

Convolutional Neural Networks (CNN)

1. Convolution on Image

Filter (or Kernel)

- Modify or enhance an image by filtering
 - Filter image to emphasize certain features or remove other features
 - Filtering include smoothing, sharpening and edge enhancement

Convolution in 2D

Table of Contents

- I. 1. Convolution on Image
- II. 2. Convolutional Neural Networks (CNN)
 - I.2.1. Convolutional Operator
 - I.2.2. Nonlinear activation function
 - I.2.3. Pooling
 - I.2.4. Inside Convolution Layer
- III. 3. CNN with TensorFlow
 - I.3.1. Import Library
 - I.3.2. Load MNIST Data
 - I.3.3. Build Model
 - IV. 3.4. Define a CNN Shape
 - V. 3.5. Define Weights, Biases and Network
 - VI. 3.6. Define Loss, Initializer and Optimizer
 - VII. 3.7. Summary of Model
 - VIII. 3.8. Define Configuration
 - IX. 3.9. Optimization
 - X. 3.10. Test
 - IV. 4. Deep Learning of Things

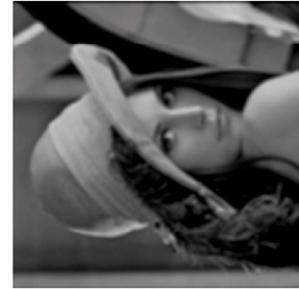
Filter (or Kernel)

- Modify or enhance an image by filtering
 - Filter image to emphasize certain features or remove other features
 - Filtering include smoothing, sharpening and edge enhancement

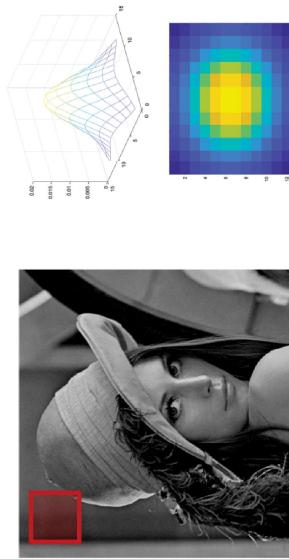
Convolution in 2D

1	1	1	0	0
0	1	1	0	
0	0	1	1	
0	0	1	1	
0	1	0	0	

Convolved Feature
Image



Output



Kernel|
Image

```
In [1]: # Import Libraries
import numpy as np
import matplotlib.pyplot as plt
from scipy.misc import imread, imresize
from scipy.signal import convolve2d
from six.moves import cPickle
%matplotlib inline
```

```
In [2]: # Import image
input_image = cPickle.load(open('./image_files/lena.pkl', 'rb'))

# Edge filter
image_filter = np.array([[1, 0, 1],
                        [-1, 0, 1],
                        [-1, 0, 1]])

# Compute feature
feature = convolve2d(input_image, image_filter, boundary='symm', mode='same')
```

```
In [3]: # Plot
fig = plt.figure(figsize=(10, 6))
ax1 = fig.add_subplot(1, 3, 1)
ax1.imshow(input_image, 'gray')
ax1.set_title('Input image (512 x 512)', fontsize=15)
ax1.set_xticks([])
ax1.set_yticks([])

ax2 = fig.add_subplot(1, 3, 2)
ax2.imshow(image_filter, 'gray')
ax2.set_title('Image filter (3 x 3)', fontsize=15)
ax2.set_xticks([])
ax2.set_yticks([])

ax3 = fig.add_subplot(1, 3, 3)
ax3.imshow(feature, 'gray')
ax3.set_title('Feature', fontsize=15)
ax3.set_xticks([])
ax3.set_yticks([])

plt.show()
```

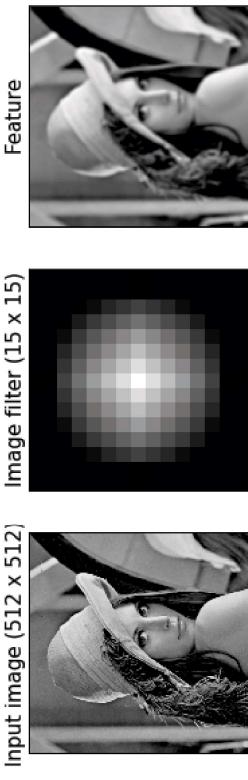


```
In [5]: # Plot
fig = plt.figure(figsize=(10, 6))
ax1 = fig.add_subplot(1, 3, 1)
ax1.imshow(input_image, 'gray')
ax1.set_title('Input image (512 x 512)', fontsize=15)
ax1.set_xticks([])
ax1.set_yticks([])

ax2 = fig.add_subplot(1, 3, 2)
ax2.imshow(image_filter, 'gray')
ax2.set_title('Image filter (15 x 15)', fontsize=15)
ax2.set_xticks([])
ax2.set_yticks([])

ax3 = fig.add_subplot(1, 3, 3)
ax3.imshow(feature, 'gray')
ax3.set_title('Feature', fontsize=15)
ax3.set_xticks([])
ax3.set_yticks([])

plt.show()
```



2. Convolutional Neural Networks (CNN)

Convolutional Networks

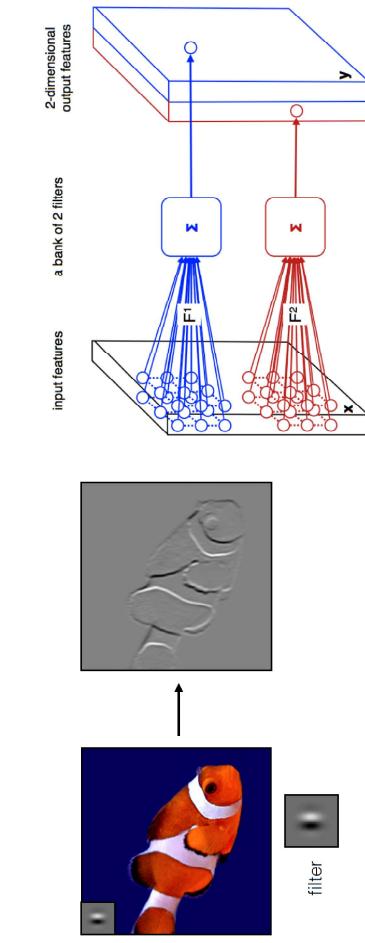
- Simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers
- Convolution can be interpreted as matrix multiplication

```
In [4]: # Import image
input_image = cPickle.load(open('./image_files/lena.pkl', 'rb'))

# Gaussian filter
image_filter = 1/273*np.array([[1, 4, 7, 4, 1]
                               ,[4, 16, 26, 16, 4]
                               ,[7, 26, 41, 26, 7]
                               ,[4, 16, 26, 16, 4]
                               ,[1, 4, 7, 4, 1]])

image_filter = imresize(image_filter, [15, 15])

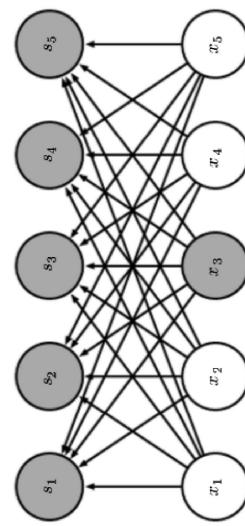
# Compute feature
feature = convolve2d(input_image, image_filter, boundary='symm', mode='same')
```



2.1. Convolutional operator

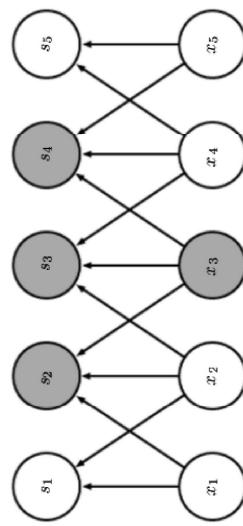
Matrix multiplication

- Every output unit interacts with every input units

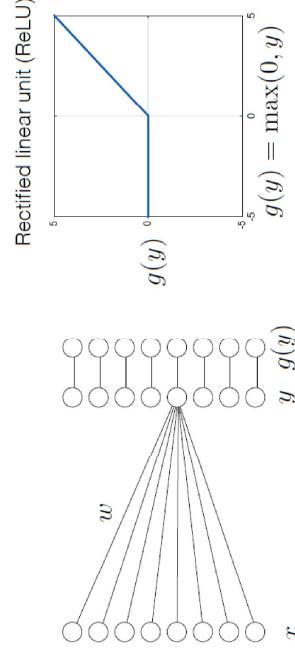


Convolution

- Local connectivity
- Weight sharing
- Typically have sparse interactions
- Accomplished by making the filter smaller than input (sparse interactions)

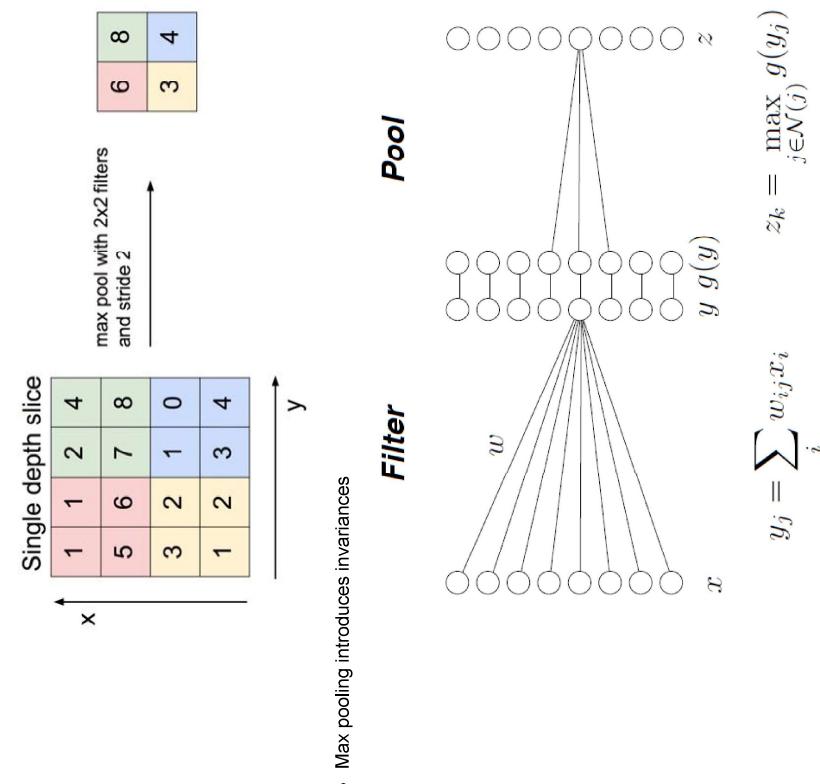


2.2. Nonlinear activation function



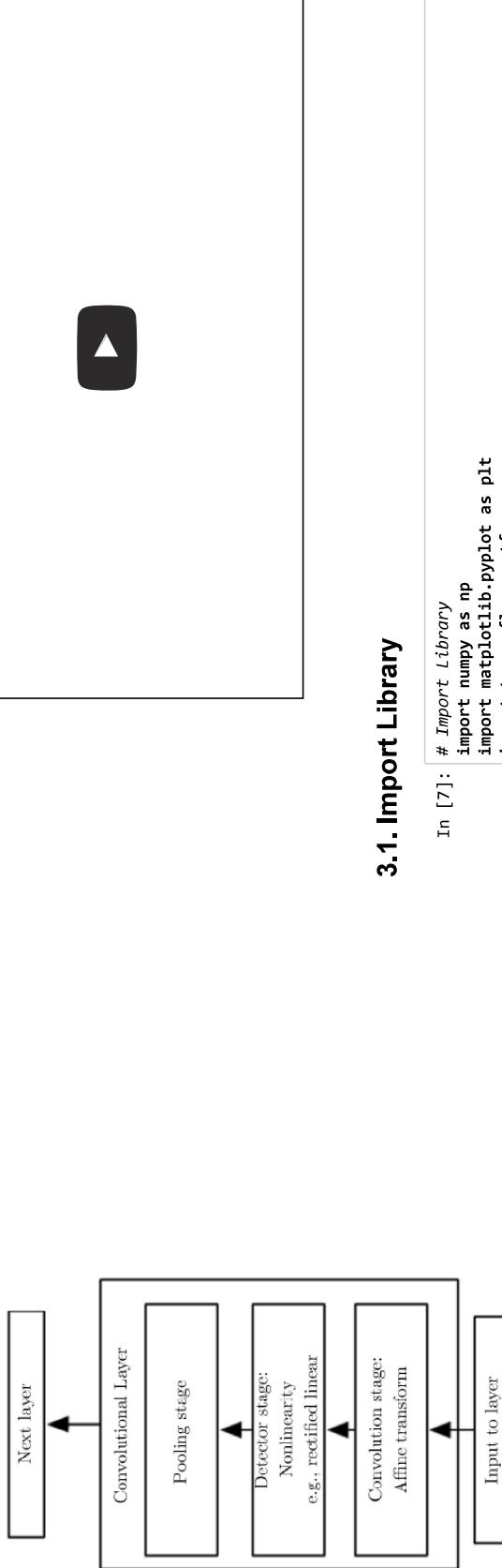
2.3. Pooling

- Compute a maximum value in a sliding window (max pooling)
- Pooling size : 2×2



2.4. Inside Convolution Layer

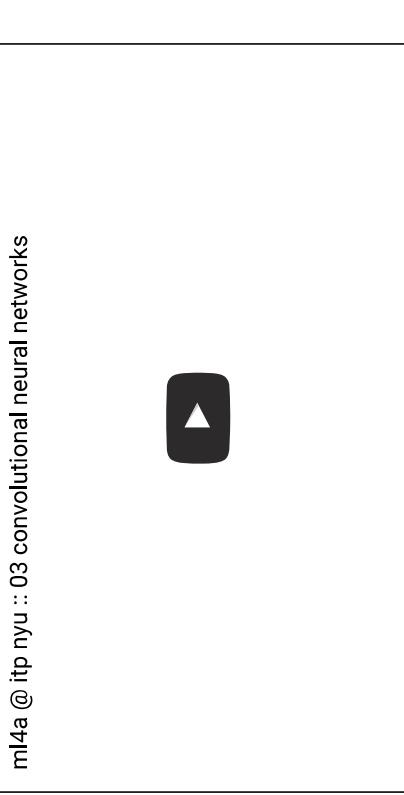
- First, the layer performs several convolutions to produce a set of linear activations
- Second, each linear activation is run through a nonlinear activation function
- Third, use pooling function to modify the output of the layer further



In [6]:

```
%%html
<center><iframe src="https://www.youtube.com/embed/z6k_RMKExIQ?start=515&end=6132"
width="560", height="315" frameborder="0" allowfullscreen></center>
```

ml4a @ ltp nyu :: 03 convolutional neural networks



3.1. Import Library

```
In [7]: # Import Library
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

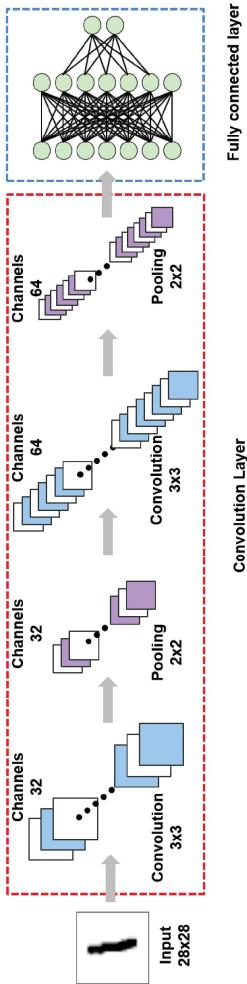
3. CNN with TensorFlow

- MNIST example
- Classifying hand written digits

3.2. Load MNIST Data

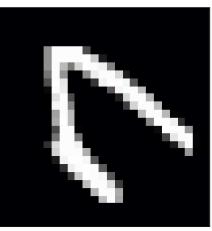
- Download MNIST data from tensorflow tutorial example

```
In [8]: from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```



```
In [9]: # Check data
train_x, train_y = mnist.train.next_batch(10)
img = train_x[9,:,:].reshape(28, 28)
plt.figure(figsize=(5, 3))
plt.imshow(img, gray')
plt.title("Label : {} ".format(np.argmax(train_y[9])))
plt.xticks([])
plt.yticks([])
plt.show()
```

Label : 7



1	1	1	0	0
0	1	1	0	
	0	0	1	1
	x ₁	x ₀	x ₁	x ₁
	0	0	1	0
	x ₀	x ₁	x ₀	x ₁
0	1	1	0	0
x ₁	x ₀	x ₁	x ₁	x ₁

Image Convolved Feature

- Filter size : 3×3
- Stride : The stride of the sliding window for each dimension of input
- Padding : Allow us to control the kernel width and the size of the output independently
 - 'SAME' : zero padding
 - 'VALID' : No padding

3.3. Build Model

Convolution layers

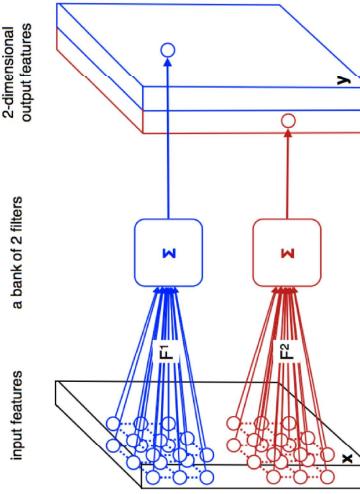
- First, the layer performs several convolutions to produce a set of linear activations
- Second, each linear activation is run through a nonlinear activation function
- Third, use pooling function to modify the output of the layer further

Fully connected layers

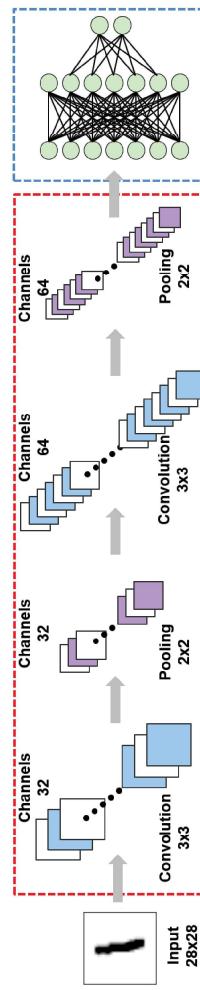
- Simple multi layer perceptrons

```
conv1 = tf.nn.conv2d(x, weights['conv1'], strides=[1,1,1,1], padding = 'SAME')
```

- The number of channels: 2

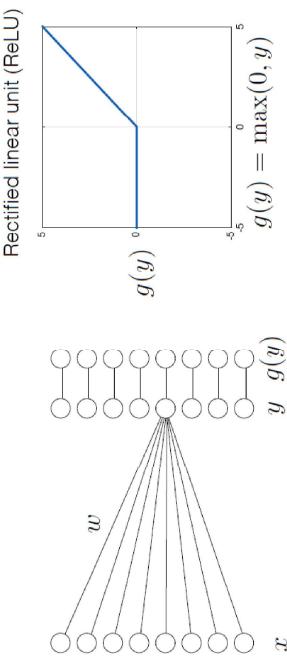


Convolution Layer Fully connected layer



First, the layer performs several convolutions to produce a set of linear activations

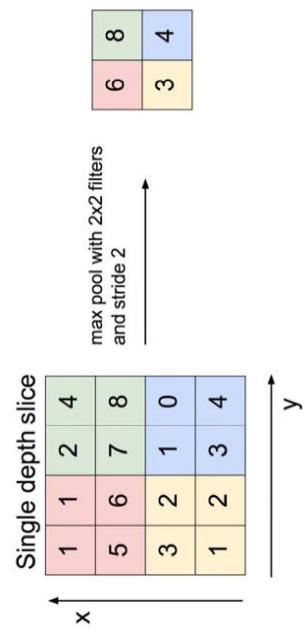
Second, each linear activation is run through a nonlinear activation function



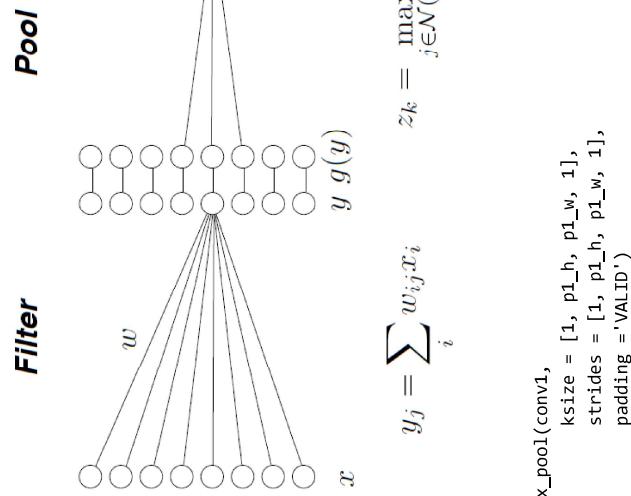
```
conv1 = tf.nn.relu(tf.add(conv1, biases['conv1']))
```

Third, use a pooling function to modify the output of the layer further

- Compute a maximum value in a sliding window (max pooling)



- Pooling size: 2×2
- Max pooling introduces invariances

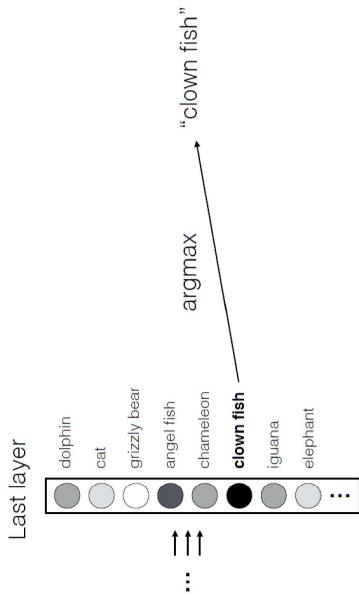


$$y_j = \sum_i w_{ij} x_i \quad z_k = \max_{j \in \mathcal{N}(j)} g(y_j)$$

```
maxp1 = tf.nn.max_pool(conv1,
                       ksize = [1, p1_h, p1_w, 1],
                       strides = [1, p1_h, p1_w, 1],
                       padding = 'VALID')
```

Fully connected layer

- Input is typically flattened features



```
In [10]: input_h = 28 # Input height
input_w = 28 # Input width
input_ch = 1 # Input channel : Gray scale
# (None, 28, 28, 1)

## First convolution layer
# Filter size
k1_h = 3
k1_w = 3
# the number of channels
k1_ch = 32
# Pooling size
p1_h = 2
p1_w = 2
# (None, 14, 14 ,32)

## Second convolution Layer
# Filter size
k2_h = 3
k2_w = 3
# the number of channels
k2_ch = 64
# Pooling size
p2_h = 2
p2_w = 2
# (None, 7, 7 ,64)

## Fully connected
# Flatten the features
# -> (None, 7*7*64)
conv_result_size = int((28/(2*2)) * (28/(2*2)) * k2_ch)
n_hidden1 = 100
n_output = 10
```

3.4. Define a CNN Shape

3.5. Define Weights, Biases and Network

- Define parameters based on predefined layer size
- Initialize with normal distribution with $\mu = 0$ and $\sigma = 0.1$

```
In [11]: weights = {
    'conv1' : tf.Variable(tf.random_normal([k1_h, k1_w, input_ch, k1_ch]),stddev = 0.1),
    'conv2' : tf.Variable(tf.random_normal([k2_h, k2_w, k1_ch, k2_ch]),stddev = 0.1),
    'hidden1' : tf.Variable(tf.random_normal([n_hidden1], stddev = 0.1)),
    'output' : tf.Variable(tf.random_normal([n_output], stddev = 0.1))
}

biases = {
    'conv1' : tf.Variable(tf.random_normal([k1_ch], stddev = 0.1)),
    'conv2' : tf.Variable(tf.random_normal([k2_ch], stddev = 0.1)),
    'hidden1' : tf.Variable(tf.random_normal([n_hidden1], stddev = 0.1)),
    'output' : tf.Variable(tf.random_normal([n_output], stddev = 0.1))
}

x = tf.placeholder(tf.float32, [None, input_h, input_w, input_ch])
y = tf.placeholder(tf.float32, [None, n_output])
```

```
In [12]: # Define Network
def net(x, weights, biases):
    ## First convolution Layer
    conv1 = tf.nn.conv2d(x, weights['conv1'],
                        strides= [1, 1, 1, 1],
                        padding = 'SAME')
    conv1 = tf.nn.relu(tf.add(conv1, biases['conv1']))
    maxp1 = tf.nn.max_pool(conv1,
                           ksize = [1, p1_h, p1_w, 1],
                           strides = [1, p1_h, p1_w, 1],
                           padding = 'VALID')

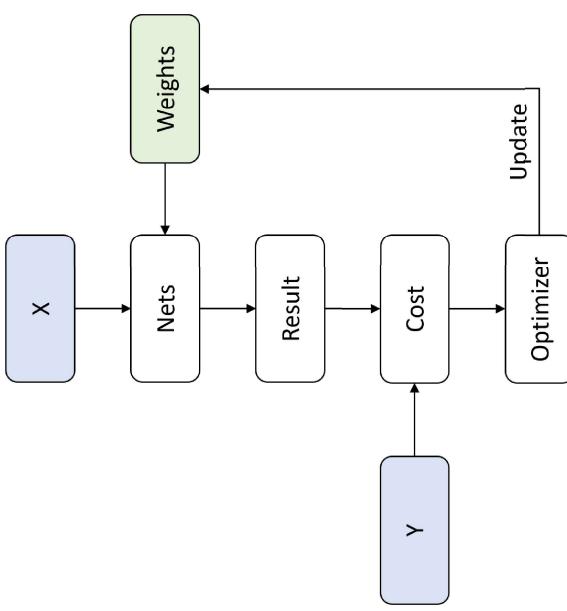
    ## Second convolution Layer
    conv2 = tf.nn.conv2d(maxp1, weights['conv2'],
                        strides= [1, 1, 1, 1],
                        padding = 'SAME')
    conv2 = tf.nn.relu(tf.add(conv2, biases['conv2']))
    maxp2 = tf.nn.max_pool(conv2,
                           ksize = [1, p2_h, p2_w, 1],
                           strides = [1, p2_h, p2_w, 1],
                           padding = 'VALID')

    # shape = conv2.get_shape().as_list()
    # maxp2_re = tf.reshape(conv2, [-1, shape[1]*shape[2]*shape[3]])
    maxp2_re = tf.reshape(maxp2, [-1, conv_result_size])

    ### Fully connected
    hidden1 = tf.add(tf.matmul(maxp2_re, weights['hidden1']), biases['hidden1'])
    hidden1 = tf.nn.relu(hidden1)
    output = tf.add(tf.matmul(hidden1, weights['output']), biases['output'])

return output
```

3.7. Summary of Model



In [13]: LR = 0.0001

```

pred = net(x, weights, biases)
loss = tf.nn.softmax_cross_entropy_with_logits(labels=y, logits=pred)
loss = tf.reduce_mean(loss)

# optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
optm = tf.train.AdamOptimizer(LR).minimize(loss)

init = tf.global_variables_initializer()
  
```

3.6. Define Loss, Initializer and Optimizer

Loss

- Classification: Cross entropy
 - Equivalent to apply logistic regression
$$-\frac{1}{N} \sum_{i=1}^N y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))$$

Initializer

- Initialize all the empty variables

Optimizer

- GradientDescentOptimizer
- AdamOptimizer: the most popular optimizer
 - n_batch : batch size for stochastic gradient descent
 - n_iter : the number of training steps
 - n_prt : check loss for every n_prt iteration

3.8. Define Configuration

- Define parameters for training CNN
 - n_batch : batch size for stochastic gradient descent
 - n_iter : the number of training steps
 - n_prt : check loss for every n_prt iteration

In [14]: n_batch = 50
n_iter = 2500
n_prt = 250

3.9. Optimization

```
In [17]: test_x, test_y = mnist.test.next_batch(1)
logits = sess.run(tf.nn.softmax(pred), feed_dict={x : test_x.reshape(-1, 28, 28, 1)})

# Run initialize
# config = tf.ConfigProto(allow_soft_placement=True) # GPU Allocating policy
# sess = tf.Session(config=config)
sess = tf.Session()
sess.run(init)

# Training cycle
for epoch in range(n_iter):
    train_x, train_y = mnist.train.next_batch(n_batch)
    train_x = np.reshape(train_x, [-1, input_h, input_w, input_ch])
    sess.run(optm, feed_dict={x: train_x, y: train_y})

    if epoch % n_prnt == 0:
        c = sess.run(loss, feed_dict={x: train_x, y: train_y})
        print("Iter : {} ".format(epoch))
        print("Cost : {}".format(c))

Iter : 0
Cost : 2.6954658031463623
Iter : 250
Cost : 0.5047059655189514
Iter : 500
Cost : 0.2169661968946457
Iter : 750
Cost : 0.2717432677745819
Iter : 1000
Cost : 0.1554456651210785
Iter : 1250
Cost : 0.28649540424346924
Iter : 1500
Cost : 0.18961450457572937
Iter : 1750
Cost : 0.09369628131389618
Iter : 2000
Cost : 0.13712839782238007
Iter : 2250
Cost : 0.018157735466957092
```

```
# Run initialize
# config = tf.ConfigProto(allow_soft_placement=True) # GPU Allocating policy
# sess = tf.Session(config=config)
sess = tf.Session()
sess.run(init)

# Training cycle
for epoch in range(n_iter):
    train_x, train_y = mnist.train.next_batch(n_batch)
    train_x = np.reshape(train_x, [-1, input_h, input_w, input_ch])
    sess.run(optm, feed_dict={x: train_x, y: train_y})

    if epoch % n_prnt == 0:
        c = sess.run(loss, feed_dict={x: train_x, y: train_y})
        print("Iter : {} ".format(epoch))
        print("Cost : {}".format(c))
```

Iter : 0

Cost : 2.6954658031463623

Iter : 250

Cost : 0.5047059655189514

Iter : 500

Cost : 0.2169661968946457

Iter : 750

Cost : 0.2717432677745819

Iter : 1000

Cost : 0.1554456651210785

Iter : 1250

Cost : 0.28649540424346924

Iter : 1500

Cost : 0.18961450457572937

Iter : 1750

Cost : 0.09369628131389618

Iter : 2000

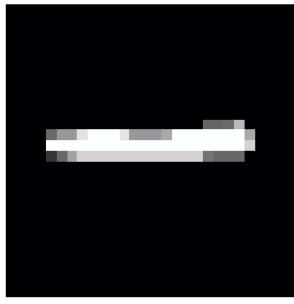
Cost : 0.13712839782238007

Iter : 2250

Cost : 0.018157735466957092

```
plt.imshow(test_x.reshape(28, 28), 'gray')

print('Prediction : {}'.format(predict))
np.set_printoptions(precision=2, suppress=True)
print('Probability : {}'.format(logits.ravel()))
```



```
Prediction : 1
Probability : [ 0.  1.  0.  0.  0.  0.  0.]
```

4. Deep Learning of Things

- CNN implemented in an Embedded System

```
In [18]: %%html
<center><iframe src="https://www.youtube.com/embed/baPLXhjsL8"
width="560" height="315" frameborder="0" allowfullscreen></iframe></center>
```

[iSystems] CNN in Raspberry Pi



In [16]: test_x, test_y = mnist.test.next_batch(100)

```
my_pred = sess.run(pred, feed_dict={x : test_x.reshape(-1, 28, 28, 1)})
my_pred = np.argmax(my_pred, axis=1)

labels = np.argmax(test_y, axis=1)

accr = np.mean(np.equal(my_pred, labels))
print('Accuracy : {}%'.format(accr*100))

Accuracy : 96.0%
```

3.10. Test