UC Berkeley Extension

Machine Learning With TensorFlow

X433.7-001 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

Course Content Outline

- **Machne Learning**
- Linear and Logistic Regression
- Softmax classification
- Multi-layer Neuaral Network
- Gradient descent and Backpropagation

- **Neural Networks**
- Object recognition with Convolutional Neural Network (CNN)
- Activation Functions
- Common layers: Conv and Pooling Layers
- CNN Overview — Midterm / Project proposal due (30pts)

- **Working with images**
- Normlization
- Loading Images
- Image formats and manipulation

- **CNN Implementation**
- Training
- Recurrent Neural Network (RNN)
- Project Presentations 1/2

- **Project Presentations**
- Project Presentation 2/2 — Final Project (40pts)

Input and Kernel

- This example code creates two tensors:

```
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]]
        ]
    ])
```

Input and Kernel

- Here is a single kernel which is the first dimension of the kernel variable:

```
conv2d = tf.nn.conv2d(input_batch, kernel, strides=[1, 1, 1,
1], padding='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 convolution of these two tensors creates a feature map

Input and Kernel

- The relationship between the input images and the output feature map can be summarized like this:
  - 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

- **Example**:

here, 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] (see previous slides)

The values correspond to: [pixel value * corresponding kernel value]

Input and Kernel

- The code:

```
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

Strides

- The value of convolutions in computer vision is their ability to reduce the dimensionality of the input
- An image's dimensionality (2D image) is its width, height and number of channels (for ex. R,G,B,(A))
- 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

Strides

- The parameter strides, causes a kernel to skip over pixels of an image and not include them in 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
- Instead of going over every element of an input, the strides parameter could configure the convolution to skip certain elements

Strides

```
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], padding='SAME')
sess.run(conv2d)
```

Strides

- The output from executing the example code is:

```
array([[[[ 2.20000005],
   [ 8.19999981]],
   [[ 2.79999995],
   [ 8.80000019]]]], dtype=float32)
```

- Steps:
  - 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.

Strides

input_batch f

$$\begin{matrix} 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\ 0.1 & 1.1 & 2.1 & 3.1 & 4.1 & 5.1 \\ 0.2 & 1.2 & 2.2 & 3.2 & 4.2 & 5.2 \\ 0.3 & 1.3 & 2.3 & 3.3 & 4.3 & 5.3 \\ 0.4 & 1.4 & 2.4 & 3.4 & 4.4 & 5.4 \\ 0.5 & 1.5 & 2.5 & 3.5 & 4.5 & 5.5 \end{matrix}$$

kernel g

$$\begin{bmatrix} 0 & 0.5 & 0 \\ 0 & 1 & 0 \\ 0 & 0.5 & 0 \end{bmatrix}$$

$$(f_0 * g_0 + \cdots + f_n * g_n)$$

$$\begin{pmatrix} 0.0 * 0 + 1.0 * 0.5 + 2.0 * 0 \\ 0.1 * 0 + 1.1 * 1 + 2.1 * 0 \\ 0.2 * 0 + 1.2 * 0.5 + 2.2 * 0 \end{pmatrix}$$

output

$$\begin{bmatrix} 2.2 & 8.2 \\ 2.8 & 8.8 \end{bmatrix}$$

Strides

- 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]

Strides

- 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

- Filling the missing area of the image is known as padding
- 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’), where:
  - **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 may be avoided

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

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

Kernels in Depth

- **Example**: edge detection in images

- Edge detection kernels are common in computer vision applications and could be implemented using basic TensorFlow operations and a single tf.nn.conv2d operation

Kernels in Depth

- **Example**: edge detection in images

```
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], pad-
ding="SAME")
activation_map = sess.run(tf.minimum(tf.nn.relu(conv2d), 255))
```

Kernels in Depth

- **Example**: edge detection in images

Kernels in Depth

- **Example**: sharpening an image

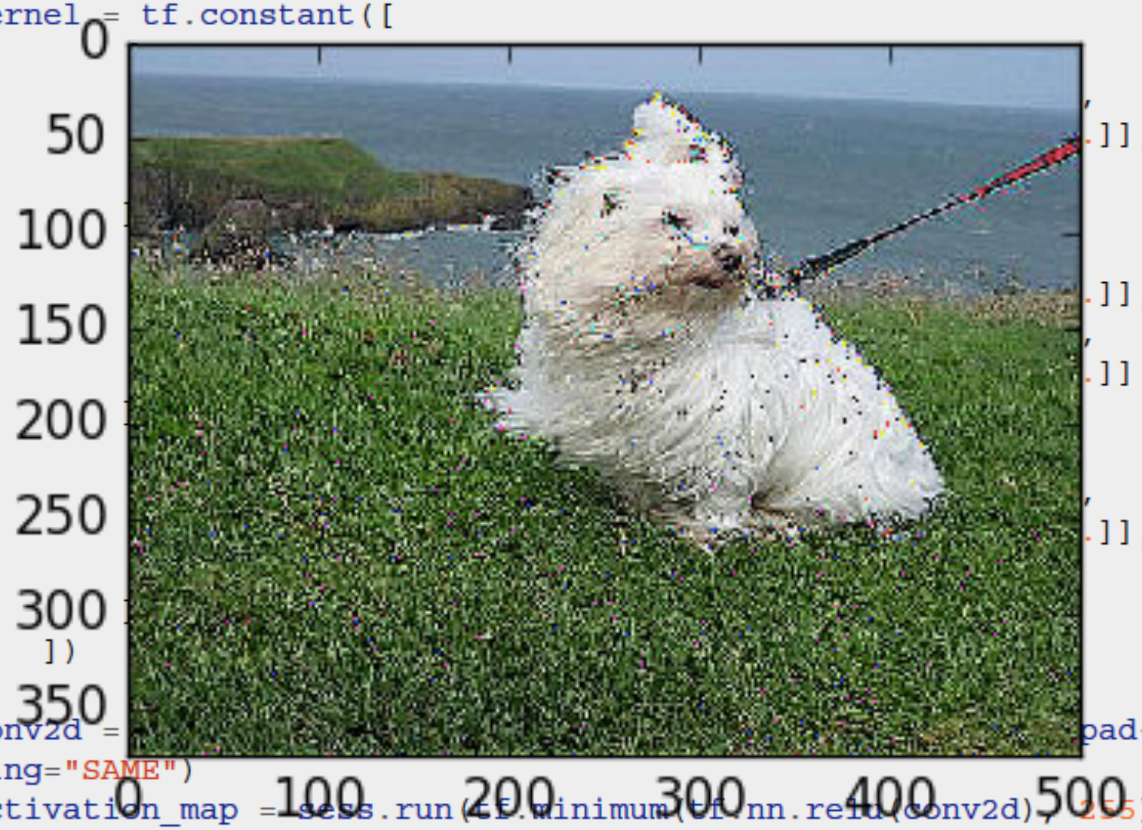
```
kernel = tf.constant([
        [
            [[ 0., 0., 0.], [ 0., 0., 0.], [ 0., 0., 0.]],
            [[ -1., 0., 0.], [ 0., -1., 0.], [ 0., 0., -1.]],
            [[ 0., 0., 0.], [ 0., 0., 0.], [ 0., 0., 0.]]
        ],
        [
            [[ -1., 0., 0.], [ 0., -1., 0.], [ 0., 0., -1.]],
            [[ 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], pad-
ding="SAME")
activation_map = sess.run(tf.minimum(tf.nn.relu(conv2d), 255))
```

Kernels in Depth

- **Example**: sharpening an image

```
kernel = tf.constant([
                                                                  .]],
                                                                  .]],
                                                                  .]]
                                                                  .]],
  ])
conv2d =                                                          pad-
ding="SAME")
activation_map = sess.run(tf.minimum(tf.nn.relu(conv2d), 255))
```



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

Common Layers

- These layers help:
  - reduce overfitting
  - speed up training and
  - decrease memory usage
- The layers covered here 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:

  – 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.

Convolution Layers

- **TF.NN.DEPTHWISE_CONV2D (1.x-2.x)**
- 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

Convolution Layers

- **TF.NN.SEPARABLE_CONV2D (1.x-2.x)**
- 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

Convolution Layers

- **TF.NN.CONV2D_TRANSPOSE (1.x-2.x)**
- 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

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, which means that the input is a curve instead of a straight line
- Curves can represent more complex changes in input

Activation Functions

- TensorFlow has multiple activation functions available
- With CNNs, tf.nn.relu is primarily used because of its performance
- When starting out, using tf.nn.relu is recommended, but advanced users may create their own

Activation Functions

- TensorFlow has multiple activation functions available
- With CNNs, tf.nn.relu is primarily used because of its performance
- ```
In [2]: tf.nn.relu?
Signature: tf.nn.relu(features, name=None)
Docstring:
Computes rectified linear: `max(features, 0)`.

Args:
  features: A `Tensor`. Must be one of the following types: `float32`, `float64`,
`int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`, `uint32`, `uint64`,
`bfloat16`.
  name: A name for the operation (optional).

Returns:
  A `Tensor`. Has the same type as `features`.
```
```

Activation Functions

- 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.

2. The function is differentiable, so there must be a derivative at any point in the function's domain.

Activation Functions

- **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

Activation Functions

- **TF.NN.RELU**

```
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], so the output will be [0, 3]

Activation Functions

- **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 the input become exaggerated

Activation Functions

- **TF.SIGMOID**

```
# 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

Activation Functions

- **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

Activation Functions

- **TF.TANH**

```
# Note, tf.tanh (tf.nn.tanh) is currently limited to float val-
ues
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

Activation Functions

- **TF.NN.DROPOUT**

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

Activation Functions

- **TF.NN.DROPOUT**

```
features = tf.constant([-0.1, 0.0, 0.1, 0.2])
# Note, the output should be different on almost ever execu-
tion. Your numbers 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 overfitting 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

Pooling Layers

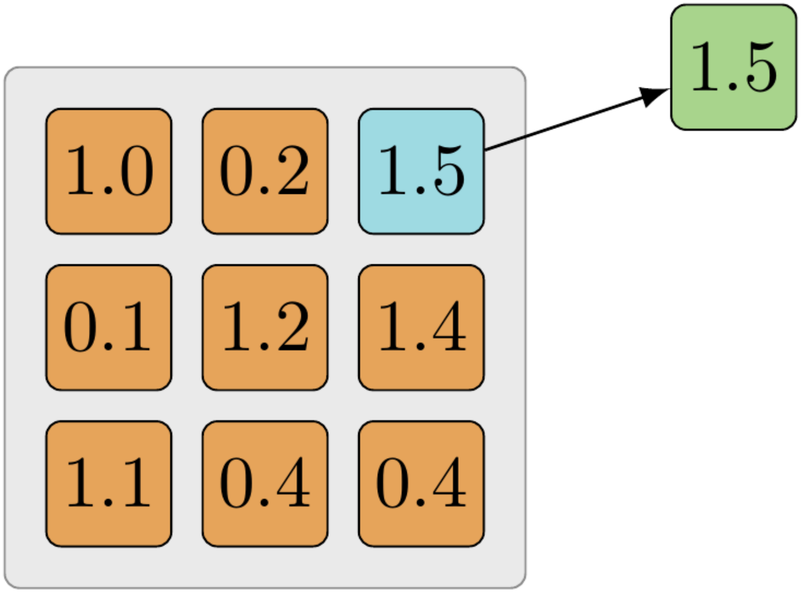
- **TF.NN.MAX_POOL**

- 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

Pooling Layers

- **TF.NN.MAX_POOL**



- The same example is modeled using example code on next slide. The goal is to find the largest value within the tensor

Pooling Layers

```
# 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 im-
age_height and image_width
kernel = [batch_size, input_height, input_width, input_chan-
nels]
max_pool = tf.nn.max_pool(layer_input, kernel, [1, 1, 1, 1],
"VALID")
sess.run(max_pool)
```

Pooling Layers

- **TF.NN.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

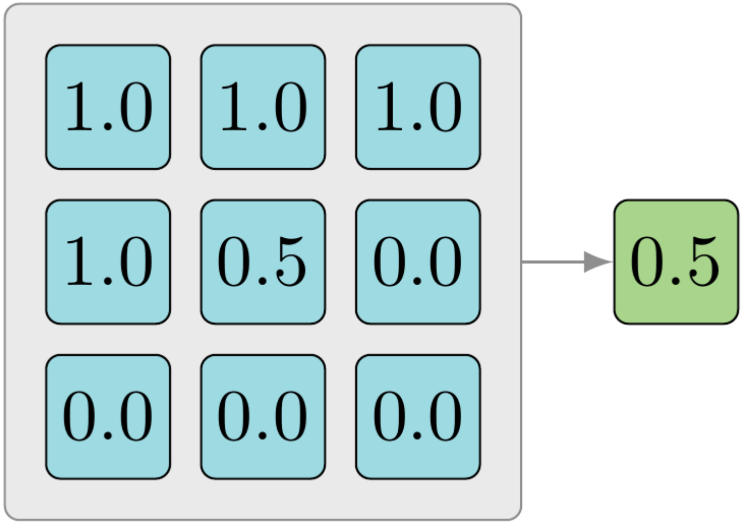
- In this case, the largest value of the tensor is 1.5 and is returned in the same format as the input.

Pooling Layers

- **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
- Example: input tensors with a large width and height but small depth

Pooling Layers

- **TF.NN.AVG_POOL**



- The same example is modeled using example code on next slide. The goal is to find the average of all the values within the tensor

Pooling Layers

```
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 im-
age_height and image_width
kernel = [batch_size, input_height, input_width, input_chan-
nels]
max_pool = tf.nn.avg_pool(layer_input, kernel, [1, 1, 1, 1],
"VALID")
sess.run(max_pool)
```

Pooling Layers

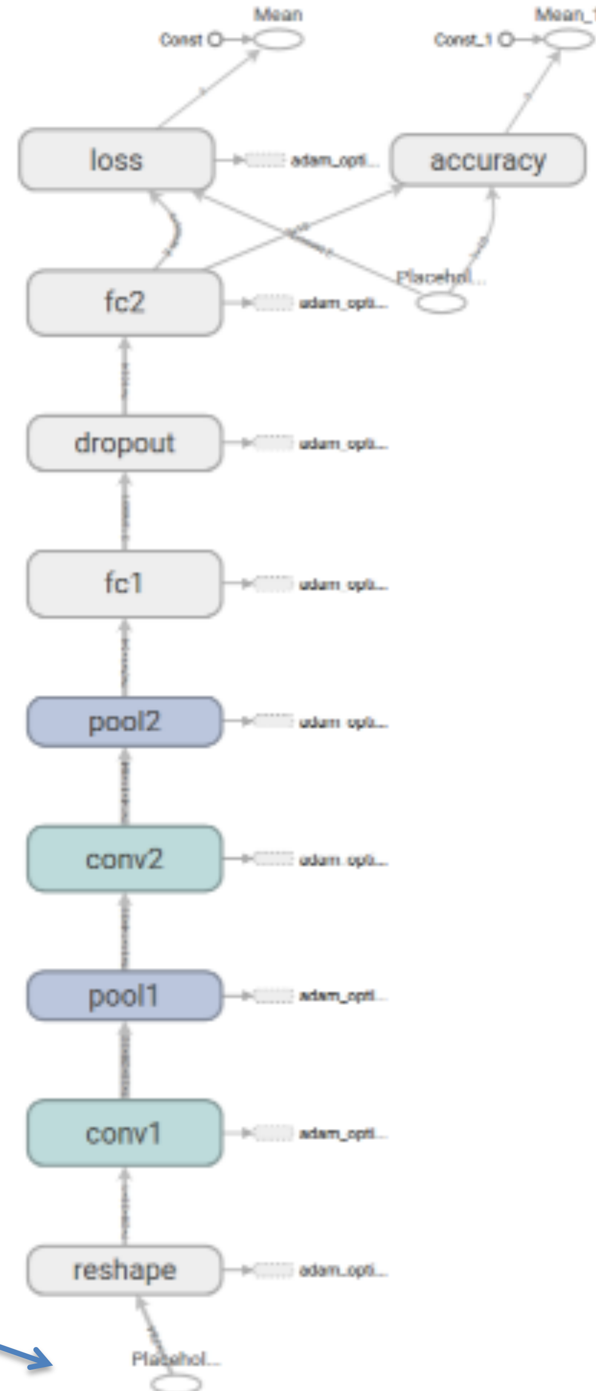
- **TF.NN.AVG_POOL**
- The output from executing the example code is:

```
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:

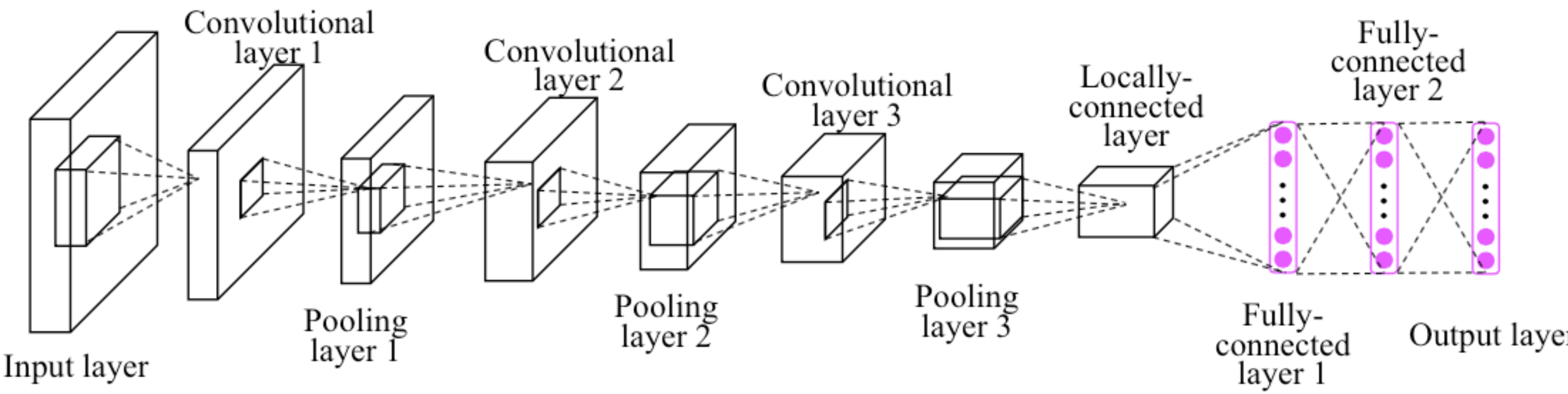
$$\frac{1.0+1.0+1.0+1.0+0.5+0.0+0.0+0.0+0.0}{9.0}$$

CNN Overview



input

CNN Overview



Source: https://github.com/aymericdamien/TensorFlow-Examples/blob/master/notebooks/3_NeuralNetworks/

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

Normalization

- **TF.NN.LOCAL_RESPONSE_NORMALIZATION (TF.NN.LRN)**

  (1.x-2.x)

- 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

Normalization

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

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 nonessential nodes to the graph
- Advise: It's worth waiting until the basics are comfortable before using these layers

High Level Layers

- **TF.CONTRIB.LAYERS.CONVOLUTION2D**

  In **(**2.x)is **tf.keras.layers.Conv2D**

- 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

- 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

High Level Layers

```
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 convolu-
tion2d's setup.
sess.run(tf.initialize_all_variables())
sess.run(conv2d)
```

High Level Layers

- 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

High Level Layers

- **TF.CONTRIB.LAYERS.FULLY_CONNECTED** v.2x **tf.keras.layers.Dense**

- A fully connected layer is one where every input is connected to every output
- This is a very 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 shorthand 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

High Level Layers

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

fc = tf.contrib.layers.fully_connected(features, num_out-
put_units=2)
# It's required to initialize all the variables first or
there'll be an error 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)
```

Layer Input

- Each layer serves a purpose in a CNN architecture
- 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

Examples: 1 and 2

- Let's use the MNIST database (Modified National Institute of Standards and Technology database)
- This is a large database of handwritten digits that is commonly used for training various image processing systems
- The database is also widely used for training and testing in the field of machine learning

Example 1

v.1.x

```python
## Import packages:
import tensorflow as tf

# reset everything to rerun:
tf.reset_default_graph()

## Configuration:
batch_size = 100
learning_rate = 0.01
training_epochs = 10

## Load Data:
# load mnist data set
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)

# input images
# None -> batch size can be any size, 784 -> flattened mnist image
x = tf.placeholder(tf.float32, shape=[None, 784], name="x-input")
# target 10 output classes
y_ = tf.placeholder(tf.float32, shape=[None, 10], name="y-input")

## Weights:
# model parameters will change during training so we use tf.Variable
W = tf.Variable(tf.zeros([784, 10]))

# bias
b = tf.Variable(tf.zeros([10]))
```

Example 1

v.1.x

```
## Implement model:
# y is our prediction
y = tf.nn.softmax(tf.matmul(x,W) + b)

## Cost function:
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))

## Accuracy:
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

## Specify optimizer:
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cross_entropy)
```

Example 1

v.1.x

```
## Execure the graph:
with tf.Session() as sess:
  # variables need to be initialized before we can use them
  sess.run(tf.global_variables_initializer())

  # perform training cycles - we will create batches from our training data and
  # iterate over them:
  for epoch in range(training_epochs):
    # number of batches in one epoch
    batch_count = int(mnist.train.num_examples/batch_size)
    for i in range(batch_count):
      batch_x, batch_y = mnist.train.next_batch(batch_size)

      # perform the train operations we defined earlier on batch, so we have to feed the data we promised when we declared the placeholders at the beginning.
      sess.run([train_op], feed_dict={x: batch_x, y_: batch_y})

    # Finally, we make sure to continuously print our progress and the final accuracy of the test images of MNIST.
    if epoch % 2 == 0:
      print("Epoch: ", epoch )
  print("Accuracy: ", accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels}))
  print("done")
```

Example 1

v.1.x

```
## Execure the graph:
with tf.Session() as sess:
  # variables need to be initialized before we can use them
  sess.run(tf.global_variables_initializer())

  # perform training cycles - we will create batches from our training data and
  # iterate over them:
  for epoch in range(training_epochs):
    # number of batches in one epoch
    batch_count = int(mnist.train.num_examp
    for i in range(batch_count):
      batch_x, batch_y = mnist.train.next_b

      # perform the train operations we defined earlier on batch, so we have to feed
the data we promised when we declared the placeholders at the beginning.
      sess.run([train_op], feed_dict={x: batch_x, y_: batch_y})

    # Finally, we make sure to contin
of the test images of MNIST.
    if epoch % 2 == 0:
      print("Epoch: ", epoch )
  print("Accuracy: ", accuracy.eval(f
mnist.test.labels}))
  print("done")
```

```
(<module>)>>> batch_y.shape
(100, 10)

(<module>)>>> batch_x.shape
(100, 784)
```

Python

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Epoch:  0
Epoch:  2
Epoch:  4
Epoch:  6
Epoch:  8
Accuracy:  0.9029
done
```

```
batch_size = 100
learning_rate = 0.01
training_epochs = 10
```

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

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

Loading Images

- TensorFlow makes 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/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)
```

Loading Images

- 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)
```

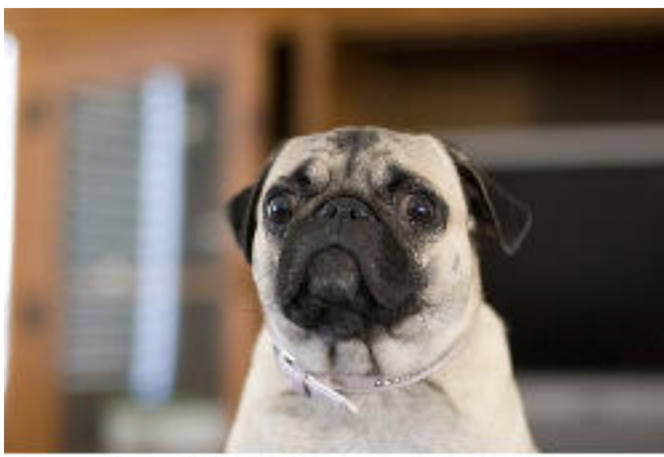
Image Formats

- An image may 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
- A large input image is counterproductive to training most CNNs
- 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

Image Formats

- In the following example there are two extremely different images of the same dog breed which should match as a Pug

- These images are filled with useless information which mislead a network during training


n02110958_2410.jpg


n02110958_4030.jpg

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 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)

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 - great for use as a preprocessed format because they aren't compressed and can be loaded into memory quickly

TFRECORD

create label

convert image

prepare to export image

bind label and image

save

```
# Reuse the image from earlier and give it a fake label
image_label = b'\x01'  # Assume the label data is in a one-hot
representation (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 Exam-
ple file to save space but not required.
example = tf.train.Example(features=tf.train.Features(feature={
            'label': tf.train.Fea-
ture(bytes_list=tf.train.BytesList(value=[image_label])),
            'image': tf.train.Fea-
ture(bytes_list=tf.train.BytesList(value=[image_bytes]))
        }))

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

load record filename

read image

parse label and image

read raw bytes

shape to fit convolution

read label

```
# 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 TFRecord files which may
# have more than one example in them.
tf_record_reader = tf.TFRecordReader()
_, tf_record_serialized = tf_record_reader.read(tf_record_file-
name_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 be-
tween 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 im-
age because it's required
# to reshape the input.

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

TFRECORD

- 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)
```

TFRECORD

- 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'
```

Image Manipulation

- Images visually highlight the importance of objects in the picture
- **Example**: A picture with a dog clearly visible in the center is considered more valuable than one with a dog in the background:

n02113978_3480.jpg n02113978_1030.jpg

Image Manipulation

- Image manipulation is best done as a preprocessing step in most scenarios
- An image can be cropped, resized and the color levels adjusted
- 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)

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)
```

CROPPING

- 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, tar-
get_height=2, target_width=1)

sess.run(bounding_crop)
```

- The output from executing the example code is:

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

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 will 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, tar-
get_height=4, target_width=4)

sess.run(pad)
```

PADDING

- 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)
```

PADDING

- This following example code increases the images height by one pixel and its width by a pixel as well
- 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)
```

PADDING

- tf.image.resize_image_with_crop_or_pad:
  - Resizes an image to a target width and height by either centrally cropping the image or padding it evenly with zeros.
  - If `width` or `height` is > the specified `target_width` or`target_height` respectively, this op centrally crops along that dimension.
  - If `width` or `height` is < the specified `target_width` or`target_height` respectively, this op centrally pads with 0 along that dimension.

PADDING

- 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)
```

FLIPPING

- When flipping each pixel’s location is reversed horizontally or vertically
- Technically speaking, flopping is the term used when flipping an image vertically
- Flipping images is useful with TensorFlow to give different perspectives of the same image for training
- 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

FLIPPING

- This example code flips a subset of the image horizontally and then vertically:

```
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)]
```

FLIPPING

- This code 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.ran-
dom_flip_left_right(top_left_pixels)
random_flip_vertical = tf.image.random_flip_up_down(ran-
dom_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)
```

SATURATION and BALANCE

- 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

SATURATION and BALANCE

- 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_pix-
el, 0.2)

sess.run(adjust_brightness)
```

- The output from executing the example code is:

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

SATURATION and BALANCE

- It’s best to avoid using this when possible and preprocess brightness changes first:

```
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)
```

SATURATION and BALANCE

- 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)
```

SATURATION and BALANCE

- 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)
```

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 network

COLORS

- Instead of a 3 component rank 1 tensors 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

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)
```

HSV

- Hue, Saturation and Value are what makes the 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]))
```

HSV

- 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 built-in 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)
```

LAB

- LAB is a useful colorspace because it can map to a larger number of perceivable colors than RGB
- Lab colorspace is not natively supported by TensorFlow
- Another Python library python-colormath has support for Lab conversion as well as other colorspaces
- The largest benefit using a Lab colorspace is it maps closer to humans perception of the difference in colors than RGB or HSV

CASTING IMAGES

- In these examples, tf.to_float is often used in order to illustrate changing an image's type to another format

- 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

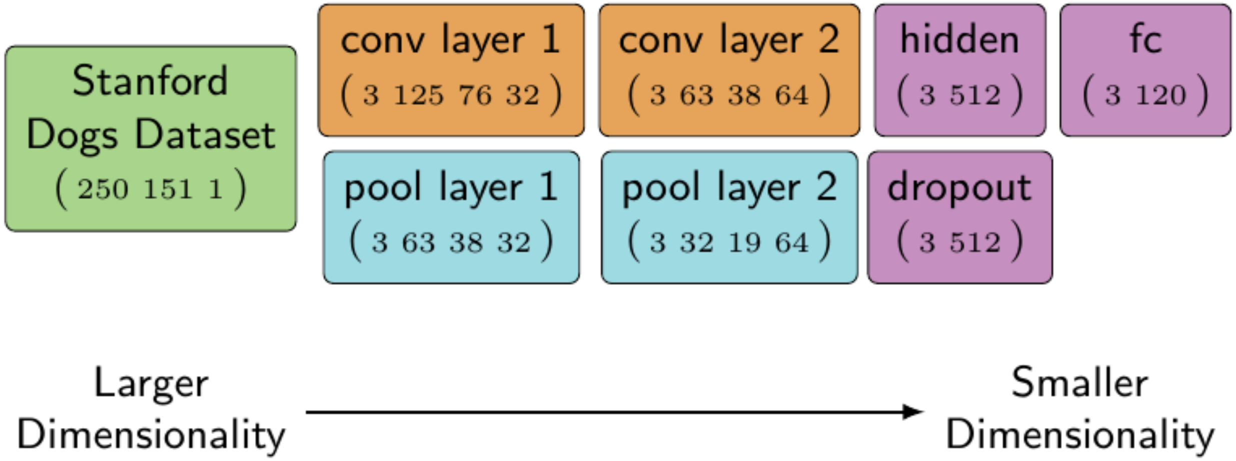
- Object recognition and categorization using TensorFlow required a basic understanding of:
  - convolutions (for CNNs), common layers (non-linearity, pooling, fc)
  - image loading, image manipulation and colorspaces

- The network needs to train on sequence of pictures

- Then it will be judged on how well it can guess a dog's breed based on a picture (for our case)

CNN Implementation

- The network described here includes the output TensorShape after each layer
- The layers are read from left to right and top to bottom where related layers are grouped together
- 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 at:
  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

Stanford Dogs Dataset

Stanford Dogs Dataset

Summary:

- 120 dog breeds
- ~150 images per class
- Total images: 20,580

Download dataset

Affenpinscher
(150 images)

ImageNet synset: n02110627

Afghan hound
(239 images)

ImageNet synset: n02088094

African hunting dog
(169 images)

ImageNet synset: n02116738

Airedale
(202 images)

ImageNet synset: n02096051

Stanford Dogs Dataset

Aditya Khosla **Nityananda Jayadevaprakash** **Bangpeng Yao** **Li Fei-Fei**

Stanford University

The Stanford Dogs dataset contains images of 120 breeds of dogs from around the world. This dataset has been built using images and annotation from ImageNet for the task of fine-grained image categorization. Contents of this dataset:

- **Number of categories:** 120
- **Number of images:** 20,580
- **Annotations:** Class labels, Bounding boxes

Download

You can download the dataset using the links below:

- Images (757MB)
- Annotations (21MB)
- Lists, with train/test splits (0.5MB)
- Train Features (1.2GB), Test Features (850MB)
- README

Dataset Reference

Primary:

Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao and Li Fei-Fei. **Novel dataset for Fine-Grained Image Categorization.** *First Workshop on Fine-Grained Visual Categorization (FGVC), IEEE Conference on Computer Vision and Pattern Recognition (CVPR),* 2011. [pdf] [poster] [BibTex]

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

Convert Images to TFRecords

- 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']
```

Convert Images to TFRecords

data separation

testing on 20%

training on 80%

way of stratification

```
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 file-
name. The breed is found by taking the directory name
image_filename_with_breed = map(lambda filename: (file-
name.split("/")[2], filename), image_filenames)

# Group each image by the breed which is the 0th element in
the tuple returned above
for dog_breed, breed_images in groupby(image_file-
name_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 im-
ages 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_testing_count), 2) > 0.18, "Not enough testing images."
```

Convert Images to TFRecords

- 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

- 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