up an interactive Session, run the `result` and close the Session automatically again after printing the `output`:

```
# 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)
# Initialize Session and run `result`
with tf.Session() as sess:
    output = sess.run(result)
    print(output)
```

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.

For example, if you add

```
config=tf.ConfigProto(log_device_placement=True)
```

to your Session, you make sure that you log the GPU or CPU device that is assigned to an operation. You will then get information which devices are used in the session for each operation. You could use the following configuration session also, for example, when you use soft constraints for the device placement:

```
config=tf.ConfigProto(allow_soft_placement=True)
```

Now that you've got TensorFlow installed and imported into your workspace and you've gone through the basics of working with this package, it's time to leave this aside for a moment and turn your attention to your data. Just like always, you'll first take your time to explore and understand your data better before you start modeling your neural network.

Belgian Traffic Signs: Background

Even though traffic is a topic that is generally known amongst you all, it doesn't hurt going briefly over the observations that are included in this dataset to see if you understand everything before you start. In essence, in this section, you'll get up to speed with the domain knowledge that you need to have to go further with this tutorial.

Of course, because I'm Belgian, I'll make sure you'll also get some anecdotes :)

- *Belgian traffic signs are usually in Dutch and French. This is good to know, but for the dataset that you'll be working with, it's not too important!*
- *There are six categories of traffic signs in Belgium: warning signs, priority signs, prohibitory signs, mandatory signs, signs related to parking and standing still on the road and, lastly, designatory signs.*
- *On January 1st, 2017, more than 30,000 traffic signs were removed from Belgian roads. These were all prohibitory signs relating to speed.*
- *Talking about removal, the overwhelming presence of traffic signs has been an ongoing discussion in Belgium (and by extension, the entire European Union).*

Loading And Exploring The Data

Now that you have gathered some more background information, it's time to download the dataset [here](#). You should get the two zip files listed next to "BelgiumTS for Classification (cropped images)", which are called "BelgiumTSC_Training" and "BelgiumTSC_Testing".

Tip: if you have downloaded the files or will do so after completing this tutorial, take a look at the folder structure of the data that you've downloaded! You'll see that the testing, as well as the training data folders, contain 61 subfolders, which are the 62 types of traffic signs that you'll use for classification in this tutorial. Additionally, you'll find that the files have the file extension `.ppm` or Portable Pixmap Format. You have downloaded images of the traffic signs!

- *Let's get started with importing the data into your workspace. Let's start with the lines of code that appear below the User-Defined Function (UDF) `load_data()`:*
- *First, set your `ROOT_PATH`. This path is the one where you have made the directory with your training and test data.*
- *Next, you can add the specific paths to your `ROOT_PATH` with the help of the `join()` function. You store these two specific paths in `train_data_directory` and `test_data_directory`.*
- *You see that after, you can call the `load_data()` function and pass in the `train_data_directory` to it.*
- *Now, the `load_data()` function itself starts off by gathering all the subdirectories that are present in the `train_data_directory`; It does so with the help of list comprehension, which is quite a natural way of constructing lists - it basically says that, if you find something in the `train_data_directory`, you'll double check whether this is a directory, and if it is one, you'll add it to your list. **Remember** that each subdirectory represents a label.*

- Next, you have to loop through the subdirectories. You first initialize two lists, `labels` and `images`. Next, you gather the paths of the subdirectories and the file names of the images that are stored in these subdirectories. After, you can collect the data in the two lists with the help of the `append()` function.

```
def load_data(data_directory):
    directories = [d for d in os.listdir(data_directory)
                   if os.path.isdir(os.path.join(data_directory, d))]
    labels = []
    images = []
    for d in directories:
        label_directory = os.path.join(data_directory, d)
        file_names = [os.path.join(label_directory, f)
                       for f in os.listdir(label_directory)
                       if f.endswith(".ppm")]
        for f in file_names:
            images.append(skimage.data.imread(f))
            labels.append(int(d))
    return images, labels
```

```
ROOT_PATH = "/your/root/path"
train_data_directory = os.path.join(ROOT_PATH, "TrafficSigns/Training")
test_data_directory = os.path.join(ROOT_PATH, "TrafficSigns/Testing")
```

```
images, labels = load_data(train_data_directory)
```

Note that in the above code chunk, the training and test data are located in folders named "Training" and "Testing", which are both subdirectories of another directory "TrafficSigns". On a local machine, this could look something like "/Users/Name/Downloads/TrafficSigns", with then two subfolders called "Training" and "Testing".

Tip: review how to write functions in Python with DataCamp's [Python Functions Tutorial](#).

Traffic Sign Statistics

With your data loaded in, it's time for some data inspection! You can start with a pretty simple analysis with the help of the `ndim` and `size` attributes of the `images` array:

Note that the `images` and `labels` variables are lists, so you might need to use `np.array()` to convert the variables to an array in your own workspace. This has been done for you here!

```
# Print the `images` dimensions
print(images.ndim)
# Print the number of `images`'s elements
print(images.size)
# Print the first instance of `images`
images[0]
```

Note that the `images[0]` that you printed out is, in fact, one single image that is represented by arrays in arrays! This might seem counterintuitive at first, but it's something that you'll get used to as you go further into working with images in machine learning or deep learning applications.

Next, you can also take a small look at the `labels`, but you shouldn't see too many surprises at this point:

```
# Print the `labels` dimensions
print(labels.ndim)
# Print the number of `labels`'s elements
print(labels.size)
# Count the number of labels
print(len(set(labels)))
```

These numbers already give you some insights into how successful your import was and the exact size of your data.

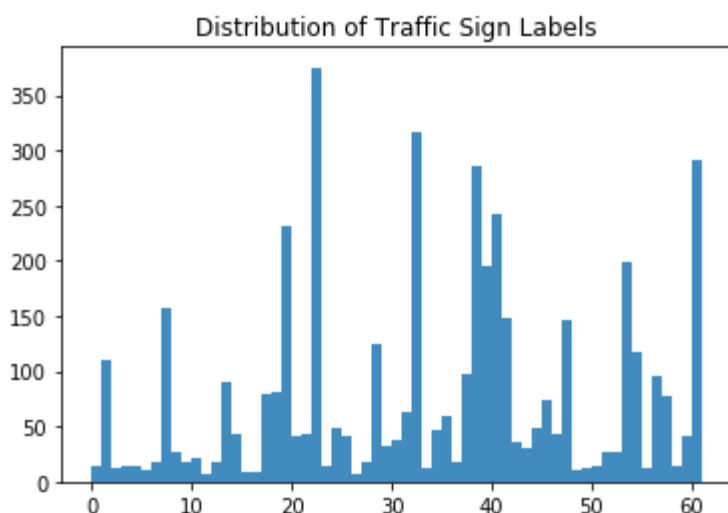
At first sight, everything has been executed the way you expected it to, and you see that the size of the array is considerable if you take into account that you're dealing with arrays within arrays.

Tip try adding the following attributes to your arrays to get more information about the memory layout, the length of one array element in bytes and the total consumed bytes by the array's elements with the `flags`, `itemsize`, and `nbytes` attributes. You can test this out in the IPython console in the DataCamp Light chunk above!

Next, you can also take a look at the distribution of the traffic signs:

```
# Import the `pyplot` module
import matplotlib.pyplot as plt
# Make a histogram with 62 bins of the `labels` data
plt.hist(labels, 62)
# Show the plot
plt.show()
```

Awesome job! Now let's take a closer look at the histogram that you made!



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.

At first sight, you see that there are labels that are more heavily present in the dataset than others: the labels 22, 32, 38, and 61 definitely jump out. At this point, it's nice to keep this in mind, but you'll definitely go further into this in the next section!

Visualizing The Traffic Signs

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.

Let's check out some random traffic signs:

- First, make sure that you import the `pyplot` module of the `matplotlib` package under the common alias `plt`.
- Then, you're going to make a list with 4 random numbers. These will be used to select traffic signs from the `images` array that you have just inspected in the previous section. In this case, you go for 300, 2250, 3650 and 4000.

- Next, you'll say that for every element in the length of that list, so from 0 to 4, you're going to create subplots without axes (so that they don't go running with all the attention and your focus is solely on the images!). In these subplots, you're going to show a specific image from the `images` array that is in accordance with the number at the index `i`. In the first loop, you'll pass 300 to `images[i]`, in the second round 2250, and so on. Lastly, you'll adjust the subplots so that there's enough width in between them.
- The last thing that remains is to show your plot with the help of the `show()` function!

There you go:

```
# Import the `pyplot` module of `matplotlib`
import matplotlib.pyplot as plt

# Determine the (random) indexes of the images that you want to see
traffic_signs = [300, 2250, 3650, 4000]

# Fill out the subplots with the random images that you defined
for i in range(len(traffic_signs)):
    plt.subplot(1, 4, i+1)
    plt.axis('off')
    plt.imshow(images[traffic_signs[i]])
    plt.subplots_adjust(wspace=0.5)

plt.show()
```

As you guessed by the 62 labels that are included in this dataset, the signs are different from each other. But what else do you notice? Take another close look at the images below:



These four images are not of the same size!

You can obviously toy around with the numbers that are contained in the `traffic_signs` list and follow up more thoroughly on this observation, but be as it may, this is an important observation which you will need to take into account when you start working more towards manipulating your data so that you can feed it to the neural network.

Let's confirm the hypothesis of the differing sizes by printing the shape, the minimum and maximum values of the specific images that you have included into the subplots.

The code below heavily resembles the one that you used to create the above plot, but differs in the fact that here, you'll alternate sizes and images instead of plotting just the images next to each other:

```
# Import `matplotlib`
import matplotlib.pyplot as plt

# Determine the (random) indexes of the images
traffic_signs = [300, 2250, 3650, 4000]
```



```
# Fill out the subplots with the random images and add shape, min and max values
for i in range(len(traffic_signs)):
    plt.subplot(1, 4, i+1)
    plt.axis('off')
    plt.imshow(images[traffic_signs[i]])
    plt.subplots_adjust(wspace=0.5)
    plt.show()
    print("shape: {0}, min: {1}, max: {2}".format(images[traffic_signs[i]].shape,
                                                images[traffic_signs[i]].min(),
                                                images[traffic_signs[i]].max()))
```

Note how you use the `format()` method on the string `"shape: {0}, min: {1}, max: {2}"` to fill out the arguments `{0}`, `{1}`, and `{2}` that you defined.



shape: (62, 61, 3), min: 3, max: 160



shape: (110, 96, 3), min: 3, max: 255



shape: (379, 153, 3), min: 0, max: 255



shape: (100, 68, 3), min: 17, max: 255

Now that you have seen loose images, you might also want to revisit the histogram that you printed out in the first steps of your data exploration; You can easily do this by plotting an overview of all the 62 classes and one image that belongs to each class:

```
# Import the `pyplot` module as `plt`
import matplotlib.pyplot as plt
```

```
# Get the unique labels
unique_labels = set(labels)
```

```
# Initialize the figure
plt.figure(figsize=(15, 15))
```

```
# Set a counter
i = 1
```

```
# For each unique label,
for label in unique_labels:
    # You pick the first image for each label
    image = images[labels.index(label)]
    # Define 64 subplots
    plt.subplot(8, 8, i)
    # Don't include axes
```

```
plt.axis('off')
# Add a title to each subplot
plt.title("Label {0} ({1})".format(label, labels.count(label)))
# Add 1 to the counter
i += 1
# And you plot this first image
plt.imshow(image)
```

```
# Show the plot
plt.show()
```

Note that even though you define 64 subplots, not all of them will show images (as there are only 62 labels!). Note also that again, you don't include any axes to make sure that the readers' attention doesn't dwell far from the main topic: the traffic signs!



As you mostly guessed in the histogram above, there are considerably more traffic signs with labels 22, 32, 38, and 61. This hypothesis is now confirmed in this plot: you see that there are 375 instances with label 22, 316 instances with label 32, 285 instances with label 38 and, lastly, 282 instances with label 61.

One of the most interesting questions that you could ask yourself now is whether there's a connection between all of these instances - maybe all of them are designatory signs?

Let's take a closer look: you see that label 22 and 32 are prohibitory signs, but that labels 38 and 61 are designatory and a priority signs, respectively. This means that there's not an immediate connection between these four, except for the fact that half of the signs that have a substantial presence in the dataset is of the prohibitory kind.

Feature Extraction

Now that you have thoroughly explored your data, it's time to get your hands dirty! Let's recap briefly what you discovered to make sure that you don't forget any steps in the manipulation:

- *The size of the images was unequal;*
- *There are 62 labels or target values (as your labels start at 0 and end at 61);*
- *The distribution of the traffic sign values is pretty unequal; There wasn't really any connection between the signs that were heavily present in the dataset.*

Now that you have a clear idea of what you need to improve, you can start with manipulating your data in such a way that it's ready to be fed to the neural network or whichever model you want to feed it too. Let's start first with extracting some features - you'll rescale the images, and you'll convert the images that are held in the `images` array to grayscale. You'll do this color conversion mainly because the color matters less in classification questions like the one you're trying to answer now. For detection, however, the color does play a big part! So in those cases, it's not needed to do that conversion!

Rescaling Images

To tackle the differing image sizes, you're going to rescale the images; You can easily do this with the help of the `skimage` or Scikit-Image library, which is a collection of algorithms for image processing.

In this case, the `transform` module will come in handy, as it offers you a `resize()` function; You'll see that you make use of list comprehension (again!) to resize each image to 28 by 28 pixels. Once again, you see that the way you actually form the list: for every image that you find in the `images` array, you'll perform the transformation operation that you borrow from the `skimage` library. Finally, you store the result in the `images28` variable:

```
# Import the `transform` module from `skimage`  
from skimage import transform
```

```
# Rescale the images in the `images` array  
images28 = [transform.resize(image, (28, 28)) for image in images]
```

This was fairly easy wasn't it?

Note that the images are now four-dimensional: if you convert `images28` to an array and if you concatenate the attribute `shape` to it, you'll see that the printout tells you that `images28`'s dimensions are `(4575, 28, 28, 3)`. The images are 784-dimensional (because your images are 28 by 28 pixels).

You can check the result of the rescaling operation by re-using the code that you used above to plot the 4 random images with the help of the `traffic_signs` variable. Just don't forget to change all references to `images` to `images28`.

Check out the result here:



shape: (28, 28, 3), min: 0.061764705882353076, max: 0.6161764705882353



shape: (28, 28, 3), min: 0.07634053621448501, max: 1.0



shape: (28, 28, 3), min: 0.08464760904361845, max: 1.0



shape: (28, 28, 3), min: 0.08907563025210051, max: 1.0

Note that because you rescaled, your `min` and `max` values have also changed; They seem to be all in the same ranges now, which is really great because then you don't necessarily need to normalize your data!

Image Conversion to Grayscale

As said in the introduction to this section of the tutorial, the color in the pictures matters less when you're trying to answer a classification question. That's why you'll also go through the trouble of converting the images to grayscale.

Note, however, that you can also test out on your own what would happen to the final results of your model if you don't follow through with this specific step.

Just like with the rescaling, you can again count on the Scikit-Image library to help you out; In this case, it's the `color` module with its `rgb2gray()` function that you need to use to get where you need to be.

That's going to be nice and easy!

However, don't forget to convert the `images28` variable back to an array, as the `rgb2gray()` function does expect an array as an argument.

```
# Import `rgb2gray` from `skimage.color`
from skimage.color import rgb2gray
```

```
# Convert `images28` to an array
images28 = np.array(images28)
```

```
# Convert `images28` to grayscale
images28 = rgb2gray(images28)
```

Double check the result of your grayscale conversion by plotting some of the images; Here, you can again re-use and slightly adapt some of the code to show the adjusted images:

```
import matplotlib.pyplot as plt
```

```
traffic_signs = [300, 2250, 3650, 4000]
```

```
for i in range(len(traffic_signs)):
    plt.subplot(1, 4, i+1)
    plt.axis('off')
    plt.imshow(images28[traffic_signs[i]], cmap="gray")
    plt.subplots_adjust(wspace=0.5)
```

```
# Show the plot
plt.show()
```

Note that you indeed have to specify the color map or `cmap` and set it to `"gray"` to plot the images in grayscale. That is because `imshow()` by default uses, by default, a heatmap-like color map. Read more [here](#).



Tip: since you have been re-using this function quite a bit in this tutorial, you might look into how you can make it into a function :)

These two steps are very basic ones; Other operations that you could have tried out on your data include data augmentation (rotating, blurring, shifting, changing brightness,...). If you want, you could also set up an entire pipeline of data manipulation operations through which you send your images.

Deep Learning With TensorFlow

Now that you have explored and manipulated your data, it's time to construct your neural network architecture with the help of the TensorFlow package!

Modeling The Neural Network

Just like you might have done with Keras, it's time to build up your neural network, layer by layer.

If you haven't done so already, import `tensorflow` into your workspace under the conventional alias `tf`. Then, you can initialize the Graph with the help of `Graph()`. You use this function to define the computation. **Note** that with the Graph, you don't compute anything, because it doesn't hold any values. It just defines the operations that you want to be running later.

In this case, you set up a default context with the help of `as_default()`, which returns a context manager that makes this specific Graph the default graph. You use this method if you want to create multiple graphs in the same process: with this function, you have a global default graph to which all operations will be added if you don't explicitly create a new graph.

Next, you're ready to add operations to your graph. As you might remember from working with Keras, you build up your model, and then in compiling it, you define a loss function, an optimizer, and a metric. This now all happens in one step when you work with TensorFlow:

- *First, you define placeholders for inputs and labels because you won't put in the "real" data yet. **Remember** that placeholders are values that are unassigned and that will be initialized by the session when you run it. So when you finally run the session, these placeholders will get the values of your dataset that you pass in the `run()` function!*
- *Then, you build up the network. You first start by flattening 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.*
- *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 previous layers, and that uses the relative scale to understand the units is linear.*
- *With the multi-layer perceptron built out, you can define the loss function. The choice for a loss function depends on the task that you have at hand: in this case, you make use of*
- `sparse_softmax_cross_entropy_with_logits()`
- *This computes sparse softmax cross entropy between logits and labels. In other words, it measures the probability error in discrete classification tasks in which the classes are mutually exclusive. This means that each entry is in exactly one class. Here, a traffic sign can only have one single label. **Remember** that, while regression is used to predict continuous values, classification is used to predict discrete values or classes of data points. You wrap this function with `reduce_mean()`, which computes the mean of elements across dimensions of a tensor.*
- *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.*

- Lastly, you initialize the operations to execute before going over to the training.

```
# Import `tensorflow`
import tensorflow as tf

# Initialize placeholders
x = tf.placeholder(dtype = tf.float32, shape = [None, 28, 28])
y = tf.placeholder(dtype = tf.int32, shape = [None])

# Flatten the input data
images_flat = tf.contrib.layers.flatten(x)

# Fully connected layer
logits = tf.contrib.layers.fully_connected(images_flat, 62, tf.nn.relu)

# Define a loss function
loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labels = y,
                                                                    logits = logits))

# Define an optimizer
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))
```

You have now successfully created your first neural network with TensorFlow!

If you want, you can also print out the values of (most of) the variables to get a quick recap or checkup of what you have just coded up:

```
print("images_flat: ", images_flat)
print("logits: ", logits)
print("loss: ", loss)
print("predicted_labels: ", correct_pred)
```

Tip: if you see an error like “module 'pandas' has no attribute 'computation'”, consider upgrading the packages `dask` by running `pip install --upgrade dask` in your command line. See [this StackOverflow post](#) for more information.

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` 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. You also pass a `feed_dict` argument, with which you feed data to the model. After every 10 epochs, you'll get a log that gives you more insights into the loss or cost of the model.

As you have seen in the section on the TensorFlow basics, there is no need to close the session manually; this is done for you. However, if you want to try out a different setup, you probably will need to do so with `sess.close()` if you have defined your session as `sess`, like in the code chunk below:

```
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: images28, y: labels})
    if i % 10 == 0:
        print("Loss: ", loss)
    print('DONE WITH EPOCH')
```

Remember that you can also run the following piece of code, but that one will immediately close the session afterward, just like you saw in the introduction of this tutorial:

```
tf.set_random_seed(1234)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(201):
        _, loss_value = sess.run([train_op, loss], feed_dict={x: images28, y: labels})
        if i % 10 == 0:
            print("Loss: ", loss)
```

Note that you make use of `global_variables_initializer()` because the `initialize_all_variables()` function is deprecated. You have now successfully trained your model! That wasn't too hard, was it?

Evaluating Your Neural Network

You're not entirely there yet; You still need to evaluate your neural network. In this case, you can already try to get a glimpse of well your model performs by picking 10 random images and by comparing the predicted labels with the real labels.

You can first print them out, but why not use `matplotlib` to plot the traffic signs themselves and make a visual comparison?

```
# Import `matplotlib`
import matplotlib.pyplot as plt
import random

# Pick 10 random images
sample_indexes = random.sample(range(len(images28)), 10)
sample_images = [images28[i] for i in sample_indexes]
sample_labels = [labels[i] for i in sample_indexes]

# Run the "correct_pred" operation
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))
```

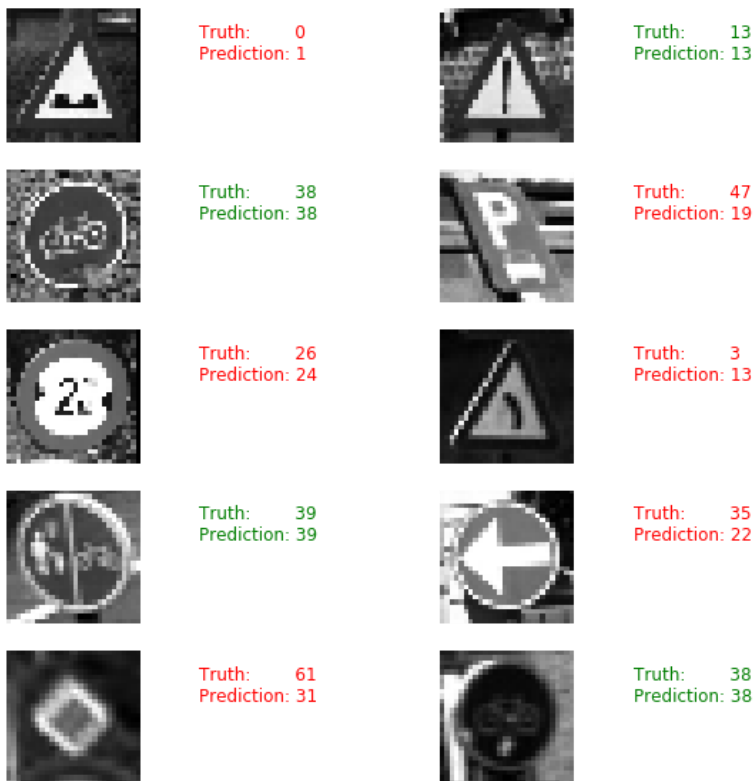


```

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], cmap="gray")

plt.show()

```



However, only looking at random images don't give you many insights into how well your model actually performs. That's why you'll load in the test data.

Note that you make use of the `load_data()` function, which you defined at the start of this tutorial.

```

# Import `skimage`
from skimage import transform

# Load the test data
test_images, test_labels = load_data(test_data_directory)

# Transform the images to 28 by 28 pixels
test_images28 = [transform.resize(image, (28, 28)) for image in test_images]

# Convert to grayscale
from skimage.color import rgb2gray
test_images28 = rgb2gray(np.array(test_images28))

# Run predictions against the full test set.
predicted = sess.run([correct_pred], feed_dict={x: test_images28})[0]

```

```
# Calculate correct matches
match_count = sum([int(y == y_) for y, y_ in zip(test_labels, predicted)])

# Calculate the accuracy
accuracy = match_count / len(test_labels)

# Print the accuracy
print("Accuracy: {:.3f}".format(accuracy))
```

Remember to close off the session with `sess.close()` in case you didn't use the `with tf.Session() as sess:` to start your TensorFlow session.

Where To Go Next?

If you want to continue working with this dataset and the model that you have put together in this tutorial, try out the following things:

- *Apply regularized LDA on the data before you feed it to your model. This is a suggestion that comes from [one of the original papers](#), written by the researchers that gathered and analyzed this dataset.*
- *You could also, as said in the tutorial itself, also look at some other data augmentation operations that you can perform on the traffic sign images. Additionally, you could also try to tweak this network further; The one that you have created now was fairly simple.*
- *Early stopping: Keep track of the training and testing error while you train the neural network. Stop training when both errors go down and then suddenly go back up - this is a sign that the neural network has started to overfit the training data.*
- *Play around with the optimizers.*

Make sure to check out the [Machine Learning With TensorFlow](#) book, written by Nishant Shukla.

Tip also check out the [TensorFlow Playground](#) and the [TensorBoard](#).

If you want to keep on working with images, definitely check out DataCamp's [scikit-learn tutorial](#), which tackles the MNIST dataset with the help of PCA, K-Means and Support Vector Machines (SVMs). Or take a look at other tutorials such as [this one](#) that uses the Belgian traffic signs dataset.