UC Berkeley Extension

Machine Learning With TensorFlow

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

Instructor Alexander I. Iliev, Ph.D.

Course Content Outline

- **Neural Networks 2/2**
- Object recognition with Convolutional Neural Network (CNN)
- Activation Functions

- **Working with images**
- Loading Images
- Image formats and manipulation
- CNN Implementation
- Training
- Recurrent Neural Network (RNN) Midterm / Project proposal due (30pts)

- **Sequence classification and labeling**
- Imdb Movie review dataset
- Gradient clipping
- Sequence labeling using optical character recognition
- Bidirectional Recurrent Neural Network (LSTM)

- **Predictive coding**
- Character-level language modeling
- Preprocessing data
- Predictive coding model
- Project Presentations 1/2

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

Object Recognition and Classification

recall

- ImageNet, a database of labeled images, is where computer vision and deep learning saw a recent rise in popularity
- Convolutional Neural Networks (CNNs) primarily used for computer vision related tasks but are not limited to working with images
- For images, the values in the tensor are pixels ordered in a grid corresponding with the width and height of the image

Object Recognition and Classification

recall

- The dataset used in training this CNN model is a subset of the images available in ImageNet named the Stanford's Dogs Dataset - http://vision.stanford.edu/aditya86/ImageNetDogs/

n02110185_712.jpg n02110185_4186.jpg n02110185_1532.jpg

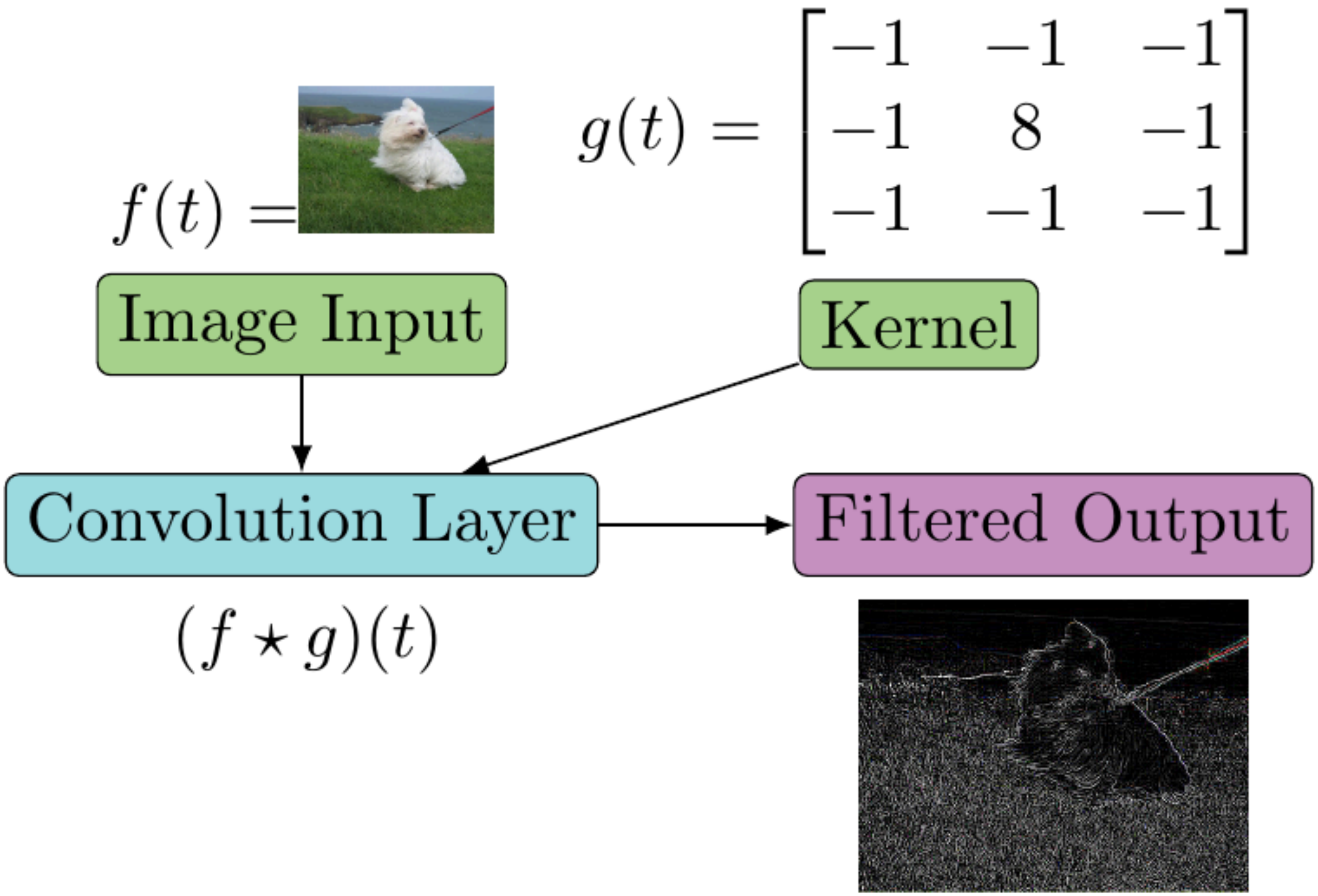
Convolutional Neural Networks

- A CNN is a neural network which has at least one layer tf.nn.conv2d that performs a convolution between its input ***f*** and a configurable kernel ***g*** generating the layer's output

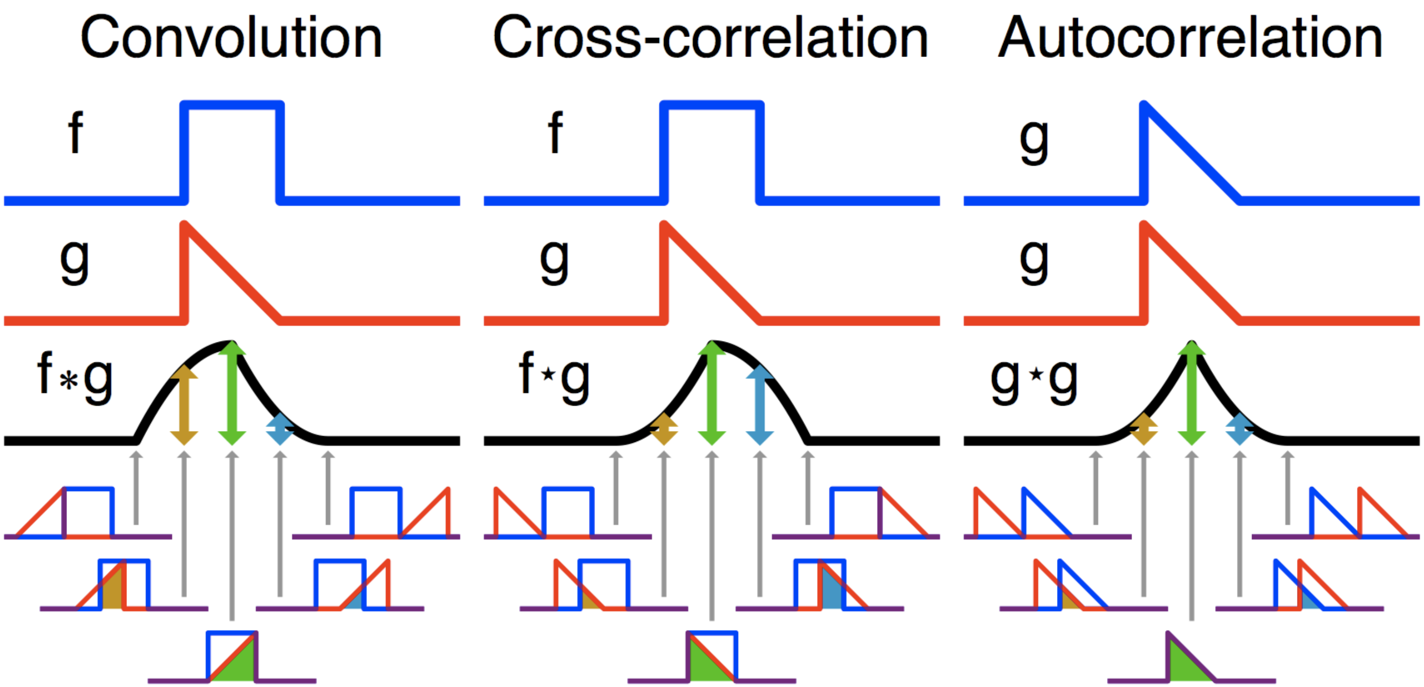
- In a simplified definition, a convolution's goal is to apply a kernel (filter) to every point in a tensor and generate a filtered output by sliding the kernel over an input tensor

- An example of the filtered output is edge detection in images

Convolutional Neural Networks



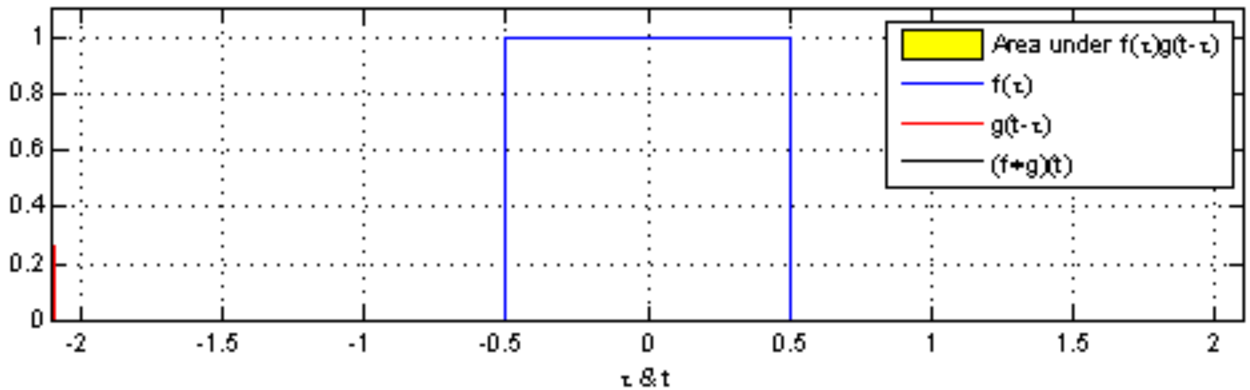
Convolution vs. Correlation



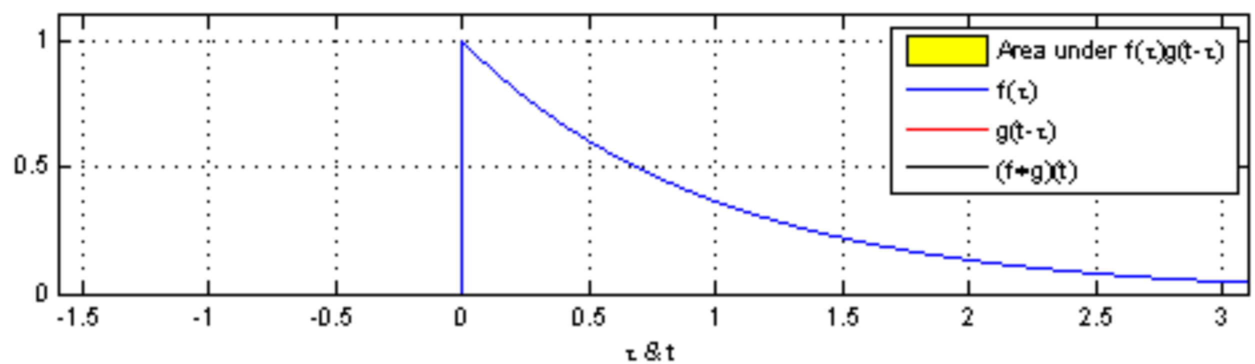
Source Wikipedia

Convolution

- Convolving two signals:



box vs box



spiky vs box

Source Wikipedia

UC Berkeley Extension

Convolutional Neural Networks

- In CNN clusters neurons will activate based on patterns learned from training

  Example: after training, a CNN will have certain layers that activate when a horizontal line passes through it

- Layering multiple simple patterns to match complex patterns, a.k.a. filters or kernels is what we need to do

- Our goal is to adjust these kernel weights until they accurately match the training data, often accomplished by combining multiple different layers and learning weights using gradient descent

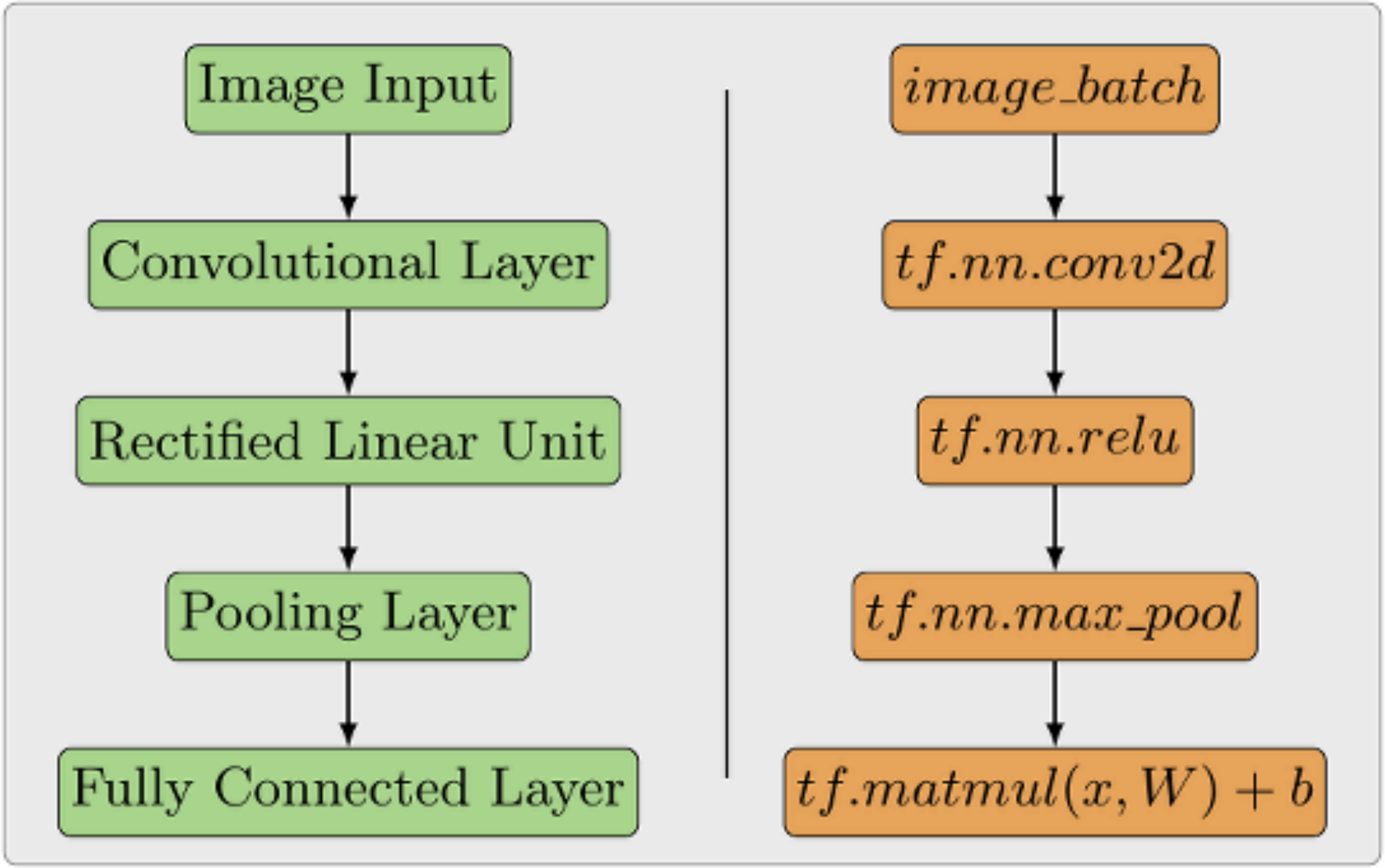
Convolutional Neural Networks

- A simple CNN architecture may combine different types of layers:
  - 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

Convolutional Neural Networks



Convolutional Neural Networks

```
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), Dimen-
sion(3)])
```

Convolutional Neural Networks

- 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:

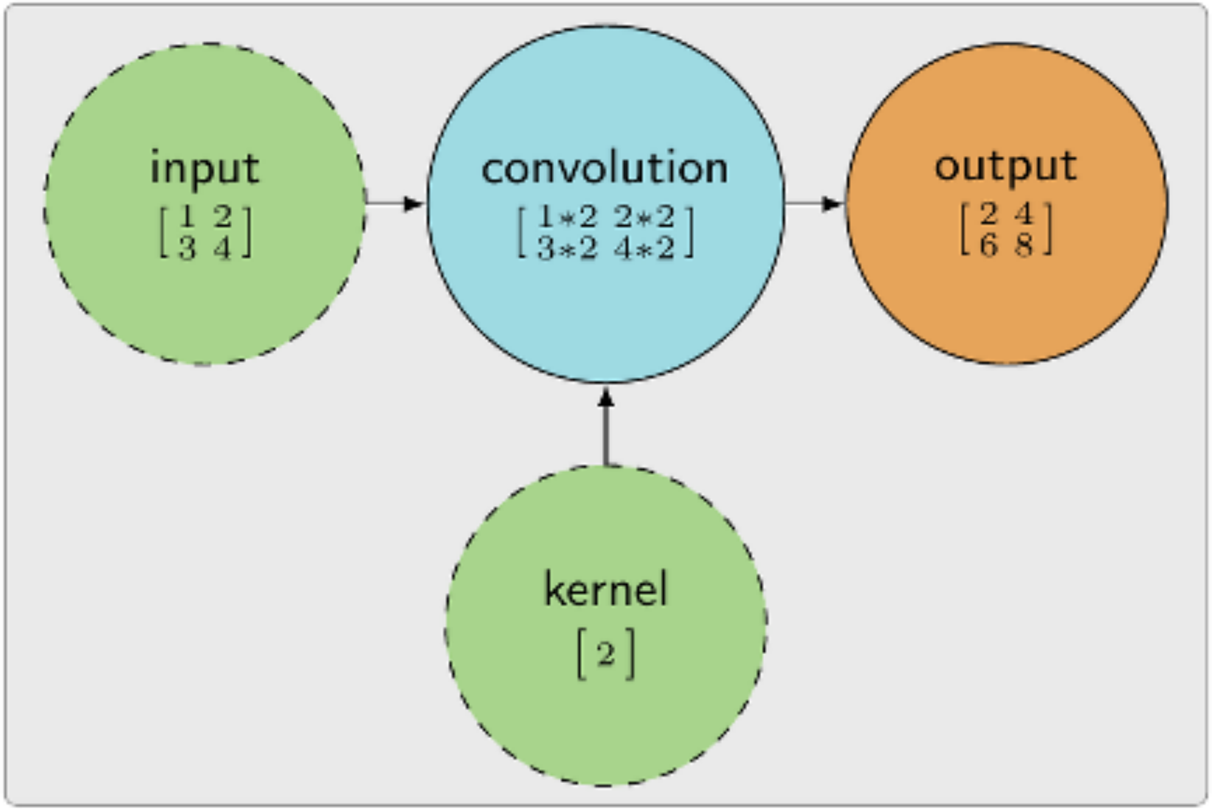
```
array([  0, 255,   0], dtype=int32)
```

- Instead of loading images from disk, the image_batch variable will act as if it were images loaded as part of an input pipeline

Convolution

- 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 slides

Convolution



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 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 the pixel value and the 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)
```

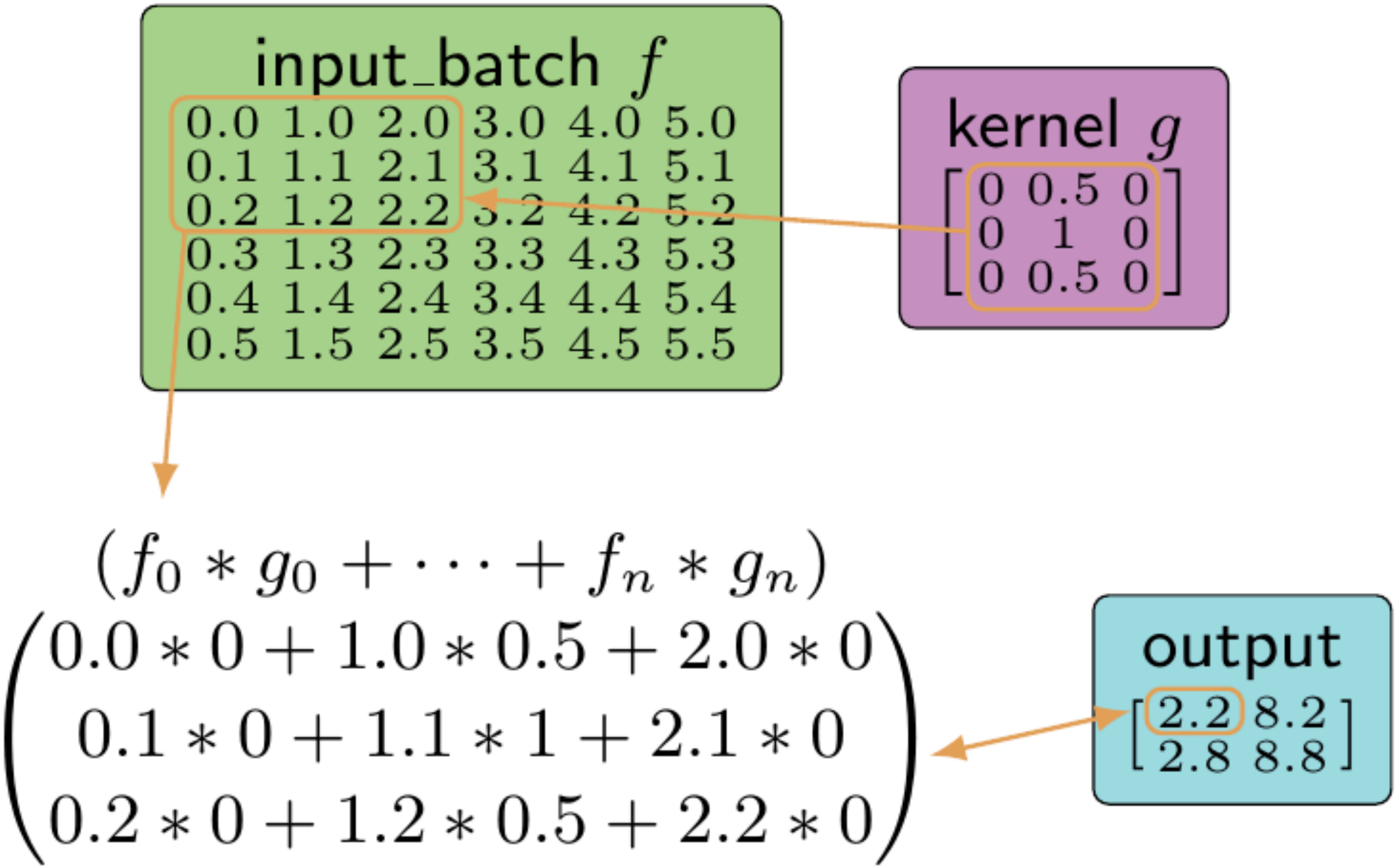
Strides

- 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
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 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 will avoid

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

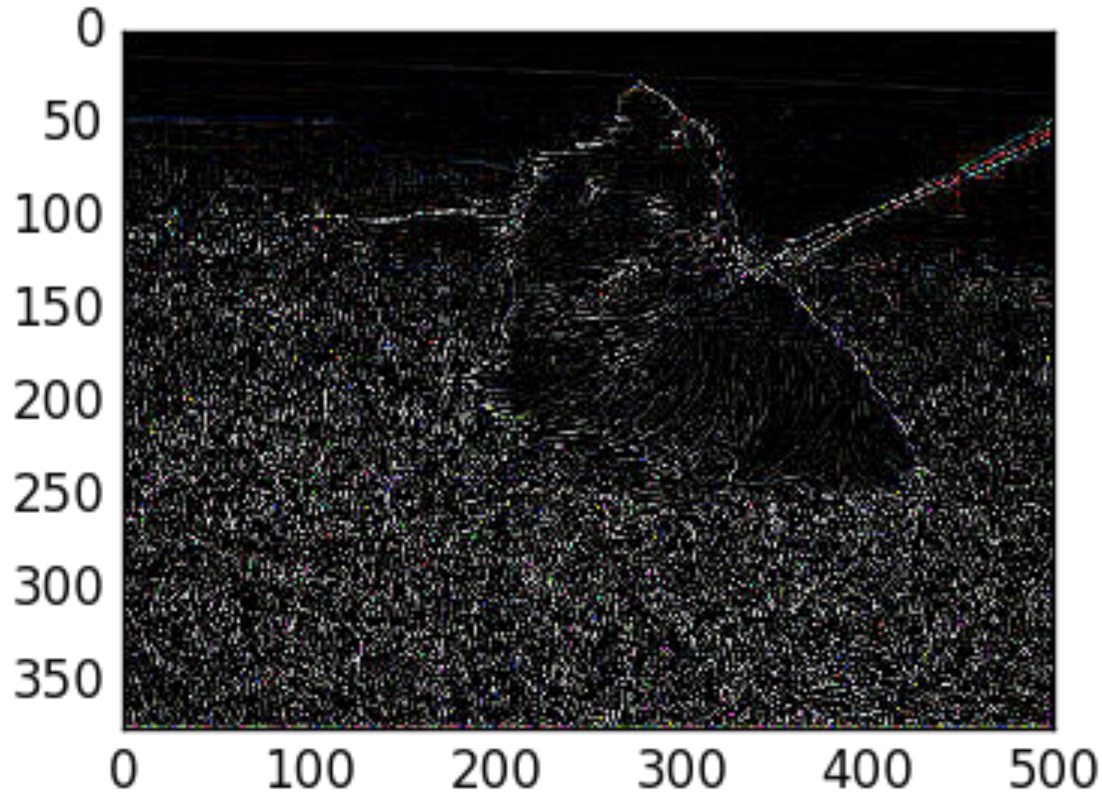
Kernels in Depth

- **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



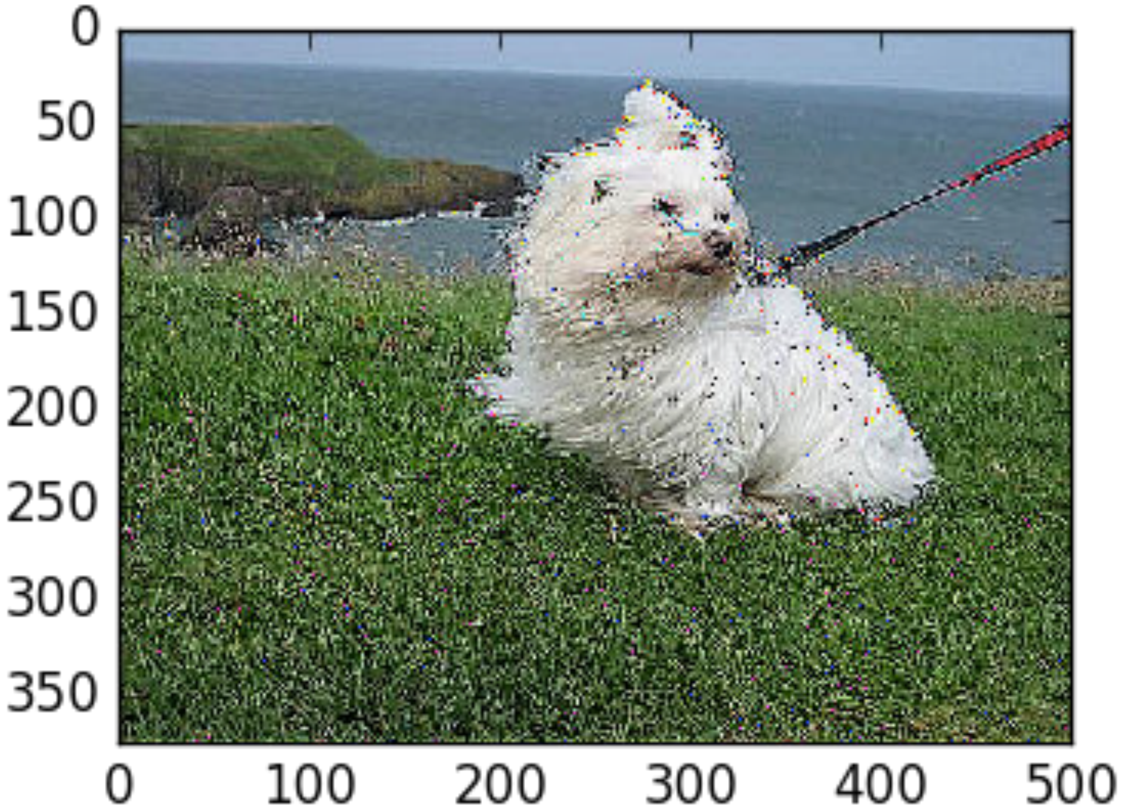
Kernels in Depth

- **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



UC Berkeley Extension

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 over-fitting
  - 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**

- 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**
- 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**
- 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 are capable of representing 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 although it sacrifices information
- When starting out, using tf.nn.relu is recommended but advanced users may create their own

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. 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

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]

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

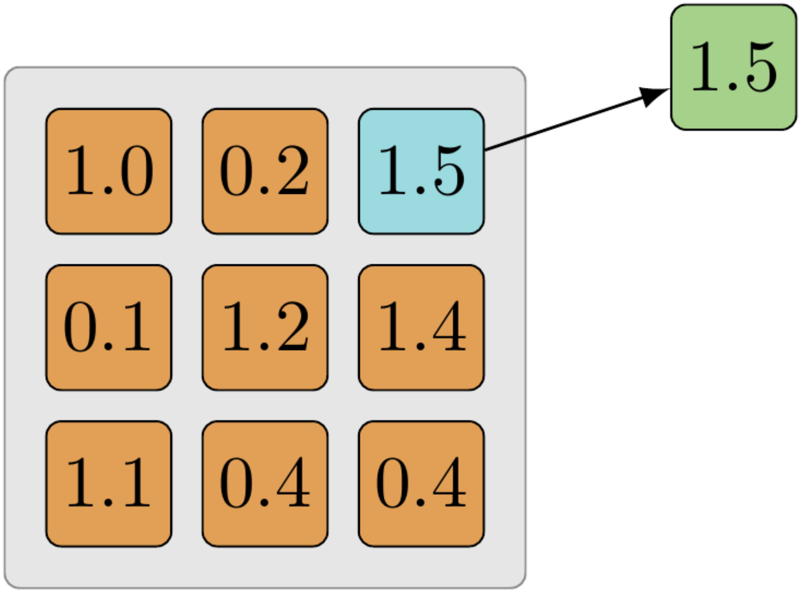
- Pooling layers reduce over-fitting and improving performance by reducing the size of the input
- They're used to scale down input while keeping important information for the next layer
- It's possible to reduce the size of the input using a tf.nn.conv2d alone but these layers execute much faster

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

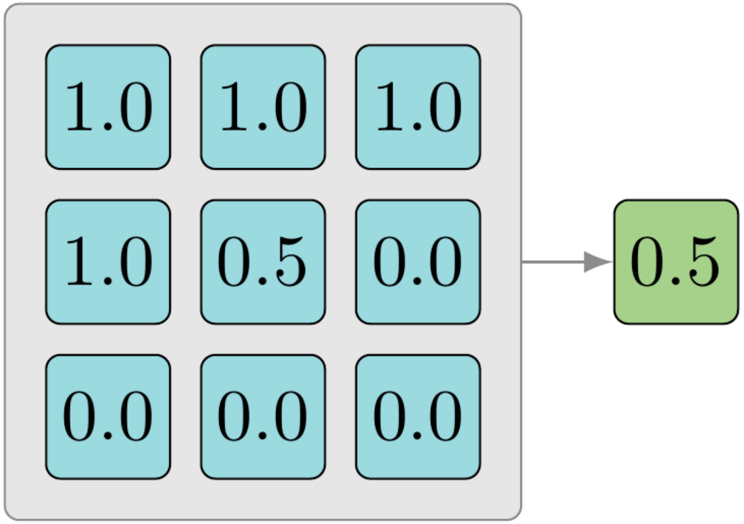
- In this case, the largest value of the tensor is 1.5 and is returned in the same format as the input.

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**

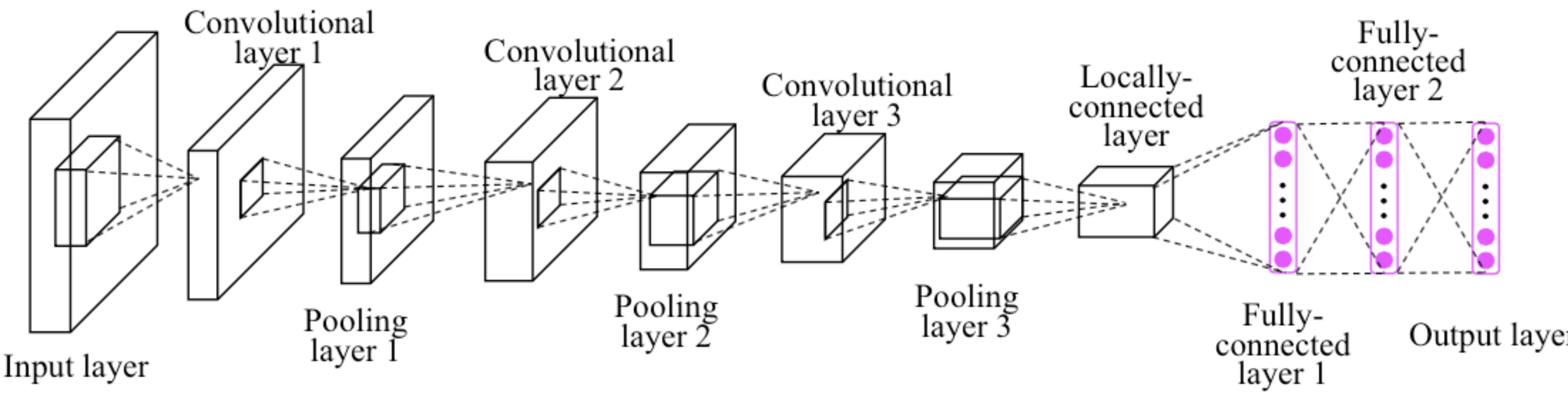
- The output from executing the example code is:

```
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:

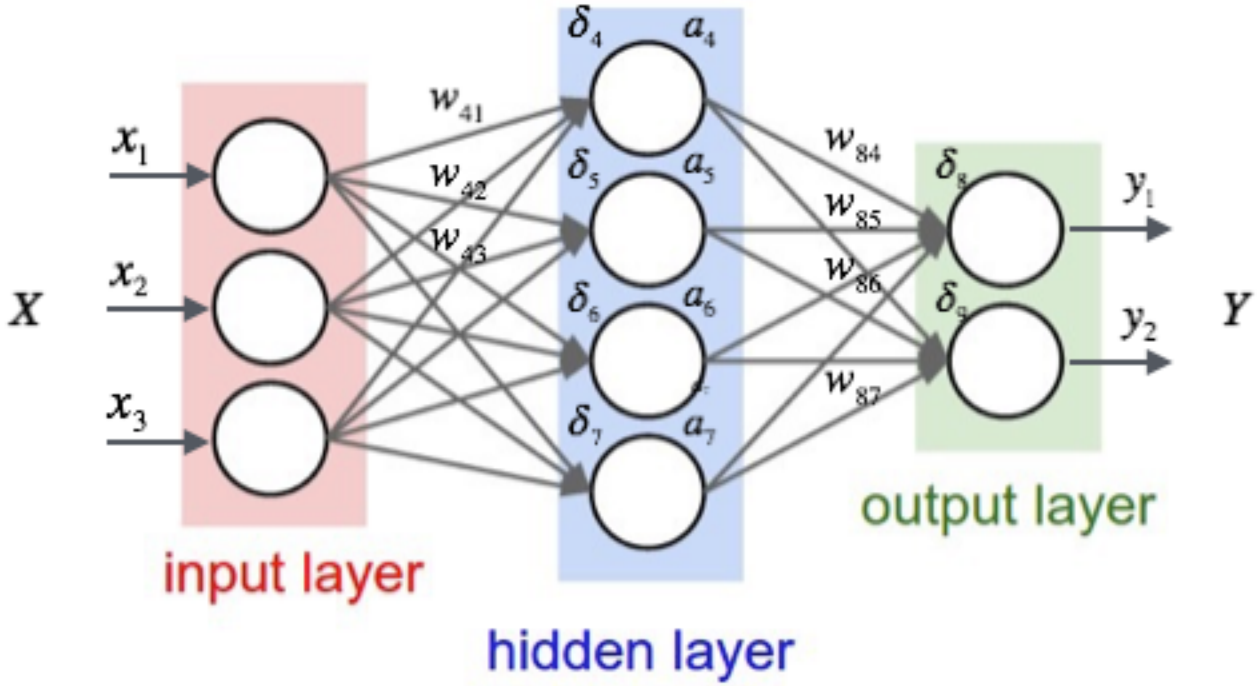
$$\frac{1.0+1.0+1.0+1.0+0.5+0.0+0.0+0.0+0.0}{9.0}$$

CNN Overview



Source github

CNN Overview



Source medium.com

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)**

- 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**
- 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
- 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.

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**
- 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 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

Source MNIST

Example 1

```
## Import packages:
#   First, we need to import tensorflow and add a few parameters which we will use:
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:
#   The network we are going to build will use the MNIST data to train its weights and
biases. In tensorflow, we feed this data into the model (tensorflow calls this a graph).
# We’ll do this later but a placeholder is such a variable. We create now two
placeholders for our flattened 28x28 big image data and our 10 labels.

# 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:
#   The weights are according to the weight matrix of a neural network and the biases of
each neuron.  The shape of these variables corresponds to the size of our network:

# 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]))
```

Source github

Example 1

```
## Implement model:
#   we have prepared all the ingredients for our model. We can now define our model which will calculate our prediction y. In this simple neural network, we have no hidden layer and perform a softmax over 10 prediction classes.

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

## Cost function:
#   here, we use the cross-entropy error based on our prediction y and our target value y_. This is our cost:
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))

## Accuracy:
#   Another value we want to calculate is the accuracy of our parameters. We don’t need to use any tensorflow specific elements since this variable is not used during the training of the model. However, it does come with some handy functions which we shall use.
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

## Specify optimizer:
# Optimizer is an "operation" which we can execute in a session
# To train our model we use a gradient descent method. Tensorflow comes with several techniques already implemented. As a result, we get an operation. This operation is tied to our graph and once we start a session, we can execute this optimizer operation.
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cross_entropy)
```

Example 1

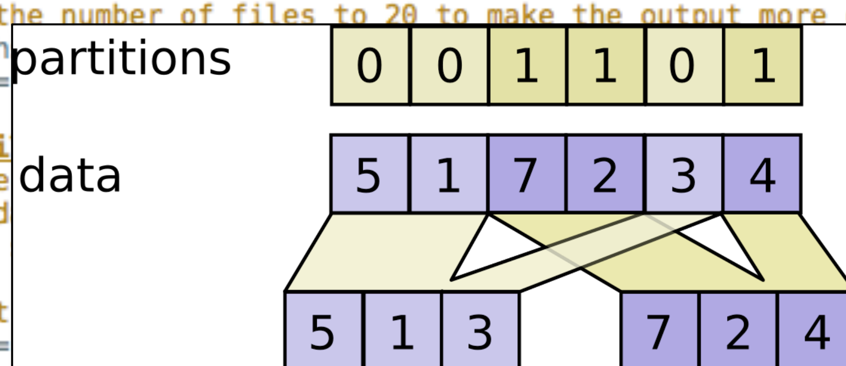
```
## Execure the graph:
#   We execute our graph on our computing hardware such as CPUs and GPUs. After we have created a session we need to initialize all the tensorflow variables. We have to do this before we do anything else. To do that we perform the initialization operation on our session. We can execute operations with the run() function of our session.

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 2

```python
## Load the data first and specify paramaters:

import tensorflow as tf
import random
from tensorflow.python.framework import ops
from tensorflow.python.framework import dtypes

# Load the Label Data:
dataset_path      = "./data/mnist/"
test_labels_file  = "test-labels.csv"
train_labels_file = "train-labels.csv"

# Define system parameters:
test_set_size = 5
IMAGE_HEIGHT  = 28
IMAGE_WIDTH   = 28
NUM_CHANNELS  = 3
BATCH_SIZE    = 5

## Lets assign an int value to our string text. For this example, we are going to
# simply convert the string label into an integer since they are all numbers:

def encode_label(label):
  return int(label)

def read_label_file(file):
  f = open(file, "r")
  filepaths = []
  labels = []
  for line in f:
    filepath, label = line.split(",")
    filepaths.append(filepath)
    labels.append(encode_label(label))
  return filepaths, labels

# reading labels and file path
train_filepaths, train_labels = read_label_file(dataset_path + train_labels_file)
test_filepaths, test_labels = read_label_file(dataset_path + test_labels_file)
```

Example 2

```
## Some Optional Processing on Our String Lists:
# transform the relative image path into a full image path and for the sake of
# this example we are also going to concat the given train and test set. We then
# shuffle the data and create our own train and test set later on:

# transform relative path into full path
train_filepaths = [ dataset_path + fp for fp in train_filepaths]
test_filepaths = [ dataset_path + fp for fp in test_filepaths]

# for this example we will create or own test partition
all_filepaths = train_filepaths + test_filepaths
all_labels = train_labels + test_labels

# we limit the number of files to 20 to make the output more clear:
all_filepath
all_labels =

## Start Bui
# With those                                                    de to use the train
# and test d                                                    the next steps for
# both sets

# convert st
all_images =                                                    .string)
all_labels = ops.convert_to_tensor(all_labels, dtype=dtypes.int32)

## Lets Partition the Data:
# This step is optional. Since we have all our 20 samples in one (big) set, we want
# to perform some partitioning to build a test and train set:

# create a partition vector
partitions = [0] * len(all_filepaths)
partitions[:test_set_size] = [1] * test_set_size
random.shuffle(partitions)

# partition our data into a test and train set according to our partition vector
train_images, test_images = tf.dynamic_partition(all_images, partitions, 2)
train_labels, test_labels = tf.dynamic_partition(all_labels, partitions, 2)
```



Example 2

```
## Build the Input Queues and Define How to Load Images:
# The slice_input_producer will slice our tensors into single instances and queue
# them up using threads. We then use the path information to read the file into
# our pipeline and decode it using the jpg decoder:

# create input queues
train_input_queue = tf.train.slice_input_producer(
                                    [train_images, train_labels],
                                    shuffle=False)
test_input_queue = tf.train.slice_input_producer(
                                    [test_images, test_labels],
                                    shuffle=False)

# process path and string tensor into an image and a label
file_content = tf.read_file(train_input_queue[0])
train_image = tf.image.decode_jpeg(file_content, channels=NUM_CHANNELS)
train_label = train_input_queue[1]

file_content = tf.read_file(test_input_queue[0])
test_image = tf.image.decode_jpeg(file_content, channels=NUM_CHANNELS)
test_label = test_input_queue[1]

## Group Samples into Batches:
# If you run 'train_image' in a session you would get a single image i.e. (28, 28, 1)
# since according to the dimensions of our mnist images. Training a model on single
# images can be inefficient which is why we would like to queue up images into a batch
# and perform our operations on a whole batch of images instead of a single one.

# define tensor shape
train_image.set_shape([IMAGE_HEIGHT, IMAGE_WIDTH, NUM_CHANNELS])
test_image.set_shape([IMAGE_HEIGHT, IMAGE_WIDTH, NUM_CHANNELS])

# To use 'tf.train_batch' we need to define the shape of our image tensors before they
# can be combined into batches. For this example, we will use a batch size of 5
# samples.
```

Example 2

```
# collect batches of images before processing
train_image_batch, train_label_batch = tf.train.batch(
                                    [train_image, train_label],
                                    batch_size=BATCH_SIZE
                                    #,num_threads=1
                                    )
test_image_batch, test_label_batch = tf.train.batch(
                                    [test_image, test_label],
                                    batch_size=BATCH_SIZE
                                    #,num_threads=1
                                    )

print("input pipeline ready")

## Finally, we run our session:
# We have finished building our input pipeline. However, if we would now try to access
# e.g. test_image_batch, we would not get any data as we have not started the threads
# who will load the queues and push data into our tensorflow objects. After doing that,
# we will have two loops one going over the training data and one going over the test
# data.

with tf.Session() as sess:

  # initialize the variables
  sess.run(tf.global_variables_initializer())

  # initialize the queue threads to start to shovel data
  coord = tf.train.Coordinator()
  threads = tf.train.start_queue_runners(coord=coord)

  print("from the train set:")
  for i in range(20):
    print(sess.run(train_label_batch))

  print("from the test set:")
  for i in range(10):
    print(sess.run(test_label_batch))

  # stop our queue threads and properly close the session
  coord.request_stop()
  coord.join(threads)
  sess.close()
```

Example 2

```
test_set_size = 5
IMAGE_HEIGHT  = 28
IMAGE_WIDTH   = 28
NUM_CHANNELS  = 3
BATCH_SIZE    = 5
```

```
from the train set:
[5 0 4 2 1]
[3 1 3 3 6]
[1 2 8 6 9]
[5 0 4 2 1]
[3 1 3 3 6]
[1 2 8 6 9]
[5 0 4 2 1]
[3 1 3 3 6]
[1 2 8 6 9]
[5 0 4 2 1]
[3 1 3 3 6]
[1 2 8 6 9]
[5 0 4 2 1]
[3 1 3 3 6]
[1 2 8 6 9]
[5 0 4 2 1]
[3 1 3 3 6]
[1 2 8 6 9]
[5 0 4 2 1]
[3 1 3 3 6]
from the test set:
[1 9 4 5 7]
[1 9 4 5 7]
[1 9 4 5 7]
[1 9 4 5 7]
[1 9 4 5 7]
[1 9 4 5 7]
[1 9 4 5 7]
[1 9 4 5 7]
[1 9 4 5 7]
[1 9 4 5 7]
```

```
test_set_size = 2
IMAGE_HEIGHT  = 28
IMAGE_WIDTH   = 28
NUM_CHANNELS  = 3
BATCH_SIZE    = 4
```

```
from the train set:
[5 0 4 1]
[9 2 1 3]
[1 4 3 5]
[3 6 1 7]
[8 6 5 0]
[4 1 9 2]
[1 3 1 4]
[3 5 3 6]
[1 7 8 6]
[5 0 4 1]
[9 2 1 3]
[1 4 3 5]
[3 6 1 7]
[8 6 5 0]
[4 1 9 2]
[1 3 1 4]
[3 5 3 6]
[1 7 8 6]
[5 0 4 1]
[9 2 1 3]
from the test set:
[2 9 2 9]
[2 9 2 9]
[2 9 2 9]
[2 9 2 9]
[2 9 2 9]
[2 9 2 9]
[2 9 2 9]
[2 9 2 9]
[2 9 2 9]
[2 9 2 9]
```

```
test_set_size = 1
IMAGE_HEIGHT  = 28
IMAGE_WIDTH   = 28
NUM_CHANNELS  = 3
BATCH_SIZE    = 3
```

```
from the train set:
[5 0 4]
[1 9 2]
[1 3 1]
[4 3 3]
[6 1 7]
[2 8 6]
[9 5 0]
[4 1 9]
[2 1 3]
[1 4 3]
[3 6 1]
[7 2 8]
[6 9 5]
[0 4 1]
[9 2 1]
[3 1 4]
[3 3 6]
[1 7 2]
[8 6 9]
[5 0 4]
from the test set:
[5 5 5]
[5 5 5]
[5 5 5]
[5 5 5]
[5 5 5]
[5 5 5]
[5 5 5]
[5 5 5]
[5 5 5]
[5 5 5]
```