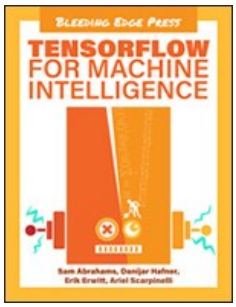


TensorFlow for Machine Intelligence: A Hands-On Introduction to Learning Algorithms

by Sam Abrahams, Danijar Hafner, Erik Erwitt and Ariel Scarpinelli

Chapter 5: Object Recognition and Classification

Overview

At this point, you should have a basic understanding of TensorFlow and its best practices. We'll follow these practices while we build a model capable of object recognition and classification. Building this model expands on the fundamentals that have been covered so far while adding terms, techniques and fundamentals of computer vision. The technique used in training the model has become popular recently due to its accuracy across challenges.

ImageNet, a database of labeled images, is where computer vision and deep learning saw a recent rise in popularity. Annually, ImageNet hosts a challenge (ILSVRC) where people build systems capable of automatically classifying and detecting objects based on ImageNet's database of images. In 2012, the challenge saw a team named **SuperVision** submit a solution using a creative neural network architecture. ILSVRC solutions are often creative but what set SuperVision's entry apart was its ability to accurately classify images. **SuperVision's entry** set a new standard for computer vision accuracy and stirred up interest in a deep learning technique named convolutional neural networks.

Convolutional Neural Networks (CNNs) have continued to grow in **popularity**. They're primarily used for computer vision related tasks but are not limited to working with images. CNNs could be used with any data that can be represented as a tensor where values are ordered next to related values (in a grid). **Microsoft Research** released a paper in 2014 where they used CNNs for speech recognition where the input tensor was a single row grid of sound frequencies ordered by the time they were recorded. For images, the values in the tensor are pixels ordered in a grid corresponding with the width and height of the image.

In this chapter, the focus is working with CNNs and images in TensorFlow. The goal is to build a CNN model using TensorFlow that categorizes images based on a subset of ImageNet's database. Training a CNN model will require working with images in TensorFlow and understanding how convolutional neural networks (CNNs) are used. The majority of the chapter is dedicated to introducing concepts of computer vision using TensorFlow.

The dataset used in training this CNN model is a subset of the images available in ImageNet named the **Stanford's Dogs Dataset**. As the name implies, this dataset is filled with images of different dog breeds and a label of the breed shown in the image. The goal of the model is to take an image and accurately guess the breed of dog shown in the image (example images are tagged as Siberian Husky from Stanford's Dog Datase).

If one of the images shown above is loaded into the model, it should output a label of Siberian Husky. These example images wouldn't be a fair test of the model's accuracy because they exist in the training dataset. Finding a fair metric to calculate the model's accuracy requires a large number of images which won't be used in training. The images which haven't been used in training the model will be used to create a separate test dataset.

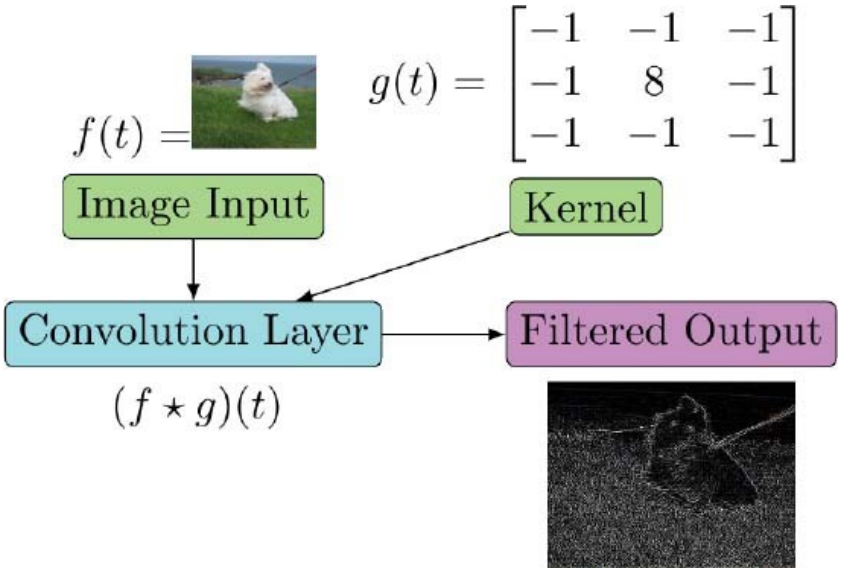
The reason to bring up the fairness of an image to test a model's accuracy is because it's part of keeping a separated test, train and cross-validation datasets. While processing input, it is a required practice to separate a large percentage of the data used to train a network. This separation is to allow a blind test of a model. Testing a model with input which was used to train it will likely create a model which accurately matches input it has already seen while not being capable of working with new input. The testing dataset is then used to see how well the model performs with data that didn't exist in the training. Over time and iterations of the model, it is possible that the changes being made to increase accuracy are making the model better equipped to the testing dataset while performing poorly in the real world. A good practice is to use a cross-validation dataset to check the final model and receive a better estimate of its accuracy. With images, it's best to separate the raw dataset while doing any preprocessing (color adjustments or cropping) keeping the input pipeline the same across all the datasets.

Convolutional Neural Networks

Technically, a convolutional neural network is a neural network which has at least one layer (`tf.nn.conv2d`) that does a convolution between its input *f* and a configurable kernel *g* generating the layer's output. In a simplified definition, a convolution's goal is to apply a kernel (filter) to every point in a tensor and generate a filtered output by sliding the kernel over an input tensor.

An example of the filtered output is edge detection in images. A special kernel is applied to each pixel of an image and the output is a new image depicting all the edges. In this case, the input tensor is an image and each point in the tensor is treated as a pixel which includes the

amount of red, green and blue found at that point. The kernel is slid over every pixel in the image and the output value increases whenever there is an edge between colors. This figure shows the simplified convolution layer where the input is an image and the output is all the horizontal lines found in the image.



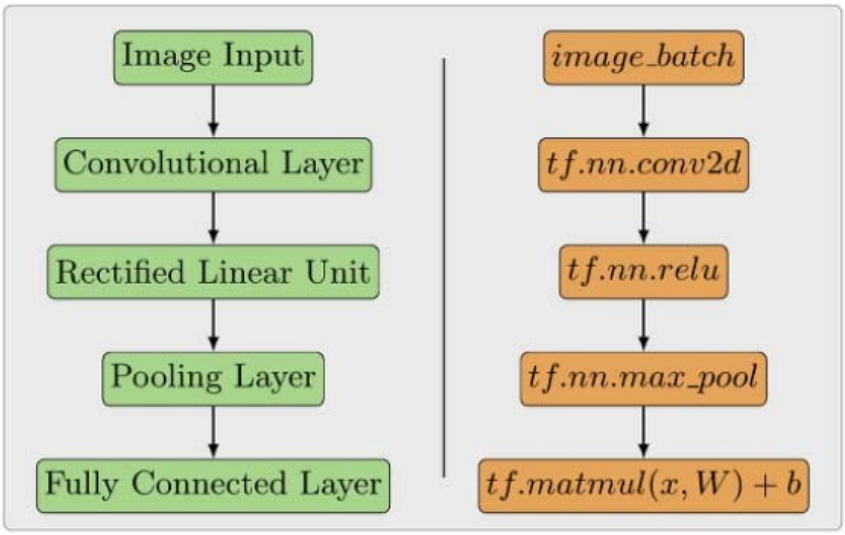
It isn't important to understand how convolutions combine input to generate filtered output, or what a kernel is, until later in this chapter when they're put in practice. Obtaining a broad sense of what a CNN does and its biological inspiration builds the technical implementation.

In 1968, **an article was published** detailing new findings on the cellular layout of a monkey striate cortex (the section of the brain thought to process visual input). The article discusses a grouping of cells that extend vertically combining together to match certain visual traits. The study of primate brains may seem irrelevant to a machine learning task, yet it was instrumental **in the development of deep learning** using CNNs.

CNNs follow a simplified process matching information similar to the structure found in the cellular layout of a monkey's striate cortex. As signals are passed through a monkey's striate cortex, certain layers signal when a visual pattern is highlighted. For example, one layer of cells activate (increase its output signal) when a horizontal line passes through it. A CNN will exhibit a similar behavior where clusters of neurons will activate based on patterns learned from training. For example, after training, a CNN will have certain layers that activate when a horizontal line passes through it.

Matching horizontal lines would be a useful neural network architecture. but CNNs take it further by layering multiple simple patterns to match complex patterns. In the context of CNNs, these patterns are known as filters or kernels and the goal is to adjust these kernel weights until they accurately match the training data. Training these filters is often accomplished by combining multiple different layers and learning weights using gradient descent.

A simple CNN architecture may combine a convolutional layer (`tf.nn.conv2d`), non-linearity layer (`tf.nn.relu`), pooling layer (`tf.nn.max_pool`) and a fully connected layer (`tf.matmul`). Without these layers, it's difficult to match complex patterns because the network will be filled with too much information. A well designed CNN architecture highlights important information while ignoring noise. We'll go into details on how these layers work together later in this chapter.



The input image for this architecture is a complex format designed to support the ability to load batches of images. Loading a batch of images allows the computation of multiple images simultaneously but it requires a more complex data structure. The data structure used is a rank four tensor including all the information required to convolve a batch of images. TensorFlow's input pipeline (which is used to read and decode files) has a special format designed to work with multiple images in a batch including required information for an image (`[image_batch_size, image_height, image_width, image_channels]`). Using the example code, it's possible to examine the structure of an example input used while working with images in TensorFlow.

```
image_batch = tf.constant([
        [  # First Image
            [[0, 255, 0], [0, 255, 0], [0, 255, 0]],
            [[0, 255, 0], [0, 255, 0], [0, 255, 0]]
        ],
        [  # Second Image
            [[0, 0, 255], [0, 0, 255], [0, 0, 255]],
            [[0, 0, 255], [0, 0, 255], [0, 0, 255]]
        ]
    ])
image_batch.get_shape()
```

The output from executing the example code is:

```
 TensorShape([Dimension(2), Dimension(2), Dimension(3), Dimension(3)])
```

NOTE: The example code and further examples in this chapter do not include the common bootstrapping required to run TensorFlow code. This includes importing the `tensorflow` (usually as `tf` for brevity), creating a TensorFlow session as `sess`, initializing all variables, and starting thread runners. Undefined variable errors may occur if the example code is executed without running these steps.

In this example code, a batch of images is created that includes two images. Each image has a height of two pixels and a width of three pixels with an RGB color space. The output from executing the example code shows the amount of images as the size of the first set of dimensions `Dimension(2)`, the height of each image as the size of the second set `Dimension(2)`, the width of each image as the third set `Dimension(3)`, and the size of the color channel as the final set `Dimension(3)`.

It's important to note each pixel maps to the height and width of the image. Retrieving the first pixel of the first image requires each dimension accessed as follows.

```
sess.run(image_batch)[0][0][0]
```

The output from executing the example code is:

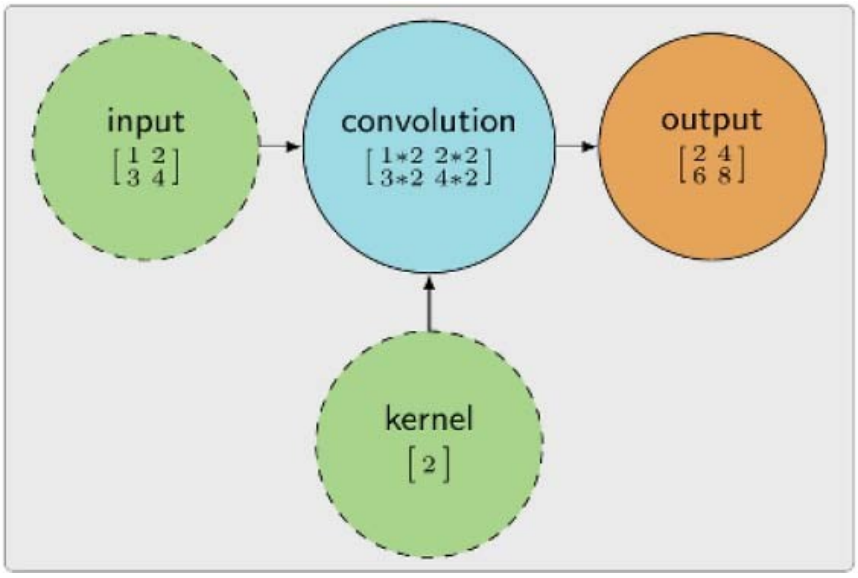
```
 array([ 0, 255, 0], dtype=int32)
```

Instead of loading images from disk, the `image_batch` variable will act as if it were images loaded as part of an input pipeline. Images loaded from disk using an input pipeline have the same format and act the same. It's often useful to create fake data similar to the `image_batch` example above to test input and output from a CNN. The simplified input will make it easier to debug any simple issues. It's important to work on simplification of debugging because CNN architectures are incredibly complex and often take days to train.

The first complexity working with CNN architectures is how a convolution layer works. After any image loading and manipulation, a convolution layer is often the first layer in the network. The first convolution layer is useful because it can simplify the rest of the network and be used for debugging. The next section will focus on how convolution layers operate and using them with TensorFlow.

Convolution

As the name implies, convolution operations are an important component of convolutional neural networks. The ability for a CNN to accurately match diverse patterns can be attributed to using convolution operations. These operations require complex input, which was shown in the previous section. In this section we'll experiment with convolution operations and the parameters that are available to tune them. Here the convolution operation convolves two input tensors (input and kernel) into a single output tensor, which represents information from each input.



Input and Kernel

Convolution operations in TensorFlow are done using `tf.nn.conv2d` in a typical situation. There are other convolution operations available using TensorFlow designed with special use cases. `tf.nn.conv2d` is the preferred convolution operation to begin experimenting with. For example, we can experiment with convolving two tensors together and inspect the result.

```
input_batch = tf.constant([
        [  # First Input
            [[0.0], [1.0]],
            [[2.0], [3.0]]
        ],
        [  # Second Input
            [[2.0], [4.0]],
            [[6.0], [8.0]]
        ]
    ])

kernel = tf.constant([
        [
            [[1.0, 2.0]]
        ]
    ])
```

The example code creates two tensors. The `input_batch` tensor has a similar shape to the `image_batch` tensor seen in the previous section. This will be the first tensor being convolved and the second tensor will be `kernel`. *Kernel* is an important term that is interchangeable with *weights*, *filter*, *convolution matrix* or *mask*. Since this task is computer vision related, it's useful to use the term kernel because it is being treated as an **image kernel**. There is no practical difference in the term when used to describe this functionality in TensorFlow. The parameter in TensorFlow is named `filter` and it expects a set of weights which will be learned from training. The amount of different weights included in the kernel (`filter` parameter) will configure the amount of kernels that will be learned.

In the example code, there is a single kernel which is the first dimension of the `kernel` variable. The kernel is built to return a tensor that will include one channel with the original input and a second channel with the original input doubled. In this case, channel is used to describe the elements in a rank 1 tensor (vector). Channel is a term from computer vision that describes the output vector, for example an RGB image has three channels represented as a rank 1 tensor `[red, green, blue]`. At this time, ignore the `strides` and `padding` parameter, which will be covered later, and focus on the convolution (`tf.nn.conv2d`) output.

```
conv2d = tf.nn.conv2d(input_batch, kernel, strides=[1, 1, 1, 1], pad-
ding='SAME')

sess.run(conv2d)
```

The output from executing the example code is:

```
 array([[[[ 0.,  0.],
    [ 1.,  2.]],
    [[ 2.,  4.],
    [ 3.,  6.]]],
    [[[ 2.,  4.],
    [ 4.,  8.]],
```

```
    [[ 6., 12.],
    [ 8., 16.]]]], dtype=float32)
```

The output is another tensor which is the same rank as the `input_batch` but includes the number of dimensions found in the kernel. Consider if `input_batch` represented an image, the image would have a single channel, in this case it could be considered a grayscale image. Each element in the tensor would represent one pixel of the image. The pixel in the bottom right corner of the image would have the value of `3.0`.

Consider the `tf.nn.conv2d` convolution operation as a combination of the image (represented as `input_batch`) and the `kernel` tenser. The convolution of these two tensors create a feature map. Feature map is a broad term except in computer vision where it relates to the output of operations which work with an image kernel. The feature map now represents the convolution of these tensors by adding new layers to the output.

The relationship between the input images and the output feature map can be explored with code. Accessing elements from the input batch and the feature map are done using the same index. By accessing the same pixel in both the input and the feature map shows how the input was changed when it convolved with the `kernel`. In the following case, the lower right pixel in the image was changed to output the value found by multiplying $3.0 * 1.0$ and $3.0 * 2.0$. The values correspond to the pixel value and the corresponding value found in the `kernel`.

```
lower_right_image_pixel = sess.run(input_batch)[0][1][1]
lower_right_kernel_pixel = sess.run(conv2d)[0][1][1]

lower_right_image_pixel, lower_right_kernel_pixel
```

The output from executing the example code is:

```
 (array([ 3.], dtype=float32), array([ 3., 6.], dtype=float32))
```

In this simplified example, each pixel of every image is multiplied by the corresponding value found in the kernel and then added to a corresponding layer in the feature map. Layer, in this context, is referencing a new dimension in the output. With this example, it's hard to see a value in convolution operations.

Strides

The value of convolutions in computer vision is their ability to reduce the dimensionality of the input, which is an image in this case. An image's dimensionality (2D image) is its width, height and number of channels. A large image dimensionality requires an exponentially larger amount of time for a neural network to scan over every pixel and judge which ones are important. Reducing dimensionality of an image with convolutions is done by altering the `strides` of the kernel.

The parameter `strides`, causes a kernel to skip over pixels of an image and not include them in the output. It's not fair to say the pixels are skipped because they still may affect the output. The `strides` parameter highlights how a convolution operation is working with a kernel when a larger image and more complex kernel are used. As a convolution is sliding the kernel over the input, it's using the strides parameter to change how it walks over the input. Instead of going over every element of an input, the `strides` parameter could configure the convolution to skip certain elements.

For example, take the convolution of a larger image and a larger kernel. In this case, it's a convolution between a 6 pixel tall, 6 pixel wide and 1 channel deep image (6x6x1) and a (3x3x1) kernel.

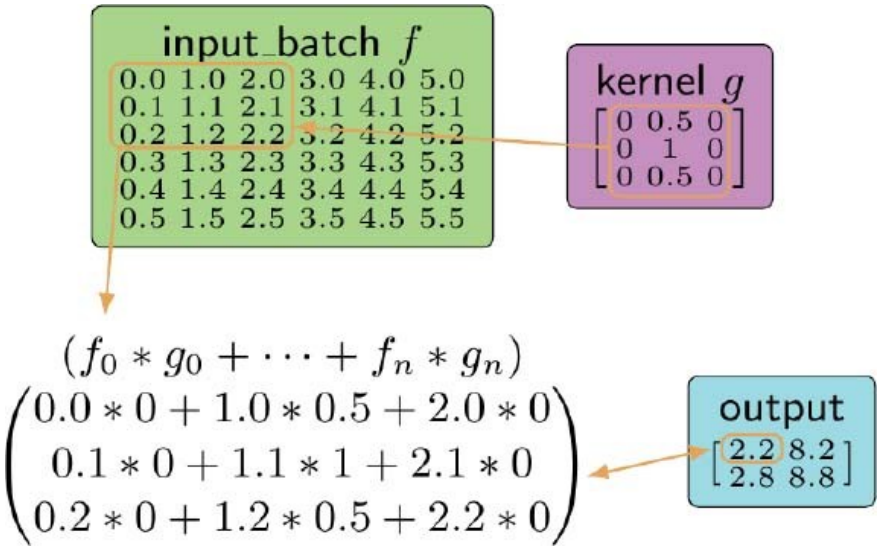
```
input_batch = tf.constant([
        [  # First Input (6x6x1)
            [[0.0], [1.0], [2.0], [3.0], [4.0], [5.0]],
            [[0.1], [1.1], [2.1], [3.1], [4.1], [5.1]],
            [[0.2], [1.2], [2.2], [3.2], [4.2], [5.2]],
            [[0.3], [1.3], [2.3], [3.3], [4.3], [5.3]],
            [[0.4], [1.4], [2.4], [3.4], [4.4], [5.4]],
            [[0.5], [1.5], [2.5], [3.5], [4.5], [5.5]],
        ],
    ])

kernel = tf.constant([  # Kernel (3x3x1)
        [[0.0]], [[0.5]], [[0.0]]],
        [[[0.0]], [[1.0]], [[0.0]]],
        [[[0.0]], [[0.5]], [[0.0]]]
    ])

# NOTE: the change in the size of the strides parameter.
conv2d = tf.nn.conv2d(input_batch, kernel, strides=[1, 3, 3, 1], pad-
ding='SAME')
sess.run(conv2d)
```

The output from executing the example code is:

```
 array([[[[ 2.20000005],
    [ 8.19999981]],
```

```
    [[ 2.79999995],
    [ 8.80000019]]]], dtype=float32)
```

The `input_batch` was combined with the `kernel` by moving the `kernel` over the `input_batch` striding (or skipping) over certain elements. Each time the `kernel` was moved, it get centered over an element of `input_batch`. Then the overlapping values are multiplied together and the result is added together. This is how a convolution combines two inputs using what's referred to as pointwise multiplication. It may be easier to visualize using the following figure.



In this figure, the same logic follows what is found in the code. Two tensors convolved together while striding over the input. The strides reduced the dimensionality of the output a large amount while the kernel size allowed the convolution to use all the input values. None of the input data was completely removed from striding but now the input is a smaller tensor.

Strides are a way to adjust the dimensionality of input tensors. Reducing dimensionality requires less processing power, and will keep from creating receptive fields which completely overlap. The `strides` parameter follows the same format as the input tensor `[image_batch_size_stride, image_height_stride, image_width_stride, image_channels_stride]`. Changing the first or last element of the stride parameter are rare, they'd skip data in a `tf.nn.conv2d` operation and not take the input into account. The `image_height_stride` and `image_width_stride` are useful to alter in reducing input dimensionality.

A challenge that comes up often with striding over the input is how to deal with a stride which doesn't evenly end at the edge of the input. The uneven striding will come up often due to image size and kernel size not matching the striding. If the image size, kernel size and strides can't be changed then padding can be added to the image to deal with the uneven area.

Padding

When a kernel is overlapped on an image it should be set to fit within the bounds of the image. At times, the sizing may not fit and a good alternative is to fill the missing area in the image. Filling the missing area of the image is known as padding the image. TensorFlow will pad the image with zeros or raise an error when the sizes don't allow a kernel to stride over an image without going past its bounds. The amount of zeros or the error state of `tf.nn.conv2d` is controlled by the parameter `padding` which has two possible values ('VALID', 'SAME').

SAME: The convolution output is the **SAME** size as the input. This doesn't take the filter's size into account when calculating how to stride over the image. This may stride over more of the image than what exists in the bounds while padding all the missing values with zero.

VALID: Take the filter's size into account when calculating how to stride over the image. This will try to keep as much of the kernel inside the image's bounds as possible. There may be padding in some cases but will avoid.

It's best to consider the size of the input but if padding is necessary then TensorFlow has the option built in. In most simple scenarios, `SAME` is a good choice to begin with. `VALID` is preferential when the input and kernel work well with the strides. For further information, TensorFlow covers this subject well in the **convolution documentation**.

Data Format

There's another parameter to `tf.nn.conv2d` which isn't shown from these examples named `data_format`. The **tf.nn.conv2d docs** explain how to change the data format so the `input`, `kernel` and `strides` follow a format other than the format being used thus far. Changing this format is useful if there is an input tensor which doesn't follow the `[batch_size, height, width, channel]` standard. Instead of changing the input to match, it's possible to change the `data_format` parameter to use a different layout.

> `data_format`: An optional string from: "NHWC", "NCHW". Defaults to "NHWC". Specify the data format of the input and output

data. With the default format "NHWC", the data is stored in the order of: [`batch`, `in_height`, `in_width`, `in_channels`]. Alternatively, the format could be "NCHW", the data storage order of: [`batch`, `in_channels`, `in_height`, `in_width`].

Data Format	Definition
N	Number of tensors in a batch, the batch_size.
H	Height of the tensors in each batch.
W	Width of the tensors in each batch.
C	Channels of the tensors in each batch.

Kernels in Depth

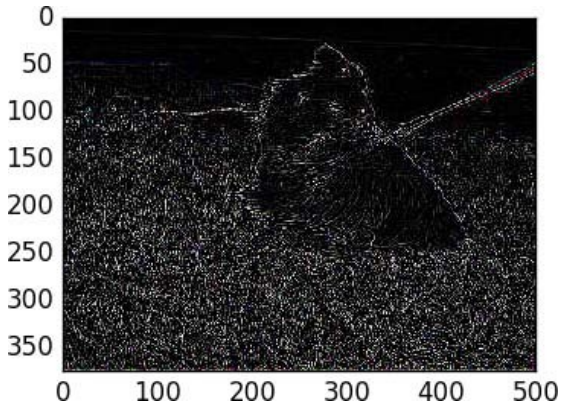
In TensorFlow the filter parameter is used to specify the kernel convolved with the input. Filters are commonly used in photography to adjust attributes of a picture, such as the amount of sunlight allowed to reach a camera's lens. In photography, filters allow a photographer to drastically alter the picture they're taking. The reason the photographer is able to alter their picture using a filter is because the filter can recognize certain attributes of the light coming in to the lens. For example, a red lens filter will absorb (block) every frequency of light which isn't red allowing only red to pass through the filter.

Before and after applying a minor red filter to n02088466_3184.jpg.

In computer vision, kernels (filters) are used to recognize important attributes of a digital image. They do this by using certain patterns to highlight when features exist in an image. A kernel which will replicate the red filter example image is implemented by using a reduced value for all colors except red. In this case, the reds will stay the same but all other colors matched are reduced.

The example seen at the start of this chapter uses a kernel designed to do edge detection. Edge detection kernels are common in computer vision applications and could be implemented using basic TensorFlow operations and a single `tf.nn.conv2d` operation.

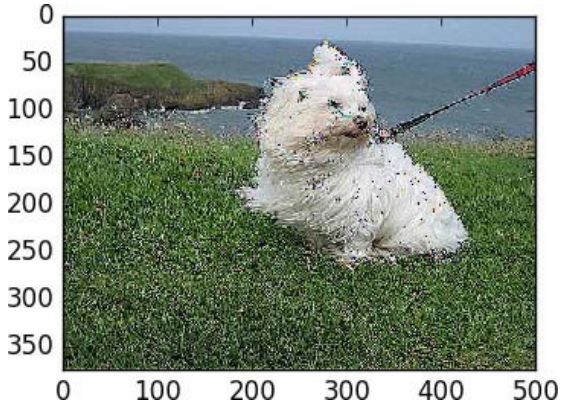
```
kernel = tf.constant([
        [
            [[ -1., 0., 0.], [ 0., -1., 0.], [ 0., 0., -1.]],
            [[ -1., 0., 0.], [ 0., -1., 0.], [ 0., 0., -1.]],
            [[ -1., 0., 0.], [ 0., -1., 0.], [ 0., 0., -1.]]
        ],
        [
            [[ -1., 0., 0.], [ 0., -1., 0.], [ 0., 0., -1.]],
            [[ 8., 0., 0.], [ 0., 8., 0.], [ 0., 0., 8.]],
            [[ -1., 0., 0.], [ 0., -1., 0.], [ 0., 0., -1.]]
        ],
        [
            [[ -1., 0., 0.], [ 0., -1., 0.], [ 0., 0., -1.]],
            [[ -1., 0., 0.], [ 0., -1., 0.], [ 0., 0., -1.]],
            [[ -1., 0., 0.], [ 0., -1., 0.], [ 0., 0., -1.]]
        ]
    ])

conv2d = tf.nn.conv2d(image_batch, kernel, [1, 1, 1, 1], padding="SAME")
activation_map = sess.run(tf.minimum(tf.nn.relu(conv2d), 255))
```



The output created from convolving an image with an edge detection kernel are all the areas where and edge was detected. The code assumes a batch of images is already available (`image_batch`) with a real image loaded from disk. In this case, the image is an example image found in the Stanford Dogs Dataset. The kernel has three input and three output channels. The channels sync up to RGB values between [0, 255] with 255 being the maximum intensity. The `tf.minimum` and `tf.nn.relu` calls are there to keep the convolution values within the range of valid RGB colors of [0, 255].

There are **many other**) common kernels which can be used in this simplified example. Each will highlight different patterns in an image with different results. The following kernel will sharpen an image by increasing the intensity of color changes.

```
kernel = tf.constant([
        [
            [[ 0., 0., 0.], [ 0., 0., 0.], [ 0., 0., 0.]],
            [[ -1., 0., 0.], [ 0., -1., 0.], [ 0., 0., -1.]],
            [[ 0., 0., 0.], [ 0., 0., 0.], [ 0., 0., 0.]]
        ],
        [
            [[ -1., 0., 0.], [ 0., -1., 0.], [ 0., 0., -1.]],
            [[ 5., 0., 0.], [ 0., 5., 0.], [ 0., 0., 5.]],
            [[ -1., 0., 0.], [ 0., -1., 0.], [ 0., 0., -1.]]
        ],
        [
            [[ 0., 0., 0.], [ 0., 0., 0.], [ 0., 0., 0.]],
            [[ -1., 0., 0.], [ 0., -1., 0.], [ 0., 0., -1.]],
            [[ 0, 0., 0.], [ 0., 0., 0.], [ 0., 0., 0.]]
        ]
    ])

conv2d = tf.nn.conv2d(image_batch, kernel, [1, 1, 1, 1], padding="SAME")
activation_map = sess.run(tf.minimum(tf.nn.relu(conv2d), 255))
```



The values in the kernel were adjusted with the center of the kernel increased in intensity and the areas around the kernel reduced in intensity. The change, matches patterns with intense pixels and increases their intensity outputting an image which is visually sharpened. Note that the corners of the kernel are all `0` and don't affect the output that operates in a plus shaped pattern.

These kernels match patterns in images at a rudimentary level. A convolutional neural network matches edges and more by using a complex kernel it learned during training. The starting values for the kernel are usually random and over time they're trained by the CNN's learning layer. When a CNN is complete, it starts running and each image sent in is convolved with a kernel which is then changed based on if the predicted value matches the labeled value of the image. For example, if a Sheepdog picture is considered a Pit Bull by the CNN being trained it will then

change the filters a small amount to try and match Sheepdog pictures better.

Learning complex patterns with a CNN involves more than a single layer of convolution. Even the example code included a `tf.nn.relu` layer used to prepare the output for visualization. Convolution layers may occur more than once in a CNN but they'll likely include other layer types as well. These layers combined form the support network required for a successful CNN architecture.

Common Layers

For a neural network architecture to be considered a CNN, it requires at least one convolution layer (`tf.nn.conv2d`). There are practical uses for a single layer CNN (edge detection), for image recognition and categorization it is common to use different layer types to support a convolution layer. These layers help reduce over-fitting, speed up training and decrease memory usage.

The layers covered in this chapter are focused on layers commonly used in a CNN architecture. A CNN isn't limited to use only these layers, they can be mixed with layers designed for other network architectures.

Convolution Layers

One type of convolution layer has been covered in detail (`tf.nn.conv2d`) but there are a few notes which are useful to advanced users. The convolution layers in TensorFlow don't do a full convolution, details can be found in **the TensorFlow API documentation**. In practice, the difference between a convolution and the operation TensorFlow uses is performance. TensorFlow uses a technique to speed up the convolution operation in all the different types of convolution layers.

There are use cases for each type of convolution layer but for `tf.nn.conv2d` is a good place to start. The other types of convolutions are useful but not required in building a network capable of object recognition and classification. A brief summary of each is included.

TF.NN.DEPTHWISE_CONV2D

This convolution is used when attaching the output of one convolution to the input of another convolution layer. An advanced use case is using a `tf.nn.depthwise_conv2d` to create a network following the **inception architecture**.

TF.NN.SEPARABLE_CONV2D

This is similar to `tf.nn.conv2d`, but not a replacement for it. For large models, it speeds up training without sacrificing accuracy. For small models, it will converge quickly with worse accuracy.

TF.NN.CONV2D_TRANSPOSE

This applies a kernel to a new feature map where each section is filled with the same values as the kernel. As the kernel strides over the new image, any overlapping sections are summed together. There is a great explanation on how `tf.nn.conv2d_transpose` is used for learnable upsampling in **Stanford's CS231n Winter 2016: Lecture 13**.

Activation Functions

These functions are used in combination with the output of other layers to generate a feature map. They're used to smooth (or differentiate) the results of certain operations. The goal is to introduce non-linearity into the neural network. Non-linearity means that the input is a curve instead of a straight line. Curves are capable of representing more complex changes in input. For example, non-linear input is capable of describing input which stays small for the majority of the time but periodically has a single point at an extreme. Introduction of non-linearity in a neural network allows it to train on the complex patterns found in data.

TensorFlow has **multiple activation functions** available. With CNNs, `tf.nn.relu` is primarily used because of its performance although it sacrifices information. When starting out, using `tf.nn.relu` is recommended but advanced users may create their own. When considering if an activation function is useful there are a few primary considerations.

1. The function is **monotonic**, so its output should always be increasing or decreasing along with the input. This allows gradient descent optimization to search for local minima.
2. The function is **differentiable**, so there must be a derivative at any point in the function's domain. This allows gradient descent optimization to properly work using the output from this style of activation function.

Any functions that satisfy those considerations could be used as activation functions. In TensorFlow there are a few worth highlighting which are common to see in CNN architectures. A brief summary of each is included with a small sample code illustrating their usage.

TF.NN.RELU

A rectifier (rectified linear unit) called a ramp function in some documentation and looks like a skateboard ramp when plotted. ReLU is linear and keeps the same input values for any positive numbers while setting all negative numbers to be 0. It has the benefits that it doesn't suffer from **gradient vanishing** and has a range of [0, +∞]. A drawback of ReLU is that it can suffer from neurons becoming saturated when too high of a learning rate is used.

```
features = tf.range(-2, 3)
# Keep note of the value for negative features
sess.run([features, tf.nn.relu(features)])
```

The output from executing the example code is:

```
[array([-2, -1, 0, 1, 2], dtype=int32), array([0, 0, 0, 1, 2], dtype=int32)]
```

In this example, the input in a rank one tensor (vector) of integer values between [– 2,3]. A `tf.nn.relu` is ran over the values the output highlights that any value less than 0 is set to be 0. The other input values are left untouched.

TF.SIGMOID

A sigmoid function returns a value in the range of [0 . 0, 1 . 0]. Larger values sent into a `tf.sigmoid` will trend closer to 1.0 while smaller values will trend towards 0.0. The ability for sigmoids to keep a values between [0 . 0, 1 . 0] is useful in networks which train on probabilities which are in the range of [0 . 0, 1 . 0]. The reduced range of output values can cause trouble with input becoming saturated and changes in input becoming exaggerated.

```
# Note, tf.sigmoid (tf.nn.sigmoid) is currently limited to float values
features = tf.to_float(tf.range(-1, 3))
sess.run([features, tf.sigmoid(features)])
```

The output from executing the example code is:

```
[array([-1., 0., 1., 2.], dtype=float32),
   array([ 0.26894143, 0.5, 0.7310586, 0.88079703], dtype=float32)]
```

In this example, a range of integers is converted to be float values (`1` becomes `1.0`) and a sigmoid function is ran over the input features. The result highlights that when a value of 0.0 is passed through a sigmoid, the result is 0.5 which is the midpoint of the simoid's domain. It's useful to note that with 0.5 being the sigmoid's midpoint, negative values can be used as input to a sigmoid.

TF.TANH

A hyperbolic tangent function (tanh) is a close relative to `tf.sigmoid` with some of the same benefits and drawbacks. The main difference between `tf.sigmoid` and `tf.tanh` is that `tf.tanh` has a range of [– 1 . 0, 1 . 0]. The ability to output negative values may be useful in certain network architectures.

```
# Note, tf.tanh (tf.nn.tanh) is currently limited to float values
features = tf.to_float(tf.range(-1, 3))
sess.run([features, tf.tanh(features)])
```

The output from executing the example code is:

```
[array([-1., 0., 1., 2.], dtype=float32),
   array([-0.76159418, 0., 0.76159418, 0.96402758], dtype=float32)]
```

In this example, all the setup is the same as the `tf.sigmoid` example but the output shows an important difference. In the output of `tf.tanh` the midpoint is 0.0 with negative values. This can cause trouble if the next layer in the network isn't expecting negative input or input of 0.0.

TF.NN.DROPOUT

Set the output to be 0.0 based on a configurable probability. This layer performs well in scenarios where a little randomness helps training. An example scenario is when there are patterns being learned that are too tied to their neighboring features. This layer will add a little noise to the output being learned.

NOTE: This layer should only be used during training because the random noise it adds will give misleading results while testing.

```
features = tf.constant([-0.1, 0.0, 0.1, 0.2])
# Note, the output should be different on almost ever execution. Your num-
bers won't match
# this output.
sess.run([features, tf.nn.dropout(features, keep_prob=0.5)])
```

The output from executing the example code is:

```
[array([-0.1, 0., 0.1, 0.2], dtype=float32),
   array([-0., 0., 0.2, 0.40000001], dtype=float32)]
```

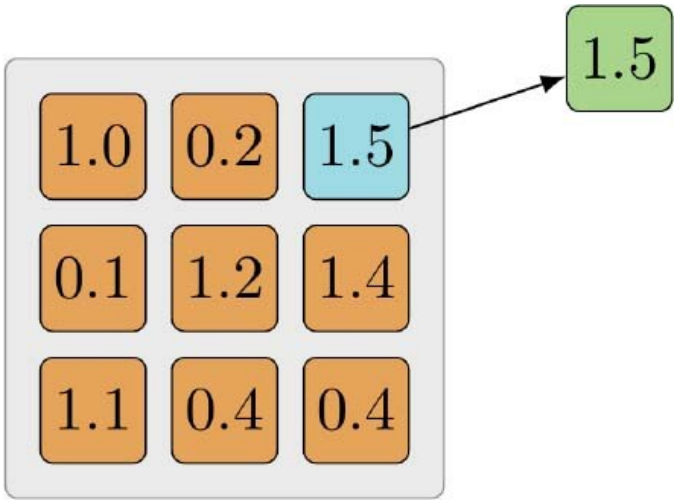
In this example, the output has a 50% probability of being kept. Each execution of this layer will have different output (most likely, it's somewhat random). When an output is dropped, its value is set to 0.0.

Pooling Layers

Pooling layers reduce over-fitting and improving performance by reducing the size of the input. They're used to scale down input while keeping important information for the next layer. It's possible to reduce the size of the input using a `tf.nn.conv2d` alone but these layers execute much faster.

TF.NN.MAX_POOL

Strides over a tensor and chooses the maximum value found within a certain kernel size. Useful when the intensity of the input data is relevant to importance in the image.



The same example is modeled using example code below. The goal is to find the largest value within the tensor.

```
# Usually the input would be output from a previous layer and not an image
directly.
batch_size=1
input_height = 3
input_width = 3
input_channels = 1

layer_input = tf.constant([
        [
            [[1.0], [0.2], [1.5]],
            [[0.1], [1.2], [1.4]],
            [[1.1], [0.4], [0.4]]
        ]
    ])

# The strides will look at the entire input by using the image_height and
image_width
kernel = [batch_size, input_height, input_width, input_channels]
max_pool = tf.nn.max_pool(layer_input, kernel, [1, 1, 1, 1], "VALID")
sess.run(max_pool)
```

The output from executing the example code is:

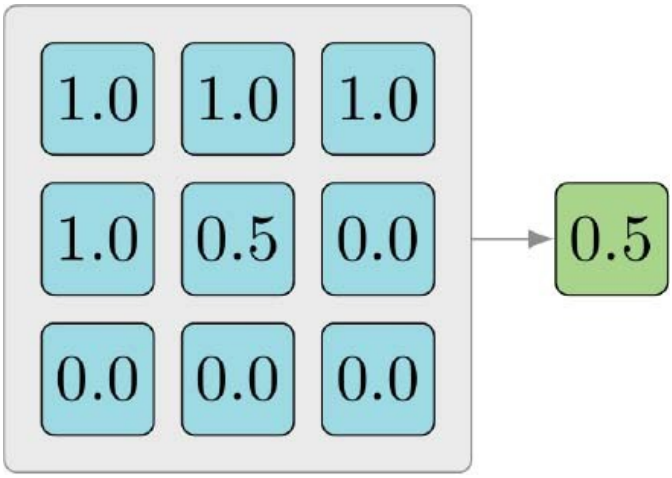
```
 array([[[[ 1.5]]]], dtype=float32)
```

The `layer_input` is a tensor with a shape similar to the output of `tf.nn.conv2d` or an activation function. The goal is to keep only one value, the largest value in the tensor. In this case, the largest value of the tensor is `1.5` and is returned in the same format as the input. If the `kernel` were set to be smaller, it would choose the largest value in each kernel size as it strides over the image.

Max-pooling will commonly be done using `2x2` receptive field (kernel with a height of 2 and width of 2) which is often written as a "2x2 max-pooling operation". One reason to use a `2x2` receptive field is that it's the smallest amount of downsampling which can be done in a single pass. If a `1x1` receptive field were used then the output would be the same as the input.

TF.NN.AVG_POOL

Strides over a tensor and averages all the values at each depth found within a kernel size. Useful when reducing values where the entire kernel is important, for example, input tensors with a large width and height but small depth.



The same example is modeled using example code below. The goal is to find the average of all the values within the tensor.

```
batch_size=1
input_height = 3
input_width = 3
input_channels = 1

layer_input = tf.constant([
        [
            [[1.0], [1.0], [1.0]],
            [[1.0], [0.5], [0.0]],
            [[0.0], [0.0], [0.0]]
        ]
    ])

# The strides will look at the entire input by using the image_height and
image_width
kernel = [batch_size, input_height, input_width, input_channels]
max_pool = tf.nn.avg_pool(layer_input, kernel, [1, 1, 1, 1], "VALID")
sess.run(max_pool)
```

The output from executing the example code is:

```
 array([[[[ 0.5]]]], dtype=float32)
```

Do a summation of all the values in the tensor, then divide them by the size of the number of scalars in the tensor:

$$\frac{1.0+1.0+1.0+1.0+0.5+0.0+0.0+0.0+0.0}{9.0}$$

This is exactly what the example code did above, but by reducing the size of the kernel, it's possible to adjust the size of the output.

Normalization

Normalization layers are not unique to CNNs and aren't used as often. When using `tf.nn.relu`, it is useful to consider normalization of the output. Since ReLU is unbounded, it's often useful to utilize some form of normalization to identify high-frequency features.

TF.NN.LOCAL_RESPONSE_NORMALIZATION (TF.NN.LRN)

Local response normalization is a function which shapes the output based on a summation operation best explained in **TensorFlow's documentation**.

> … Within a given vector, each component is divided by the weighted, squared sum of inputs within depth_radius.

One goal of normalization is to keep the input in a range of acceptable numbers. For instance, normalizing input in the range of [0.0,1.0] where the full range of possible values is normalized to be represented by a number greater than or equal to `0.0` and less than or equal to `1.0`. Local response normalization normalizes values while taking into account the significance of each value.

Cuda-Convnet includes further details on why using local response normalization is useful in some CNN architectures. **ImageNet** uses this layer to normalize the output from `tf.nn.relu`.

```
# Create a range of 3 floats.
#  TensorShape([batch, image_height, image_width, image_channels])
```

```
layer_input = tf.constant([
        [[[ 1.]], [[ 2.]], [[ 3.]]]
    ])

lrn = tf.nn.local_response_normalization(layer_input)
sess.run([layer_input, lrn])
```

The output from executing the example code is:

```
 [array([[[[ 1.]],
    [[ 2.]],
    [[ 3.]]]], dtype=float32), array([[[[ 0.70710677]],
    [[ 0.89442718]],
    [[ 0.94868326]]]], dtype=float32)]
```

In this example code, the layer input is in the format `[batch, image_height, image_width, image_channels]`. The normalization reduced the output to be in the range of [-1.0.1.0]. For `tf.nn.relu`, this layer will reduce its unbounded output to be in the same range.

High Level Layers

TensorFlow has introduced high level layers designed to make it easier to create fairly standard layer definitions. These aren't required to use but they help avoid duplicate code while following best practices. While getting started, these layers add a number of non-essential nodes to the graph. It's worth waiting until the basics are comfortable before using these layers.

TF.CONTRIB.LAYERS.CONVOLUTION2D

The `convolution2d` layer will do the same logic as `tf.nn.conv2d` while including weight initialization, bias initialization, trainable variable output, bias addition and adding an activation function. Many of these steps haven't been covered for CNNs yet but should be familiar. A kernel is a trainable variable (the CNN's goal is to train this variable), weight initialization is used to fill the kernel with values (`tf.truncated_normal`) on its first run. The rest of the parameters are similar to what have been used before except they are reduced to short-hand version. Instead of declaring the full kernel, now it's a simple tuple `(1,1)` for the kernel's height and width.

```
image_input = tf.constant([
            [
                [[0., 0., 0.], [255., 255., 255.], [254., 0., 0.]],
                [[0., 191., 0.], [3., 108., 233.], [0., 191., 0.]],
                [[254., 0., 0.], [255., 255., 255.], [0., 0., 0.]]
            ]
        ])

conv2d = tf.contrib.layers.convolution2d(
    image_input,
    num_output_channels=4,
    kernel_size=(1,1),          # It's only the filter height and width.
    activation_fn=tf.nn.relu,
    stride=(1, 1),              # Skips the stride values for image_batch
and input_channels.
    trainable=True)

# It's required to initialize the variables used in convolution2d's setup.
sess.run(tf.initialize_all_variables())
sess.run(conv2d)
```

The output from executing the example code is:

```
 array([[[[ 0., 0., 0., 0.],
    [ 166.44549561, 0., 0., 0.],
    [ 171.00466919, 0., 0., 0.]],
    [[ 28.54177475, 0., 59.9046936, 0.],
    [ 0., 124.69891357, 0., 0.],
    [ 28.54177475, 0., 59.9046936, 0.]],
    [[ 171.00466919, 0., 0., 0.],
    [ 166.44549561, 0., 0., 0.],
    [ 0., 0., 0., 0.]]]], dtype=float32)
```

This example sets up a full convolution against a batch of a single image. All the parameters are based off of the steps done throughout this chapter. The main difference is that `tf.contrib.layers.convolution2d` does a large amount of setup without having to write it all again. This can be a great time saving layer for advanced users.

NOTE: `tf.to_float` should not be used if the input is an image, instead use `tf.image.convert_image_dtype` which will properly change the range of values used to describe colors. In this example code, float values of `255.` were used which aren't what TensorFlow

expects when is sees an image using float values. TensorFlow expects an image with colors described as floats to stay in the range of [0,1].

TF.CONTRIB.LAYERS.FULLY_CONNECTED

A fully connected layer is one where every input is connected to every output. This is a fairly common layer in many architectures but for CNNs, the last layer is quite often fully connected. The `tf.contrib.layers.fully_connected` layer offers a great short-hand to create this last layer while following best practices.

Typical fully connected layers in TensorFlow are often in the format of `tf.matmul(features, weight) + bias` where `feature`, `weight` and `bias` are all tensors. This short-hand layer will do the same thing while taking care of the intricacies involved in managing the `weight` and `bias` tensors.

```
features = tf.constant([
        [[1.2], [3.4]]
    ])

fc = tf.contrib.layers.fully_connected(features, num_output_units=2)
# It's required to initialize all the variables first or there'll be an er-
ror about precondition failures.
sess.run(tf.initialize_all_variables())
sess.run(fc)
```

The output from executing the example code is:

```
 array([[[-0.53210509, 0.74457598],
    [-1.50763106, 2.10963178]]], dtype=float32)
```

This example created a fully connected layer and associated the input tensor with each neuron of the output. There are plenty of other parameters to tweak for different fully connected layers.

LAYER INPUT

Each layer serves a purpose in a CNN architecture. It's important to understand them at a high level (at least) but without practice they're easy to forget. A crucial layer in any neural network is the input layer, where raw input is sent to be trained and tested. For object recognition and classification, the input layer is a `tf.nn.conv2d` layer which accepts images. The next step is to use real images in training instead of example input in the form of `tf.constant` or `tf.range` variables.

Images and TensorFlow

TensorFlow is designed to support working with images as input to neural networks. TensorFlow supports loading common file formats (JPG, PNG), working in different color spaces (RGB, RGBA) and common image manipulation tasks. TensorFlow makes it easier to work with images but it's still a challenge. The largest challenge working with images are the size of the tensor which is eventually loaded. Every image requires a tensor the same size as the image's *height * width * channels*. As a reminder, channels are represented as a rank 1 tensor including a scalar amount of color in each channel.

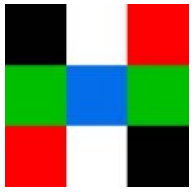
A red RGB pixel in TensorFlow would be represented with the following tensor.

```
red = tf.constant([255, 0, 0])
```

Each scalar can be changed to make the pixel another color or a mix of colors. The rank 1 tensor of a pixel is in the format of `[red, green, blue]` for an RGB color space. All the pixels in an image are stored in files on a disk which need to be read into memory so TensorFlow may operate on them.

Loading images

TensorFlow is designed to make it easy to load files from disk quickly. Loading images is the same as loading any other large binary file until the contents are decoded. Loading this example 3x3 pixel RGB JPG image is done using a similar process to loading any other type of file.



```
# The match_filenames_once will accept a regex but there is no need for this
example.
image_filename = "./images/chapter-05-object-recognition-and-classification/
working-with-images/test-input-image.jpg"
filename_queue = tf.train.string_input_producer(
```

```
    tf.train.match_filenames_once(image_filename))

image_reader = tf.WholeFileReader()
_, image_file = image_reader.read(filename_queue)
image = tf.image.decode_jpeg(image_file)
```

The image, which is assumed to be located in a relative directory from where this code is ran. An input producer (`tf.train.string_input_producer`) finds the files and adds them to a queue for loading. Loading an image requires loading the entire file into memory (`tf.WholeFileReader`) and onces a file has been read (`image_reader.read`) the resulting image is decoded (`tf.image.decode_jpeg`).

Now the image can be inspected, since there is only one file by that name the queue will always return the same image.

```
sess.run(image)
```

The output from executing the example code is:

```
array([[[ 0,  0,  0],
    [255, 255, 255],
    [254,  0,  0]],
    [[ 0, 191,  0],
    [ 3, 108, 233],
    [ 0, 191,  0]],
    [[254,  0,  0],
    [255, 255, 255],
    [ 0,  0,  0]]], dtype=uint8)
```

Inspect the output from loading an image, notice that it's a fairly simple rank 3 tensor. The RGB values are found in 9 rank 1 tensors. The higher rank of the image should be familiar from earlier sections. The format of the image loaded in memory is now `[batch_size, image_height, image_width, channels]`.

The `batch_size` in this example is 1 because there are no batching operations happening. Batching of input is covered in **the TensorFlow documentation** with a great amount of detail. When dealing with images, note the amount of memory required to load the raw images. If the images are too large or too many are loaded in a batch, the system may stop responding.

Image Formats

It's important to consider aspects of images and how they affect a model. Consider what would happen if a network is trained with input from a single frame of a **RED Weapon Camera**, which at the time of writing this, has an effective pixel count of `6144x3160`. That'd be 19,415,040 rank one tensors with 3 dimensions of color information.

Practically speaking, an input of that size will use a huge amount of system memory. Training a CNN takes a large amount of time and loading very large files slow it down more. Even if the increase in time is acceptable, the size a single image would be hard to fit in memory on the majority of system's GPUs.

A large input image is counterproductive to training most CNNs as well. The CNN is attempting to find inherent attributes in an image, which are unique but generalized so that they may be applied to other images with similar results. Using a large input is flooding a network with irrelevant information which will keep from generalizing the model.

In the **Stanford Dogs Dataset** there are two extremely different images of the same dog breed which should both match as a Pug. Although cute, these images are filled with useless information which mislead a network during training. For example, the hat worn by the Pug in n02110958_4030.jpg isn't a feature a CNN needs to learn in order to match a Pug. Most Pugs prefer pirate hats so the jester hat is training the network to match a hat which most Pugs don't wear.

n02110958_2410.jpg n02110958_4030.jpg

Highlighting important information in images is done by storing them in an appropriate file format and manipulating them. Different formats can be used to solve different problems encountered while working with images.

JPEG and PNG

TensorFlow has two image formats used to decode image data, one is `tf.image.decode_jpeg` and the other is `tf.image.decode_png`. These are common file formats in computer vision applications because they're trivial to convert other formats to.

Something important to keep in mind, JPEG images don't store any alpha channel information and PNG images do. This could be important if what you're training on requires alpha information (transparency). An example usage scenario is one where you've manually cut out some pieces of an image, for example, irrelevant jester hats on dogs. Setting those pieces to black would make them seem of similar importance to other black colored items in the image. Setting the removed hat to have an alpha of 0 would help in distinguishing its removal.

When working with JPEG images, don't manipulate them too much because it'll leave **artifacts**. Instead, plan to take raw images and export them to JPEG while doing any manipulation needed. Try to manipulate images before loading them whenever possible to save time in training.

PNG images work well if manipulation is required. PNG format is lossless so it'll keep all the information from the original file (unless they've been resized or downsampled). The downside to PNGs is that the files are larger than their JPEG counterpart.

TFRECORD

TensorFlow has a built-in file format designed to keep binary data and label (category for training) data in the same file. The format is called TFRecord and the format requires a preprocessing step to **convert images** to a TFRecord format before training. The largest benefit is keeping each input image in the same file as the label associated with it.

Technically, TFRecord files are protobuf formatted files. They are great for use as a preprocessed format because they aren't compressed and can be loaded into memory quickly. In this example, an image is written to a new TFRecord formatted file and it's label is stored as well.

```
# Reuse the image from earlier and give it a fake label
image_label = b'\x01'  # Assume the label data is in a one-hot representa-
tion (00000001)

# Convert the tensor into bytes, notice that this will load the entire image
file
image_loaded = sess.run(image)
image_bytes = image_loaded.tobytes()
image_height, image_width, image_channels = image_loaded.shape

# Export TFRecord
writer = tf.python_io.TFRecordWriter("./output/training-image.tfrecord")

# Don't store the width, height or image channels in this Example file to
save space but not required.
example = tf.train.Example(features=tf.train.Features(feature={
            'label': tf.train.Feature(bytes_list=tf.train.BytesList(val-
ue=[image_label])),
            'image': tf.train.Feature(bytes_list=tf.train.BytesList(val-
ue=[image_bytes]))
        }))

# This will save the example to a text file tfrecord
writer.write(example.SerializeToString())
writer.close()
```

The label is in a format known as one-hot encoding which is a common way to work with label data for categorization of multi-class data. The Stanford Dogs Dataset is being treated as multi-class data because the dogs are being categorized as a single breed and not a mix of breeds. In the real world, a multilabel solution would work well to predict dog breeds because it'd be capable of matching a dog with multiple breeds.

In the example code, the image is loaded into memory and converted into an array of bytes. The bytes are then added to the `tf.train.Example` file which are serialized `SerializeToString` before storing to disk. Serialization is a way of converting the in memory object into a format safe to be transferred to a file. The serialized example is now saved in a format which can be loaded and deserialized back to the example format saved here.

Now that the image is saved as a TFRecord it can be loaded again but this time from the TFRecord file. This would be the loading required in a training step to load the image and label for training. This will save time from loading the input image and its corresponding label separately.

```
# Load TFRecord
tf_record_filename_queue = tf.train.string_input_producer(
    tf.train.match_filenames_once("./output/training-image.tfrecord"))

# Notice the different record reader, this one is designed to work with TFRe-
cord files which may
# have more than one example in them.
```

```
tf_record_reader = tf.TFRecordReader()
_, tf_record_serialized = tf_record_reader.read(tf_record_filename_queue)

# The label and image are stored as bytes but could be stored as int64 or
float64 values in a
# serialized tf.Example protobuf.
tf_record_features = tf.parse_single_example(
    tf_record_serialized,
    features={
        'label': tf.FixedLenFeature([], tf.string),
        'image': tf.FixedLenFeature([], tf.string),
    })

# Using tf.uint8 because all of the channel information is between 0-255
tf_record_image = tf.decode_raw(
    tf_record_features['image'], tf.uint8)

# Reshape the image to look like the image saved, not required
tf_record_image = tf.reshape(
    tf_record_image,
    [image_height, image_width, image_channels])
# Use real values for the height, width and channels of the image because
it's required
# to reshape the input.

tf_record_label = tf.cast(tf_record_features['label'], tf.string)
```

At first, the file is loaded in the same way as any other file. The main difference is that the file is then read using a `TFRecordReader`. Instead of decoding the image, the TFRecord is parsed `tf.parse_single_example` and then the image is read as raw bytes (`tf.decode_raw`).

After the file is loaded, it is reshaped (`tf.reshape`) in order to keep it in the same layout as `tf.nn.conv2d` expects it `[image_height, image_width, image_channels]`. It'd be save to expand the dimensions (`tf.expand`) in order to add in the `batch_size` dimension to the `input_batch`.

In this case a single image is in the TFRecord but these record files support multiple examples being written to them. It'd be safe to have a single TFRecord file which stores an entire training set but splitting up the files doesn't hurt.

The following code is useful to check that the image saved to disk is the same as the image which was loaded from TensorFlow.

```
sess.run(tf.equal(image, tf_record_image))
```

The output from executing the example code is:

```
 array([[[ True, True, True],
    [ True, True, True],
    [ True, True, True]],
    [[ True, True, True],
    [ True, True, True],
    [ True, True, True]],
    [[ True, True, True],
    [ True, True, True],
    [ True, True, True]]], dtype=bool)
```

All of the attributes of the original image and the image loaded from the TFRecord file are the same. To be sure, load the label from the TFRecord file and check that it is the same as the one saved earlier.

```
# Check that the label is still 0b00000001.
sess.run(tf_record_label)
```

The output from executing the example code is:

```
 b'\x01'
```

Creating a file that stores both the raw image data and the expected output label will save complexities during training. It's not required to use TFRecord files but it's highly recommend when working with images. If it doesn't work well for a workflow, it's still recommended to preprocess images and save them before training. Manipulating an image each time it's loaded is not recommended.

Image Manipulation

CNNs work well when they're given a large amount of diverse quality training data. Images capture complex scenes in a way which visually communicates an intended subject. In the Stanford Dog's Dataset, it's important that the images visually highlight the importance of dogs in the

picture. A picture with a dog clearly visible in the center is considered more valuable than one with a dog in the background.

Not all datasets have the most valuable images. The following are two images from the **Stanford Dogs Dataset**, which are supposed to highlight dog breeds. The image on the left `n02113978_3480.jpg` highlights important attributes of a typical Mexican Hairless Dog, while the image on the right `n02113978_1030.jpg` highlights the look of inebriated party goers scaring a Mexican Hairless Dog. The image on the right `n02113978_1030.jpg` is filled with irrelevant information which may train a CNN to categorize party goer faces instead of Mexican Hairless Dog breeds. Images like this may still include an image of a dog and could be manipulated to highlight the dog instead of people.

n02113978_3480.jpg n02113978_1030.jpg

Image manipulation is best done as a preprocessing step in most scenarios. An image can be cropped, resized and the color levels adjusted. On the other hand, there is an important use case for manipulating an image while training. After an image is loaded, it can be flipped or distorted to diversify the input training information used with the network. This step adds further processing time but helps with overfitting.

TensorFlow is not designed as an image manipulation framework. There are libraries available in Python which support more image manipulation than TensorFlow (**PIL** and **OpenCV**). For TensorFlow, we'll summarize a few useful image manipulation features available which are useful in training CNNs.

CROPPING

Cropping an image will remove certain regions of the image without keeping any information. Cropping is similar to `tf.slice` where a section of a tensor is cut out from the full tensor. Cropping an input image for a CNN can be useful if there is extra input along a dimension which isn't required. For example, cropping dog pictures where the dog is in the center of the images to reduce the size of the input.

```
sess.run(tf.image.central_crop(image, 0.1))
```

The output from executing the example code is:

```
 array([[[ 3, 108, 233]]], dtype=uint8)
```

The example code uses `tf.image.central_crop` to crop out 10% of the image and return it. This method always returns based on the center of the image being used.

Cropping is usually done in preprocessing but it can be useful when training if the background is useful. When the background is useful then cropping can be done while randomizing the center offset of where the crop begins.

```
# This crop method only works on real value input.
real_image = sess.run(image)

bounding_crop = tf.image.crop_to_bounding_box(
    real_image, offset_height=0, offset_width=0, target_height=2, tar-
get_width=1)

sess.run(bounding_crop)
```

The output from executing the example code is:

```
 array([[[ 0, 0, 0]],
    [[ 0, 191, 0]]], dtype=uint8)
```

The example code uses `tf.image.crop_to_bounding_box` in order to crop the image starting at the upper left pixel located at `(0, 0)`. Currently, the function only works with a tensor which has a defined shape so an input image needs to be executed on the graph first.

PADDING

Pad an image with zeros in order to make it the same size as an expected image. This can be accomplished using `tf.pad` but TensorFlow has another function useful for resizing images which are too large or too small. The method will pad an image which is too small including zeros along the edges of the image. Often, this method is used to resize small images because any other method of resizing with distort the image.

```
# This padding method only works on real value input.
```

```
real_image = sess.run(image)

pad = tf.image.pad_to_bounding_box(
    real_image, offset_height=0, offset_width=0, target_height=4, tar-
get_width=4)

sess.run(pad)
```

The output from executing the example code is:

```
 array([[[ 0,  0,  0],
    [255, 255, 255],
    [254,  0,  0],
    [ 0,  0,  0]],
    [[ 0, 191,  0],
    [ 3, 108, 233],
    [ 0, 191,  0],
    [ 0,  0,  0]],
    [[254,  0,  0],
    [255, 255, 255],
    [ 0,  0,  0],
    [ 0,  0,  0]],
    [[ 0,  0,  0],
    [ 0,  0,  0],
    [ 0,  0,  0],
    [ 0,  0,  0]]], dtype=uint8)
```

This example code increases the images height by one pixel and its width by a pixel as well. The new pixels are all set to 0. Padding in this manner is useful for scaling up an image which is too small. This can happen if there are images in the training set with a mix of aspect ratios. TensorFlow has a useful shortcut for resizing images which don't match the same aspect ratio using a combination of `pad` and `crop`.

```
# This padding method only works on real value input.
real_image = sess.run(image)

crop_or_pad = tf.image.resize_image_with_crop_or_pad(
    real_image, target_height=2, target_width=5)

sess.run(crop_or_pad)
```

The output from executing the example code is:

```
 array([[[ 0,  0,  0],
    [ 0,  0,  0],
    [255, 255, 255],
    [254,  0,  0],
    [ 0,  0,  0]],
    [[ 0,  0,  0],
    [ 0, 191,  0],
    [ 3, 108, 233],
    [ 0, 191,  0],
    [ 0,  0,  0]]], dtype=uint8)
```

The `real_image` has been reduced in height to be 2 pixels tall and the width has been increased by padding the image with zeros. This function works based on the center of the image input.

FLIPPING

Flipping an image is exactly what it sounds like. Each pixel's location is reversed horizontally or vertically. Technically speaking, flopping is the term used when flipping an image vertically. Terms aside, flipping images is useful with TensorFlow to give different perspectives of the same image for training. For example, a picture of an Australian Shepherd with crooked left ear could be flipped in order to allow matching of crooked right ears.

TensorFlow has functions to flip images vertically, horizontally and choose randomly. The ability to randomly flip an image is a useful method to keep from overfitting a model to flipped versions of images.

```
top_left_pixels = tf.slice(image, [0, 0, 0], [2, 2, 3])

flip_horizon = tf.image.flip_left_right(top_left_pixels)
flip_vertical = tf.image.flip_up_down(flip_horizon)

sess.run([top_left_pixels, flip_vertical])
```

The output from executing the example code is:

```
 [array([[[ 0,  0,  0],
    [255, 255, 255]],
    [[ 0, 191,  0],
    [ 3, 108, 233]]], dtype=uint8), array([[[ 3, 108, 233],
    [ 0, 191,  0]],
    [[255, 255, 255],
    [ 0,  0,  0]]], dtype=uint8)]
```

This example code flips a subset of the image horizontally and then vertically. The subset is used with `tf.slice` because the original image flipped returns the same images (for this example only). The subset of pixels illustrates the change which occurs when an image is flipped. `tf.image.flip_left_right` and `tf.image.flip_up_down` both operate on tensors which are not limited to images. These will flip an image a single time, randomly flipping an image is done using a separate set of functions.

```
top_left_pixels = tf.slice(image, [0, 0, 0], [2, 2, 3])

random_flip_horizon = tf.image.random_flip_left_right(top_left_pixels)
random_flip_vertical = tf.image.random_flip_up_down(random_flip_horizon)

sess.run(random_flip_vertical)
```

The output from executing the example code is:

```
 array([[[ 3, 108, 233],
    [ 0, 191,  0]],
    [[255, 255, 255],
    [ 0,  0,  0]]], dtype=uint8)
```

This example does the same logic as the example before except that the output is random. Every time this runs, a different output is expected. There is a parameter named `seed` which may be used to control how random the flipping occurs.

SATURATION AND BALANCE

Images which are found on the internet are often edited in advance. For instance, many of the images found in the Stanford Dogs dataset have too much saturation (lots of color). When an edited image is used for training, it may mislead a CNN model into finding patterns which are related to the edited image and not the content in the image.

TensorFlow has useful functions which help in training on images by changing the saturation, hue, contrast and brightness. The functions allow for simple manipulation of these image attributes as well as randomly altering these attributes. The random altering is useful in training in for the same reason randomly flipping an image is useful. The random attribute changes help a CNN be able to accurately match a feature in images which have been edited or were taken under different lighting.

```
example_red_pixel = tf.constant([254., 2., 15.])
adjust_brightness = tf.image.adjust_brightness(example_red_pixel, 0.2)

sess.run(adjust_brightness)
```

The output from executing the example code is:

```
 array([ 254.19999695,  2.20000005,  15.19999981], dtype=float32)
```

This example brightens a single pixel, which is primarily red, with a delta of `0.2`. Unfortunately, in the current version of TensorFlow 0.9, this method doesn't work well with a `tf.uint8` input. It's best to avoid using this when possible and preprocess brightness changes.

```
adjust_contrast = tf.image.adjust_contrast(image, -.5)

sess.run(tf.slice(adjust_contrast, [1, 0, 0], [1, 3, 3]))
```

The output from executing the example code is:

```
 array([[[170,  71, 124],
    [168, 112,  7],
    [170,  71, 124]]], dtype=uint8)
```

The example code changes the contrast by `-0.5` which makes the new version of the image fairly unrecognizable. Adjusting contrast is best done in small increments to keep from blowing out an image. Blowing out an image means the same thing as saturating a neuron, it reached its maximum value and can't be recovered. With contrast changes, an image can become completely white and completely black from the same adjustment.

The `tf.slice` operation is for brevity, highlighting one of the pixels which has changed. It is not required when running this operation.

```
adjust_hue = tf.image.adjust_hue(image, 0.7)

sess.run(tf.slice(adjust_hue, [1, 0, 0], [1, 3, 3]))
```

The output from executing the example code is:

```
 array([[[191,  38,  0],
```

```
    [ 62, 233, 3],
    [191, 38, 0]]], dtype=uint8)
```

The example code adjusts the hue found in the image to make it more colorful. The adjustment accepts a `delta` parameter which controls the amount of hue to adjust in the image.

```
adjust_saturation = tf.image.adjust_saturation(image, 0.4)

sess.run(tf.slice(adjust_saturation, [1, 0, 0], [1, 3, 3]))
```

The output from executing the example code is:

```
 array([[[114, 191, 114],
    [141, 183, 233],
    [114, 191, 114]]], dtype=uint8)
```

The code is similar to adjusting the contrast. It is common to oversaturate an image in order to identify edges because the increased saturation highlights changes in colors.

Colors

CNNs are commonly trained using images with a single color. When an image has a single color it is said to use a grayscale colorspace meaning it uses a single channel of colors. For most computer vision related tasks, using grayscale is reasonable because the shape of an image can be seen without all the colors. The reduction in colors equates to a quicker to train image. Instead of a 3 component rank 1 tensor to describe each color found with RGB, a grayscale image requires a single component rank 1 tensor to describe the amount of gray found in the image.

Although grayscale has benefits, it's important to consider applications which require a distinction based on color. Color in images is challenging to work with in most computer vision because it isn't easy to mathematically define the similarity of two RGB colors. In order to use colors in CNN training, it's useful to convert the colorspace the image is natively in.

GRAYSCALE

Grayscale has a single component to it and has the same range of color as RGB [0,255].

```
gray = tf.image.rgb_to_grayscale(image)

sess.run(tf.slice(gray, [0, 0, 0], [1, 3, 1]))
```

The output from executing the example code is:

```
 array([[[ 0],
    [255],
    [ 76]]], dtype=uint8)
```

This example converted the RGB image into grayscale. The `tf.slice` operation took the top row of pixels out to investigate how their color has changed. The grayscale conversion is done by averaging all the color values for a pixel and setting the amount of grayscale to be the average.

HSV

Hue, saturation and value are what make up HSV colorspace. This space is represented with a 3 component rank 1 tensor similar to RGB. HSV is not similar to RGB in what it measures, it's measuring attributes of an image which are closer to human perception of color than RGB. It is sometimes called HSB, where the B stands for brightness.

```
hsv = tf.image.rgb_to_hsv(tf.image.convert_image_dtype(image, tf.float32))

sess.run(tf.slice(hsv, [0, 0, 0], [3, 3, 3]))
```

The output from executing the example code is:

```
 array([[[ 0., 0., 0.],
    [ 0., 0., 1.],
    [ 0., 1., 0.99607849]],
    [[ 0.33333334, 1., 0.74901962],
    [ 0.59057975, 0.98712444, 0.91372555],
    [ 0.33333334, 1., 0.74901962]],
    [[ 0., 1., 0.99607849],
    [ 0., 0., 1.],
    [ 0., 0., 0.]]], dtype=float32)
```

RGB

RGB is the colorspace which has been used in all the example code so far. It's broken up into a 3 component rank 1 tensor which includes the amount of red [0,255], green [0,255] and blue [0,255]. Most images are already in RGB but TensorFlow has builtin functions in case the

images are in another colorspace.

```
rgb_hsv = tf.image.hsv_to_rgb(hsv)
rgb_grayscale = tf.image.grayscale_to_rgb(gray)
```

The example code is straightforward except that the conversion from grayscale to RGB doesn't make much sense. RGB expects three colors while grayscale only has one. When the conversion occurs, the RGB values are filled with the same value which is found in the grayscale pixel.

LAB

Lab is not a colorspace which TensorFlow has native support for. It's a useful colorspace because it can map to a larger number of perceivable colors than RGB. Although TensorFlow doesn't support this natively, it is a colorspace, which is often used in professional settings. Another Python library **python-colormath** has support for Lab conversion as well as other colorspaces not described here.

The largest benefit using a Lab colorspace is it maps closer to humans perception of the difference in colors than RGB or HSV. The euclidean distance between two colors in a Lab colorspace are somewhat representative of how different the colors look to a human.

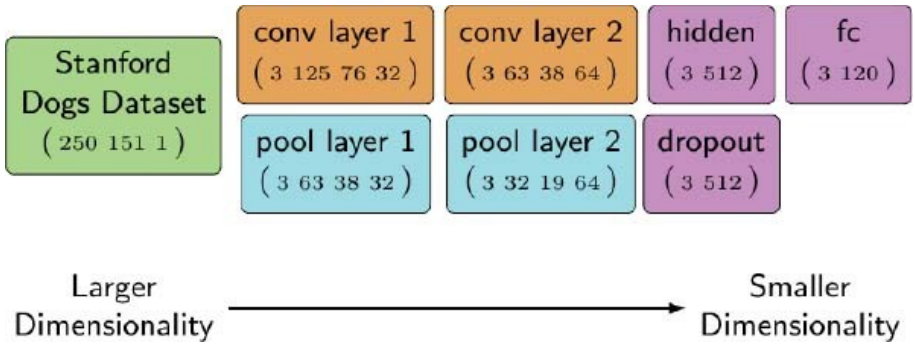
CASTING IMAGES

In these examples, `tf.to_float` is often used in order to illustrate changing an image's type to another format. For examples, this works OK but TensorFlow has a built in function to properly scale values as they change types. `tf.image.convert_image_dtype(image, dtype, saturate=False)` is a useful shortcut to change the type of an image from `tf.uint8` to `tf.float`.

CNN Implementation

Object recognition and categorization using TensorFlow required a basic understanding of convolutions (for CNNs), common layers (non-linearity, pooling, fc), image loading, image manipulation and colorspaces. With these areas covered, it's possible to build a CNN model for image recognition and classification using TensorFlow. In this case, the model is a dataset provided by Stanford which includes pictures of dogs and their corresponding breed. The network needs to train on these pictures then be judged on how well it can guess a dog's breed based on a picture.

The network architecture follows a simplified version of **Alex Krizhevsky's AlexNet** without all of AlexNet's layers. This architecture was described earlier in the chapter as the network which won ILSVRC'12 top challenge. The network uses layers and techniques familiar to this chapter which are similar to the **TensorFlow provided** tutorial on CNNs.



The network described in this section including the output TensorShape after each layer. The layers are read from left to right and top to bottom where related layers are grouped together. As the input progresses further into the network, its height and width are reduced while its depth is increased. The increase in depth reduces the computation required to use the network.

Stanford Dogs Dataset

The dataset used for training this model can be found on Stanford's computer vision site http://vision.stanford.edu/aditya86/ImageNetDogs/. Training the model requires downloading relevant data. After downloading the Zip archive of all the images, extract the archive into a new directory called `imagenet-dogs` in the same directory as the code building the model.

The Zip archive provided by Stanford includes pictures of dogs organized into 120 different breeds. The goal of this model is to train on 80% of the dog breed images and then test using the remaining 20%. If this were a production model, part of the raw data would be reserved for cross-validation of the results. Cross-validation is a useful step to validate the accuracy of a model but this model is designed to illustrate the process and not for competition.

The organization of the archive follows ImageNet's practices. Each dog breed is a directory name similar to `n02085620-Chihuahua` where the second half of the directory name is the dog's breed in English (`Chihuahua`). Within each directory there is a variable amount of images related to that breed. Each image is in JPEG format (RGB) and of varying sizes. The different sized images is a challenge because TensorFlow is expecting tensors of the same dimensionality.

Convert Images to TFRecords

The raw images organized in a directory doesn't work well for training because the images are not of the same size and their dog breed isn't included in the file. Converting the images into TFRecord files in advance of training will help keep training fast and simplify matching the label of the image. Another benefit is that the training and testing related images can be separated in advance. Separated training and testing datasets allows continual testing of a model while training is occurring using checkpoint files.

Converting the images will require changing their colorspace into grayscale, resizing the images to be of uniform size and attaching the label to each image. This conversion should only happen once before training commences and likely will take a long time.

```
import glob

image_filenames = glob.glob("./imagenet-dogs/n02*/*.jpg")

image_filenames[0:2]
```

The output from executing the example code is:

```
 ['./imagenet-dogs/n02085620-Chihuahua/n02085620_10074.jpg',
    './imagenet-dogs/n02085620-Chihuahua/n02085620_10131.jpg']
```

An example of how the archive is organized. The `glob` module allows directory listing which shows the structure of the files which exist in the dataset. The eight digit number is tied to the **WordNet ID** of each category used in ImageNet. ImageNet has a browser for image details which accepts the WordNet ID, for example the Chihuahua example can be accessed via http://www.image-net.org/synset?wnid=n02085620.

```
from itertools import groupby
from collections import defaultdict

training_dataset = defaultdict(list)
testing_dataset = defaultdict(list)

# Split up the filename into its breed and corresponding filename. The breed
is found by taking the directory name
image_filename_with_breed = map(lambda filename: (filename.split("/")[2],
filename), image_filenames)

# Group each image by the breed which is the 0th element in the tuple re-
turned above
for dog_breed, breed_images in groupby(image_filename_with_breed, lambda x:
x[0]):
    # Enumerate each breed's image and send ~20% of the images to a testing
set
    for i, breed_image in enumerate(breed_images):
        if i % 5 == 0:
            testing_dataset[dog_breed].append(breed_image[1])
        else:
            training_dataset[dog_breed].append(breed_image[1])

    # Check that each breed includes at least 18% of the images for testing
    breed_training_count = len(training_dataset[dog_breed])
    breed_testing_count = len(testing_dataset[dog_breed])

    assert round(breed_testing_count / (breed_training_count + breed_test-
ing_count), 2) > 0.18, "Not enough testing images."
```

This example code organized the directory and images ('./imagenet-dogs/n02085620-Chihuahua/n02085620_10131.jpg') into two dictionaries related to each breed including all the images for that breed. Now each dictionary would include Chihuahua images in the following format:

```
   training_dataset["n02085620-Chihuahua"]
["n02085620_10131.jpg", ...]
```

Organizing the breeds into these dictionaries simplifies the process of selecting each type of image and categorizing it. During preprocessing, all the image breeds can be iterated over and their images opened based on the filenames in the list.

```
def write_records_file(dataset, record_location):
    """
    Fill a TFRecords file with the images found in `dataset` and include
their category.

    Parameters
    ----------
    dataset : dict(list)
      Dictionary with each key being a label for the list of image filenames
```

```
of its value.
    record_location : str
      Location to store the TFRecord output.
    """
    writer = None

    # Enumerating the dataset because the current index is used to breakup
the files if they get over 100
    # images to avoid a slowdown in writing.
    current_index = 0
    for breed, images_filenames in dataset.items():
        for image_filename in images_filenames:
            if current_index % 100 == 0:
                if writer:
                    writer.close()

                record_filename = "{record_location}-
{current_index}.tfrecords".format(
                    record_location=record_location,
                    current_index=current_index)

                writer = tf.python_io.TFRecordWriter(record_filename)
            current_index += 1

            image_file = tf.read_file(image_filename)

            # In ImageNet dogs, there are a few images which TensorFlow
doesn't recognize as JPEGs. This
            # try/catch will ignore those images.
            try:
                image = tf.image.decode_jpeg(image_file)
            except:
                print(image_filename)
                continue

            # Converting to grayscale saves processing and memory but isn't
required.
            grayscale_image = tf.image.rgb_to_grayscale(image)
            resized_image = tf.image.resize_images(grayscale_image, 250, 151)

            # tf.cast is used here because the resized images are floats but
haven't been converted into
            # image floats where an RGB value is between [0,1).
            image_bytes = sess.run(tf.cast(resized_image, tf.uint8)).to-
bytes()

            # Instead of using the label as a string, it'd be more efficient
to turn it into either an
            # integer index or a one-hot encoded rank one tensor.
            # https://en.wikipedia.org/wiki/One-hot
            image_label = breed.encode("utf-8")

            example = tf.train.Example(features=tf.train.Features(feature={
                'label': tf.train.Feature(bytes_list=tf.train.BytesList(val-
ue=[image_label])),
                'image': tf.train.Feature(bytes_list=tf.train.BytesList(val-
ue=[image_bytes]))
            }))

            writer.write(example.SerializeToString())
    writer.close()

write_records_file(testing_dataset, "./output/testing-images/testing-image")
write_records_file(training_dataset, "./output/training-images/training-
image")
```

The example code is opening each image, converting it to grayscale, resizing it and then adding it to a TFRecord file. The logic isn't different from earlier examples except that the operation `tf.image.resize_images` is used. The resizing operation will scale every image to be

the same size even if it distorts the image. For example, if an image in portrait orientation and an image in landscape orientation were both resized with this code then the output of the landscape image would become distorted. These distortions are caused because `tf.image.resize_images` doesn't take into account aspect ratio (the ratio of height to width) of an image. To properly resize a set of images, cropping or padding is a preferred method because it ignores the aspect ratio stopping distortions.

Load Images

Once the testing and training dataset have been transformed to TFRecord format, they can be read as TFRecords instead of as JPEG images. The goal is to load the images a few at a time with their corresponding labels.

```
filename_queue = tf.train.string_input_producer(
    tf.train.match_filenames_once("./output/training-images/*.tfrecords"))
reader = tf.TFRecordReader()
_, serialized = reader.read(filename_queue)

features = tf.parse_single_example(
    serialized,
    features={
        'label': tf.FixedLenFeature([], tf.string),
        'image': tf.FixedLenFeature([], tf.string),
    })

record_image = tf.decode_raw(features['image'], tf.uint8)

# Changing the image into this shape helps train and visualize the output by
converting it to
# be organized like an image.
image = tf.reshape(record_image, [250, 151, 1])

label = tf.cast(features['label'], tf.string)

min_after_dequeue = 10
batch_size = 3
capacity = min_after_dequeue + 3 * batch_size
image_batch, label_batch = tf.train.shuffle_batch(
    [image, label], batch_size=batch_size, capacity=capacity, min_after_de-
queue=min_after_dequeue)
```

This example code loads training images by matching all the TFRecord files found in the training directory. Each TFRecord includes multiple images but the `tf.parse_single_example` will take a single example out of the file. The batching operation discussed earlier is used to train multiple images simultaneously. Batching multiple images is useful because these operations are designed to work with multiple images the same as with a single image. The primary requirement is that the system have enough memory to work with them all.

With the images available in memory, the next step is to create the model used for training and testing.

Model

The model used is similar to the **mnist convolution example** which is often used in tutorials describing convolutional neural networks in TensorFlow. The architecture of this model is simple yet it performs well for illustrating different techniques used in image classification and recognition. An advanced model may borrow more from **Alex Krizhevsky's AlexNet** design that includes more convolution layers.

```
# Converting the images to a float of [0,1) to match the expected input to
convolution2d
float_image_batch = tf.image.convert_image_dtype(image_batch, tf.float32)

conv2d_layer_one = tf.contrib.layers.convolution2d(
    float_image_batch,
    num_output_channels=32,     # The number of filters to generate
    kernel_size=(5,5),          # It's only the filter height and width.
    activation_fn=tf.nn.relu,
    weight_init=tf.random_normal,
    stride=(2, 2),
    trainable=True)
pool_layer_one = tf.nn.max_pool(conv2d_layer_one,
    ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1],
    padding='SAME')
# Note, the first and last dimension of the convolution output hasn't
changed but the
# middle two dimensions have.
```

```
conv2d_layer_one.get_shape(), pool_layer_one.get_shape()
```

The output from executing the example code is:

```
 (TensorShape([Dimension(3), Dimension(125), Dimension(76), Dimension(32)]),
    TensorShape([Dimension(3), Dimension(63), Dimension(38), Dimension(32)]))
```

The first layer in the model is created using the shortcut `tf.contrib.layers.convolution2d`. It's important to note that the `weight_init` is set to be a random normal, meaning that the first set of filters are filled with random numbers following a normal distribution (this parameter is renamed in TensorFlow 0.9 to be `weights_initializer`). The filters are set as `trainable` so that as the network is fed information, these weights are adjusted to improve the accuracy of the model.

After a convolution is applied to the images, the output is downsized using a `max_pool` operation. After the operation, the output shape of the convolution is reduced in half due to the `ksize` used in the pooling and the `strides`. The reduction didn't change the number of filters (output channels) or the size of the image batch. The components that were reduced dealt with the height and width of the image (filter).

```
conv2d_layer_two = tf.contrib.layers.convolution2d(
    pool_layer_one,
    num_output_channels=64,        # More output channels means an increase
in the number of filters
    kernel_size=(5,5),
    activation_fn=tf.nn.relu,
    weight_init=tf.random_normal,
    stride=(1, 1),
    trainable=True)

pool_layer_two = tf.nn.max_pool(conv2d_layer_two,
    ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1],
    padding='SAME')

conv2d_layer_two.get_shape(), pool_layer_two.get_shape()
```

The output from executing the example code is:

```
 (TensorShape([Dimension(3), Dimension(63), Dimension(38), Dimension(64)]),
    TensorShape([Dimension(3), Dimension(32), Dimension(19), Dimension(64)]))
```

The second layer changes little from the first except the depth of the filters. The number of filters is now doubled while again reducing the size of the height and width of the image. The multiple convolution and pool layers are continuing to reduce the height and width of the input while adding further depth.

At this point, further convolution and pool steps could be taken. In many architectures there are over 5 different convolution and pooling layers. The most advanced architectures take longer to train and debug but they can match more sophisticated patterns. In this example, the two convolution and pooling layers are enough to illustrate the mechanics at work.

The tensor being operated on is still fairly complex tensor, the next step is to fully connect every point in each image with an output neuron. Since this example is using `softmax` later, the fully connected layer needs to be changed into a rank two tensor. The tensor's first dimension will be used to separate each image while the second dimension is a rank one tensor of each input tensor.

```
flattened_layer_two = tf.reshape(
    pool_layer_two,
    [
        batch_size,  # Each image in the image_batch
        -1           # Every other dimension of the input
    ])

flattened_layer_two.get_shape()
```

The output from executing the example code is:

```
 TensorShape([Dimension(3), Dimension(38912)])
```

`tf.reshape` has a special value that can be used to signify, use all the dimensions remaining. In this example code, the `-1` is used to reshape the last pooling layer into a giant rank one tensor. With the pooling layer flattened out, it can be combined with two fully connected layers which associate the current network state to the breed of dog predicted.

```
# The weight_init parameter can also accept a callable, a lambda is used
here  returning a truncated normal
# with a stddev specified.
hidden_layer_three = tf.contrib.layers.fully_connected(
    flattened_layer_two,
    512,
    weight_init=lambda i, dtype: tf.truncated_normal([38912, 512],
```

```
stddev=0.1),
    activation_fn=tf.nn.relu
)

# Dropout some of the neurons, reducing their importance in the model
hidden_layer_three = tf.nn.dropout(hidden_layer_three, 0.1)

# The output of this are all the connections between the previous layers and
the 120 different dog breeds
# available to train on.
final_fully_connected = tf.contrib.layers.fully_connected(
    hidden_layer_three,
    120,  # Number of dog breeds in the ImageNet Dogs dataset
    weight_init=lambda i, dtype: tf.truncated_normal([512, 120], stddev=0.1)
)
```

This example code creates the final fully connected layer of the network where every pixel is associated with every breed of dog. Every step of this network has been reducing the size of the input images by converting them into filters which are then matched with a breed of dog (label). This technique has reduced the processing power required to train or test a network while generalizing the output.

Training

Once a model is ready to be trained, the last steps follow the same process discussed in earlier chapters of this book. The model's loss is computed based on how accurately it guessed the correct labels in the training data which feeds into a training optimizer which updates the weights of each layer. This process continues one iteration at a time while attempting to increase the accuracy of each step.

An important note related to this model, during training most classification functions (`tf.nn.softmax`) require numerical labels. This was highlighted in the section describing loading the images from TFRecords. At this point, each label is a string similar to `n02085620-Chihuahua`. Instead of using `tf.nn.softmax` on this string, the label needs to be converted to be a unique number for each label. Converting these labels into an integer representation should be done in preprocessing.

For this dataset, each label will be converted into an integer which represents the index of each name in a list including all the dog breeds. There are many ways to accomplish this task, for this example a new TensorFlow utility operation will be used (`tf.map_fn`).

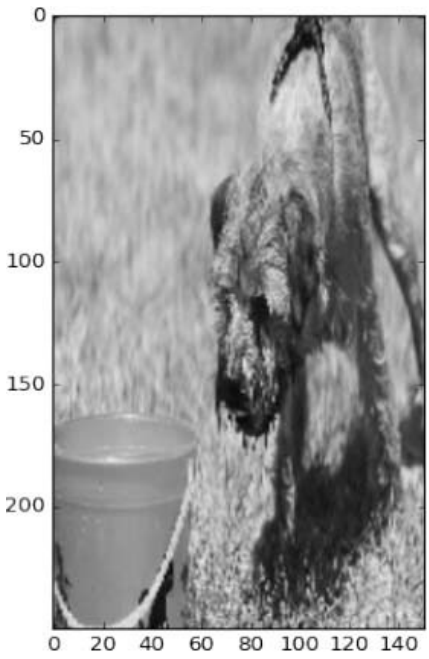
```
import glob

# Find every directory name in the imagenet-dogs directory (n02085620-
Chihuahua, ...)
labels = list(map(lambda c: c.split("/")[-1], glob.glob("./imagenet-dogs/
*")))

# Match every label from label_batch and return the index where they exist
in the list of classes
train_labels = tf.map_fn(lambda l: tf.where(tf.equal(labels, l))[0,0:1][0],
label_batch, dtype=tf.int64)
```

This example code uses two different forms of a `map` operation. The first form of `map` is used to create a list including only the dog breed name based on a list of directories. The second form of `map` is `tf.map_fn` which is a TensorFlow operation that will map a function over a tensor on the graph. The `tf.map_fn` is used to generate a rank one tensor including only the integer indexes where each label is located in the list of all the class labels. These unique integers can now be used with `tf.nn.softmax` to classify output predictions.

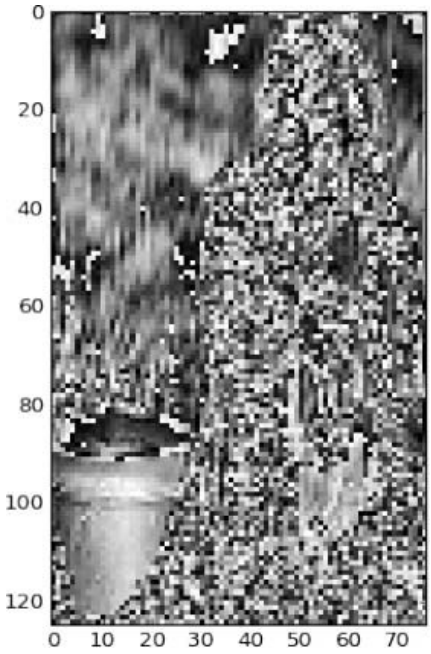
Debug the Filters with Tensorboard

CNNs have multiple moving parts which can cause issues during training resulting in poor accuracy. Debugging problems in a CNN often start with investigating how the filters (kernels) are changing every iteration. Each weight used in a filter is constantly changing as the network attempts to learn the most accurate set of weights to use based on the train method.

In a well designed CNN, when the first convolution layer is started, the initialized input weights are set to be random (in this case using `weight_init=tf.random_normal`). These weights activate over an image and the output of the activation (feature map) is random as well. Visualizing the feature map as if it were an image, the output looks like the original image with static applied. The static is caused by all the weights activating at random. Over many iterations, each filter becomes more uniform as the weights are adjusted to fit the training feedback. As the network converges, the filters resemble distinct small patterns which can be found in the image. Here is an original grayscale training image before it is passed through the first convolution layer:



And, here is a single feature map from the first convolution layer highlighting randomness in the output:



Debugging a CNN requires a familiarity working with these filters. Currently there isn't any built in support in tensorboard to display filters or feature maps. A simple view of the filters can be done using a `tf.image_summary` operation on the filters being trained and the feature maps generated. Adding an image summary output to a graph gives a good overview of the filters being used and the feature map generated by applying them to the input images.

The Jupyter notebook extension worth mentioning is **TensorDebugger**, which is in an early state of development. The extension has a mode capable of viewing changes in filters as an animated Gif over iterations.

Conclusion

Convolutional Neural Networks are a useful network architecture that are implemented with a minimal amount of code in TensorFlow. While they're designed with images in mind, a CNN is not limited to image input. Convolutions are used in multiple industries from music to medical and a CNN can be applied in a similar manner. Currently, TensorFlow is designed for two dimensional convolutions but it's still possible to work with higher dimensionality input using TensorFlow.

While a CNN could theoretically work with natural language data (text), it isn't designed for this type of input. Text input is often stored in a `SparseTensor` where the majority of the input is 0. CNNs are designed to work with dense input where each value is important and the

majority of the input is not 0. Working with text data is a challenge which is addressed in the next chapter on "Recurrent Neural Networks and Natural Language Processing".