Convolutional Neural Network with TensorFlow

In this notebook, we will build a convolutional neural network from scratch using TensorFlow.

The structure of the network is: CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN -> FULLYCONNECTED -> SOFTMAX.

The training dataset is split into **mini batches**, which seeks to find a balance between stochastic gradient descent and batch gradient descent.

Xavier **initialization** method is used.

Two **optimization** algorithms are applied: gradient descent and Adam.

The constructed model is then applied to "Sign Language" project as an illustration.

In [1]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline

import warnings
warnings.filterwarnings("ignore")
import math
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.python.framework import ops
import h5py

plt.rcParams['figure.figsize'] = (6.0, 8.0)
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray' # set colormap
```

Part I: Build Convolutional Neural Network

1. Introduction

We are going to build a convolutional neural network using tensorflow.

Note:

- Tensorflow is a programming framework used in deep learning
- The two main object classes in tensorflow are Tensors and Operators.
- When we code in tensorflow we have to take the following steps:
  - Create a graph containing Tensors (Variables, Placeholders ...) and Operations (tf.matmul, tf.add, ...)
  - Create a session
  - Initialize the session
  - Run the session to execute the graph
- We can execute the graph multiple times
- The backpropagation and optimization is automatically done when running the session on the "optimizer" object.

2. Outline of the ConvNet

Notation:

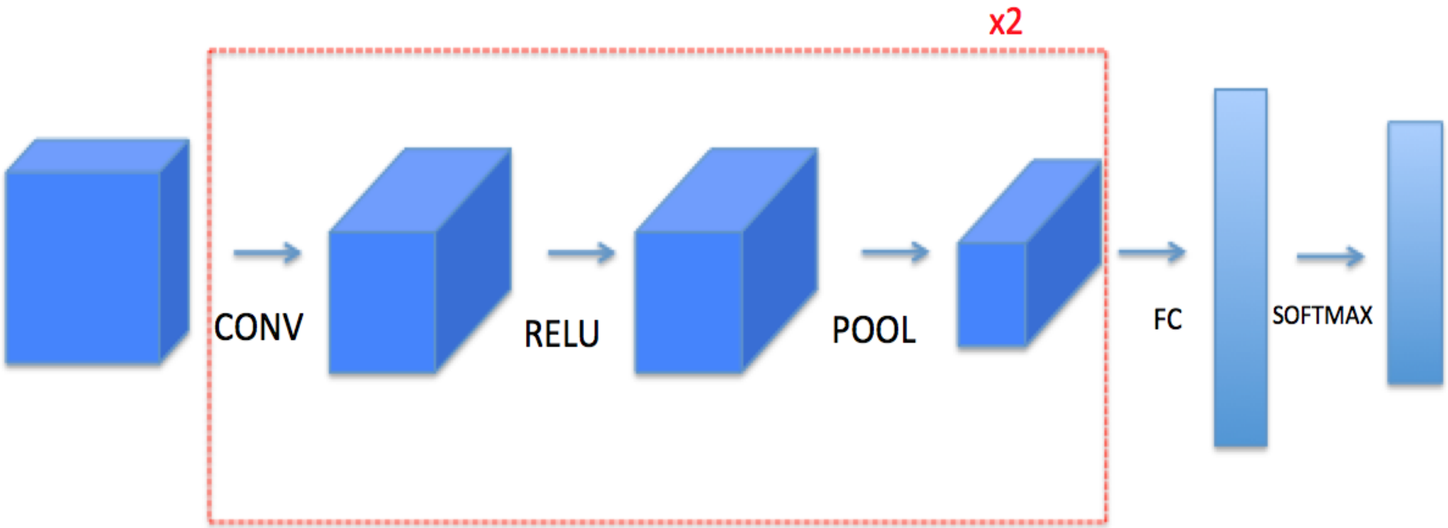
- Superscript $[l]$ denotes an object of the l^{th} layer.
  - Example: $a^{[4]}$ is the 4^{th} layer activation. $W^{[5]}$ and $b^{[5]}$ are the 5^{th} layer parameters.

- Superscript (i) denotes an object from the i^{th} example.
  - Example: $x^{(i)}$ is the i^{th} training example input.

- Lowerscript i denotes the i^{th} entry of a vector.
  - Example: $a^{[l]}_i$ denotes the i^{th} entry of the activations in layer l, assuming this is a fully connected (FC) layer.

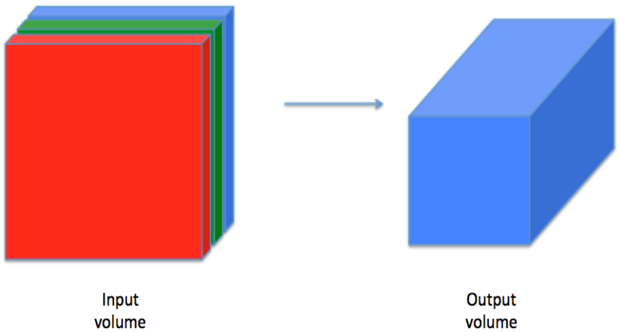
- n_H, n_W and n_C denote respectively the height, width and number of channels of a given layer. If you want to reference a specific layer l, you can also write $n_H^{[l]}$, $n_W^{[l]}$, $n_C^{[l]}$.
- $n_{H_{prev}}$, $n_{W_{prev}}$ and $n_{C_{prev}}$ denote respectively the height, width and number of channels of the previous layer. If referencing a specific layer l, this could also be denoted $n_H^{[l-1]}$, $n_W^{[l-1]}$, $n_C^{[l-1]}$.

We will use the TensorFlow to build the following model:



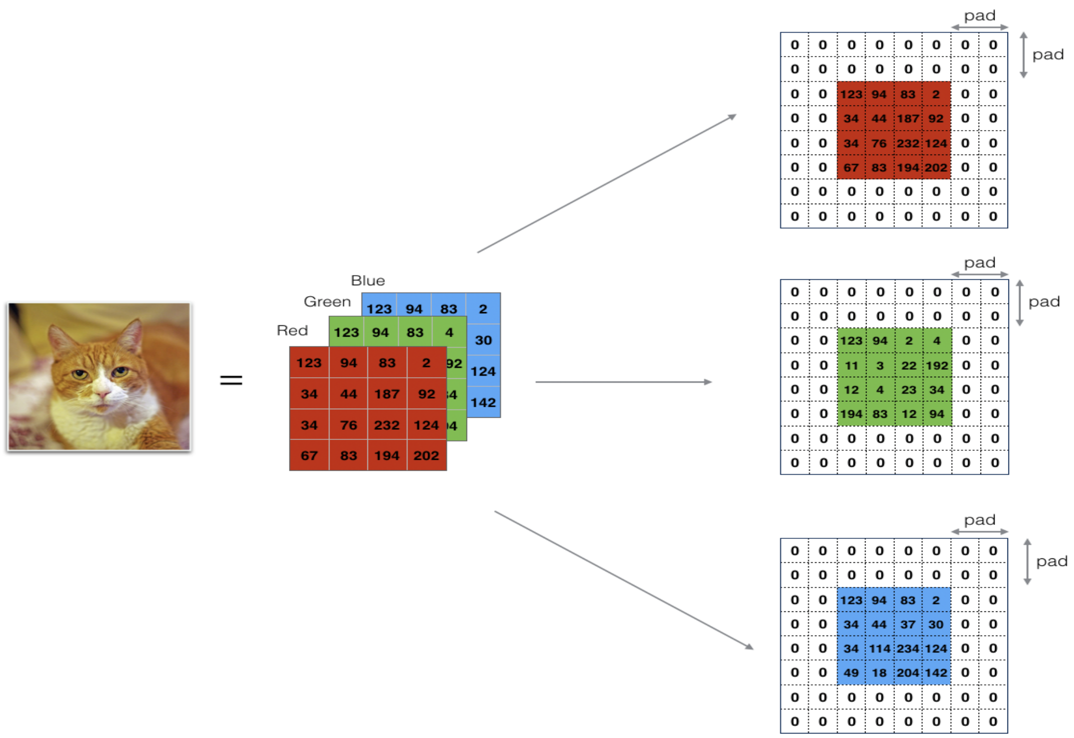
2.1 Convolutional Layer

A convolution layer transforms an input volume into an output volume of different size, as shown below.



Zero-Padding

Zero-padding adds zeros around the border of an image, as shown in the picture below with a padding of 2:



The main benefits of padding are the following:

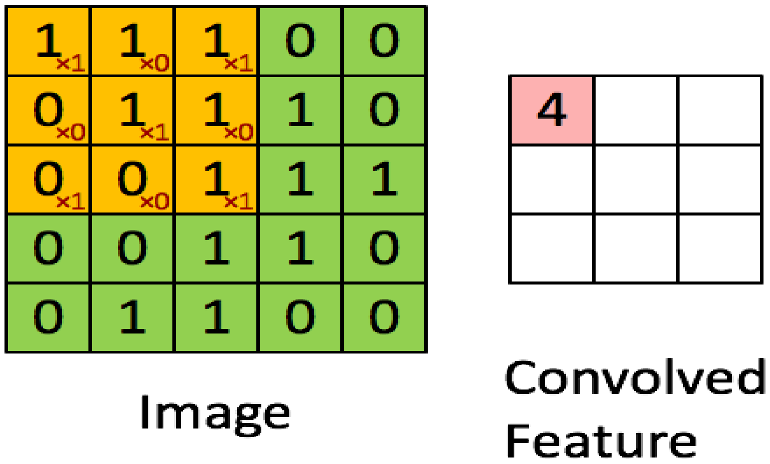
- It allows us to use a CONV layer without necessarily shrinking the height and width of the volumes. This is important for building deeper networks, since otherwise the height/width would shrink as we go to deeper layers. An important special case is the "same" convolution, in which the height/width is exactly preserved after one layer.
- It helps us keep more of the information at the border of an image. Without padding, very few values at the next layer would be affected by pixels at the edges of an image.

Convolution

To build a convolutional unit, we do the following:

- Takes an input volume
- Applies a filter at every position of the input
- Outputs another volume (usually of different size)

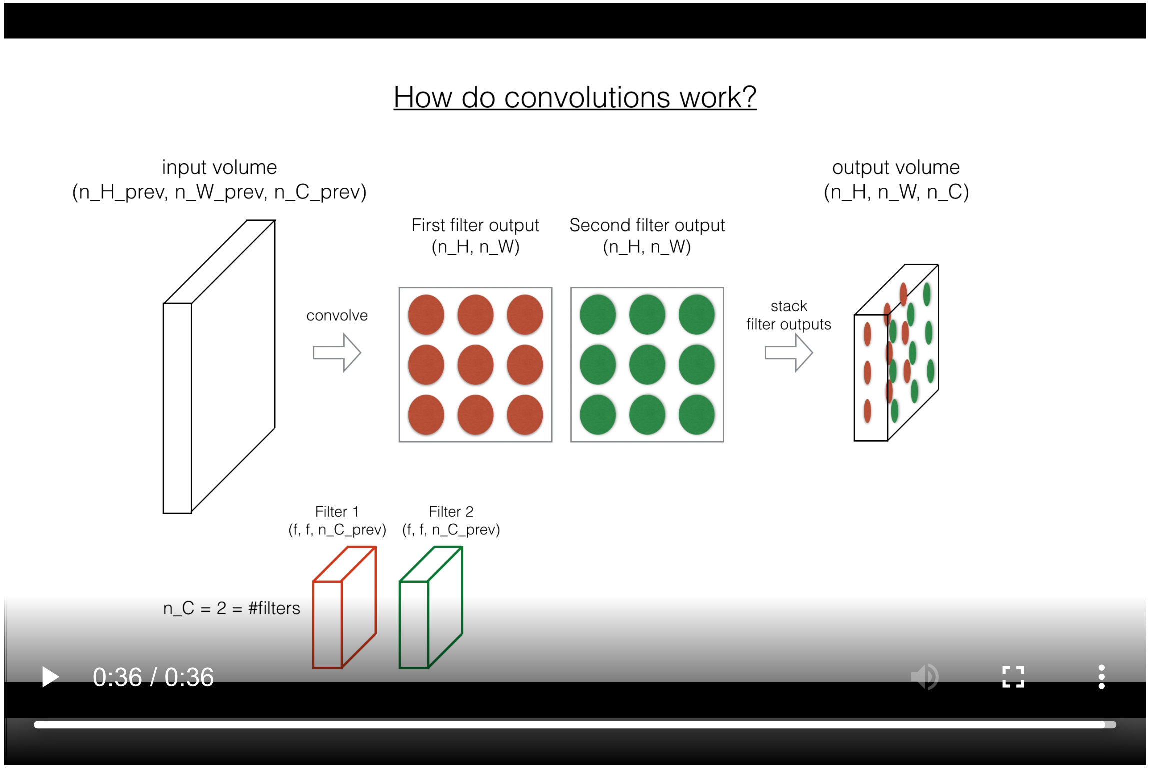
The example below has a filter of 3x3 and a stride of 1 (stride = amount we move the window each time we slide)



In a computer vision application, each value in the matrix on the left corresponds to a single pixel value, and we convolve a 3x3 filter with the image by multiplying its values element-wise with the original matrix, then summing them up and adding a bias.

Forward Pass

In the forward pass, we will take many filters and convolve them on the input. Each 'convolution' gives us a 2D matrix output. We will then stack these outputs to get a 3D volume:



Reminder: The formulas relating the output shape of the convolution to the input shape is:

$$n_H = \lfloor \frac{n_{H_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$

$$n_W = \lfloor \frac{n_{W_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$

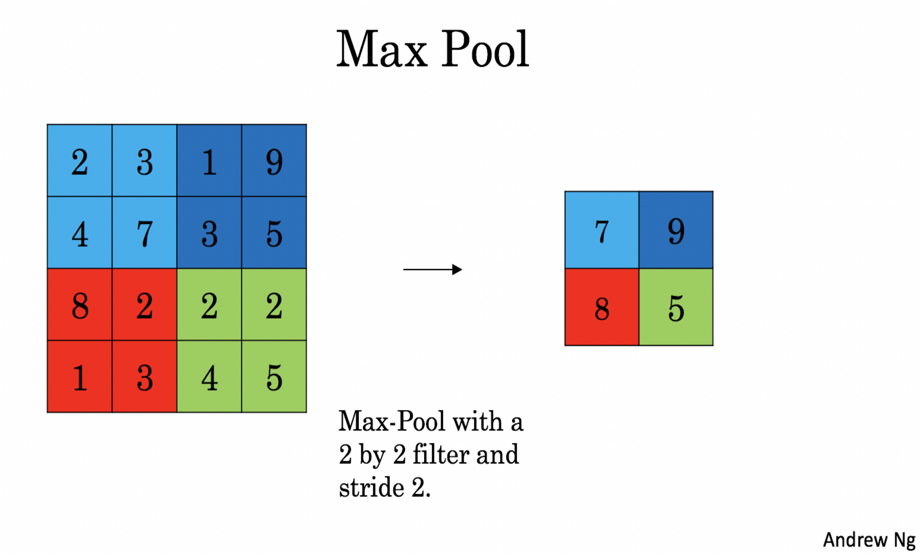
$$n_C = \text{number of filters used in the convolution}$$

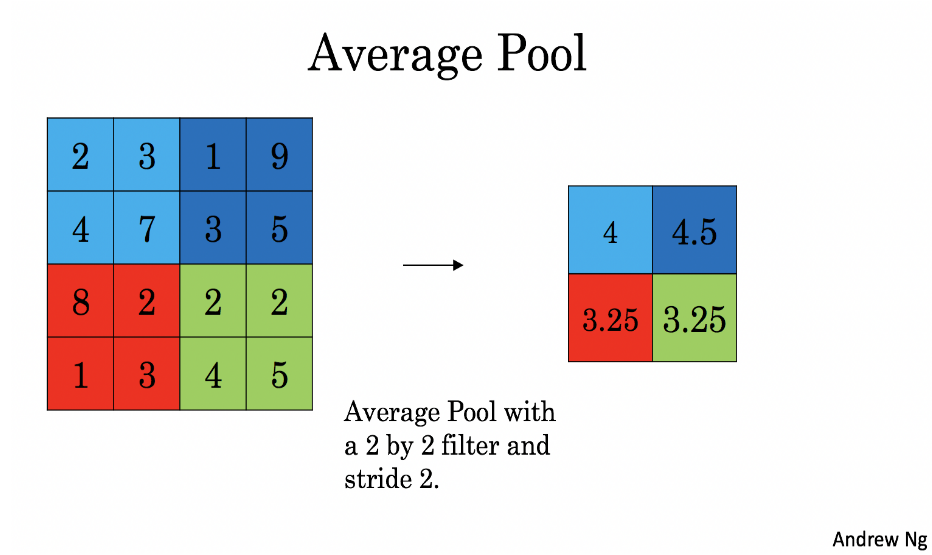
in which $\lfloor \rfloor$ denotes floor function.

2.2 Pooling Layer

The pooling (POOL) layer reduces the height and width of the input. It helps reduce computation, as well as helps make feature detectors more invariant to its position in the input. The two types of pooling layers are:

- Max-pooling layer: slides an (f, f) window over the input and stores the max value of the window in the output.
- Average-pooling layer: slides an (f, f) window over the input and stores the average value of the window in the output.





These pooling layers have no parameters for backpropagation to train. However, they have hyperparameters such as the window size f. This specifies the height and width of the fxf window we would compute a max or average over.

Reminder: As there's no padding, the formulas binding the output shape of the pooling to the input shape is:

$$n_H = \lfloor \frac{n_{H_{prev}} - f}{stride} \rfloor + 1$$
$$n_W = \lfloor \frac{n_{W_{prev}} - f}{stride} \rfloor + 1$$
$$n_C = n_{C_{prev}}$$

in which $\lfloor \rfloor$ denotes floor function.

3. Build Model with TensorFlow

3.1 Create Placeholders for Data

TensorFlow requires that we create placeholders for the input data that will be fed into the model when running the session.

Note that we do not define the number of training examples for the moment. To do so, we use "None" as the batch size, it will give us the flexibility to choose it later.

In [2]:
```python
def create_data_placeholders(nh_x, nw_x, nc_x, n_y):
    """
    Create placeholders for the data

    Arguments:
    nh_x: scalar, height of the input image
    nw_x: scalar, width of the input image
    nc_x: scalar, number of channels of the input image
    n_y: scalar, number of the output classes

    Returns:
    X: placeholder for the input data, with the shape [None, nh_x, nw_x, nc_x] and dtype 'float'
    Y: placeholder for the input labels, with the shape [None, n_y] and dtype 'float'

    Note that we use None in the shape because it lets us be flexible on the number of examples.
    """

    X = tf.placeholder(tf.float32, shape = [None, nh_x, nw_x, nc_x])
    Y = tf.placeholder(tf.float32, shape = [None, n_y])

    return X, Y
```

3.2 Split Data into Mini Batches

In [3]:
```python
# create mini batches randomly, each with the size of mini_batch_size.

def random_mini_batches_conv(X, Y, mini_batch_size, seed = 0):
    """
    Argument:
    X: input features, with dimension of [number of examples, nh_x, nw_x, nc_x]
    Y: true labels, the actual y values, with the dimension of (number of examples, number of classes)
    mini_batch_size: size of each mini batch.
    seed: random seed

    Returns:
    mini_batches: list of (mini_batch_X, mini_batch_Y)
    """

    np.random.seed(seed)

    m = X.shape[0] # total number of examples
    mini_batches = []

    # batch gradient descent
    if mini_batch_size == m:
        mini_batch = (X, Y)
        mini_batches.append(mini_batch)
        return mini_batches

    # Step 1: shuffle
    permutation_indices = list(np.random.permutation(m))
    shuffled_X = X[permutation_indices, :, :, :]
    shuffled_Y = Y[permutation_indices, :]

    # Step 2: partition
    num_batch_except_last = math.floor(m / mini_batch_size)
    for i in range(num_batch_except_last):
        mini_batch_X = shuffled_X[i * mini_batch_size:(i + 1) * mini_batch_size - 1, :, :, :]
        mini_batch_Y = shuffled_Y[i * mini_batch_size:(i + 1) * mini_batch_size - 1, :]
        mini_batches.append((mini_batch_X, mini_batch_Y))
    # last mini batch
    if m % mini_batch_size != 0:
        mini_batch_X = shuffled_X[num_batch_except_last * mini_batch_size:, :, :, :]
        mini_batch_Y = shuffled_Y[num_batch_except_last * mini_batch_size:, :]
        mini_batches.append((mini_batch_X, mini_batch_Y))

    return mini_batches
```

3.3 Initialize the Parameters

Initialize the parameters for the conv layers, i.e., the weights/filters and bias vectors b.

Note also that we will only initialize the weights/filters for the conv2d functions. TensorFlow initializes the layers for the fully connected part automatically.

We use Xavier Initialization for weights and Zero Initialization for biases.

In [4]:
```
# initialize parameters: weights/filters and biases for each convolutional layer

def init_params(conv2D_filter_dims, initialization, seed = 1):
    """
    Arguments:
    conv2D_filter_dims: python array, conv2D_filter_dims[l] is the dimension of the lth filter,
                with the dimension [f_height, f_width, nc_prev, nc]
    initialization: choose which initialization to use:
                "xavier": Xavier initialization
    seed: random seed.

    Returns:
    params: a dictionary of tensors containing weight/filter matrices wl and biases bl for the lth filter,
            params['W1'], params['W2'], ..., params['Wl'], ..., and params['b1'], params['b2'], ..., params['bl'], ...
            Wl has the dimension of [f_height, f_width, nc_prev, nc]
            bl is scalar

    Use random initialization for the weight matrices, and use zeros initialization for the biases.
    """

    params = {}
    L = len(conv2D_filter_dims) # total number of filters/convolutional layers.

    tf.set_random_seed(seed)

    for l in range(L):
        if initialization == "xavier":
            params['W' + str(l + 1)] = tf.get_variable('W' + str(l + 1), shape = conv2D_filter_dims[l],
                                        initializer = tf.contrib.layers.xavier_initializer(seed = seed))
            params['b' + str(l + 1)] = tf.get_variable('b' + str(l + 1), shape = (1, 1),
                                        initializer = tf.zeros_initializer())

    return params
```

3.4 Forward Propagation

In TensorFlow, there are built-in functions that carry out the convolution steps.

- **tf.nn.conv2d(X,W1, strides = [1,s,s,1], padding = 'SAME'):** given an input X and a group of filters $W1$, this function convolves $W1$'s filters on X. The third input ([1,s,s,1]) represents the strides for each dimension of the input (m, n_H_prev, n_W_prev, n_C_prev). You can read the full documentation here (https://www.tensorflow.org/api_docs/python/tf/nn/conv2d)
- **tf.nn.max_pool(A, ksize = [1,f,f,1], strides = [1,s,s,1], padding = 'SAME'):** given an input A, this function uses a window of size (f, f) and strides of size (s, s) to carry out max pooling over each window. You can read the full documentation here (https://www.tensorflow.org/api_docs/python/tf/nn/max_pool)
- **tf.nn.relu(Z1):** computes the elementwise ReLU of $Z1$ (which can be any shape). You can read the full documentation here. (https://www.tensorflow.org/api_docs/python/tf/nn/relu)
- **tf.contrib.layers.flatten(P)**: given an input P, this function flattens each example into a 1D vector while maintaining the batch-size. It returns a flattened tensor with shape [batch_size, k]. You can read the full documentation here. (https://www.tensorflow.org/api_docs/python/tf/contrib/layers/flatten)
- **tf.contrib.layers.fully_connected(F, num_outputs):** given a flattened input F, it returns the output computed using a fully connected layer. You can read the full documentation here. (https://www.tensorflow.org/api_docs/python/tf/contrib/layers/fully_connected)

In the last function above (`tf.contrib.layers.fully_connected`), the fully connected layer automatically initializes weights in the graph and keeps on training them as we train the model. Hence, we did not need to initialize those weights when initializing the parameters.

We build the following model: `CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN -> FULLYCONNECTED` .

In detail, we will use the following parameters for all the steps:

- Conv2D: stride 1, padding is "SAME"
- ReLU
- Max pool: Use an 8 by 8 filter size and an 8 by 8 stride, padding is "SAME"
- Conv2D: stride 1, padding is "SAME"
- ReLU
- Max pool: Use a 4 by 4 filter size and a 4 by 4 stride, padding is "SAME"
- Flatten the previous output.
- FULLYCONNECTED (FC) layer: Apply a fully connected layer without an non-linear activation function. Do not call the softmax here. This will result in 6 neurons in the output layer, which then get passed later to a softmax. In TensorFlow, the softmax and cost function are lumped together into a single function, which we'll call in a different function when computing the cost.

You can find the explanation of **"SAME"** and **"VALID"** padding here (https://stackoverflow.com/questions/37674306/what-is-the-difference-between-same-and-valid-padding-in-tf-nn-max-pool-of-t).

In [5]:

```
# entire forward propagation of convolutional neural network.
# The output is z[L] rather than a[L]

def conv_forward(X, params):
    """
    Arguments:
    X: input features placeholder, with dimension of [None, nh_x, nw_x, nc_x]
    params: a dictionary of tensors containing weight/filter matrices wl and biases bl for the lth filter,
            params['W1'], params['W2'], ..., params['Wl'], ..., and params['b1'], params['b2'], ..., params['bl'], ...
            Wl has the dimension of [f_height, f_width, nc_prev, nc]
            bl is scalar

    Returns:
    Zout: the linear output of the last layer, with the shape (batch_size, number of classes)
    """

    # Retrieve the parameters for the filters
    W1 = params['W1']
    b1 = params['b1']
    W2 = params['W2']
    b2 = params['b2']

    # Conv2D, stride 1, 'same' padding
    Z1 = tf.nn.conv2d(X, W1, strides = [1, 1, 1, 1], padding = 'SAME')
    # ReLU
    A1 = tf.nn.relu(tf.add(Z1, b1))
    # Max Pool, 8 x 8 filter, stride 8, 'same' padding
    P1 = tf.nn.max_pool(A1, ksize = [1, 8, 8, 1], strides = [1, 8, 8, 1], padding = 'SAME')
    # Conv2D, stride 1, 'same' padding
    Z2 = tf.nn.conv2d(P1, W2, strides = [1, 1, 1, 1], padding = 'SAME')
    # ReLU
    A2 = tf.nn.relu(tf.add(Z2, b2))
    # Max Pool, 4 x 4 filter, stride 4, 'same' padding
    P2 = tf.nn.max_pool(A2, ksize = [1, 4, 4, 1], strides = [1, 4, 4, 1], padding = 'SAME')
    # Flatten
    F = tf.contrib.layers.flatten(P2)
    # Fully Connected (FC) layer, without nonlinear activation function (no softmax)
    # 6 neurons in the ouput layer.
    Zout = tf.contrib.layers.fully_connected(F, num_outputs = 6, activation_fn = None)

    return Zout
```

3.5 Calculate Cost

Compute the cross entropy cost.

Note:

- It is important to know that the " `logits` " and " `labels` " inputs of `tf.nn.softmax_cross_entropy_with_logits_v2` are expected to be of shape (number of examples, number of classes).
- The returned shape of `tf.nn.softmax_cross_entropy_with_logits_v2` is the same as labels except that it does not have the last dimension of labels. `tf.reduce_mean` basically does the mean over the examples.

In [6]:
```
# calculate the cross-entropy cost function

def cost_func(Zout, Y):
    """
    Arguments:
    Zout: the linear output of the last layer, with the shape (batch_size, number of classes)
    Y: true labels placeholder, the actual y values, with the dimension of (batch_size, number of classes)

    Returns:
    cost: the cross-entropy cost tensor
    """

    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(labels = Y, logits = Zout))

    return cost
```

3.6 Backward Propagation

This is where we become grateful to programming frameworks. All the backpropagation and the parameters update is taken care of in 1 line of code. It is very easy to incorporate this line in the model.

After we compute the cost function, we create an " `optimizer` " object. We have to call this object along with the cost when running the tf.session. When called, it will perform an optimization on the given cost with the chosen method and learning rate.

For instance, for gradient descent the optimizer would be:

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate = learning_rate).minimize(cost)
```

To make the optimization we would do:

```
_ , c = sess.run([optimizer, cost], feed_dict={X: minibatch_X, Y: minibatch_Y})
```

This computes the backpropagation by passing through the tensorflow graph in the reverse order, from cost to inputs.

Note When coding, we often use `_` as a "throwaway" variable to store values that we won't need to use later. Here, `_` takes on the evaluated value of `optimizer` , which we don't need (and `c` takes the value of the `cost` variable).

3.7 Complete Model

In [7]:

```python
# Build the entire convolutional neural network model

def conv_model(train_X, train_Y, test_X, test_Y, conv2D_filter_dims, params_seed = 0, mini_batch_size = None,
                   optimizer = "adam", learning_rate = 0.0001, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8,
                   epochs = 1500, print_cost_freq = 0, save_cost_freq = 100, initialization = "xavier"):
    """
    Arguments:
    train_X: training set features, with the dimension of [batch_size, nh_x, nw_x, nc_x]
    train_Y: training set labels, with the dimension of [batch_size, n_y]
    test_X: training set features, with the dimension of [batch_size, nh_x, nw_x, nc_x]
    test_Y: training set labels, with the dimension of [batch_size, n_y]
    conv2D_filter_dims: python array, conv2D_filter_dims[l] is the dimension of the lth filter,
            with the dimension [f_height, f_width, nc_prev, nc]
    params_seed: random seed for initialization.
    mini_batch_size: size of each mini batch. By default it is the total number of examples, i.e., batch gradient descent
    optimizer: choose the optimization method:
           "gd": gradient descent
           "adam": Adam optimization
    learning_rate: learning rate for gradient descent
    beta1, beta2: hyperparameters for Adam optimization
    epsilon: hyperparameter preventing division by zero in Adam updates
    epochs: number of epochs
    print_cost_freq: if > 0, print the cost value every print_cost_freq epochs.
    save_cost_freq: save the cost value every save_cost_freq epochs into costs, for ploting the learning curve.
    initialization: choose which initialization to use:
            "xavier": Xavier initialization

    Returns:
    params: a dictionary of tensors containing weight/filter matrices wl and biases bl for the lth filter,
        params['W1'], params['W2'], ..., params['Wl'], ..., and params['b1'], params['b2'], ..., params['bl'], ...
        Wl has the dimension of [f_height, f_width, nc_prev, nc]
        bl is scalar
    """

    ops.reset_default_graph() # to be able to rerun the model without overwriting tf variables

    costs = []
    tf.set_random_seed(1)
    mini_batch_seed = 3 # random seed for creating mini batches
    (m, nh_x, nw_x, nc_x) = train_X.shape # number of training examples, and height, width, channel of the image
    n_y = train_Y.shape[1]
    if mini_batch_size is None:
        mini_batch_size = m
    num_mini_batches = math.ceil(m / mini_batch_size) # number of mini batches
```

```
    # create placeholders for data
    X, Y = create_data_placeholders(nh_x, nw_x, nc_x, n_y)

    # initialization
    params = init_params(conv2D_filter_dims, initialization, params_seed)

    # forward propagation:
    Zout = conv_forward(X, params)

    # calculate the cost
    cost = cost_func(Zout, Y)

    # backward propagation: define the tensorflow optimizer
    if optimizer == "gd":
        opt = tf.train.GradientDescentOptimizer(learning_rate = learning_rate).minimize(cost)
    if optimizer == "adam":
        opt = tf.train.AdamOptimizer(learning_rate = learning_rate, beta1 = beta1, beta2 = beta2,
                                     epsilon = epsilon).minimize(cost)

    # initialize all variables
    init = tf.global_variables_initializer()

    # start a session to compute the tensorflow graph
    with tf.Session() as sess:

        # run initializer
        sess.run(init)

        for i in range(1, epochs + 1):
            # define a cost related to an epoch, as the average cost over all mini batches within a single epoch
            epoch_cost = 0
            # create mini batches
            mini_batch_seed += 1
            mini_batches = random_mini_batches_conv(train_X, train_Y, mini_batch_size, mini_batch_seed)
            for mini_batch in mini_batches:
                (mini_batch_X, mini_batch_Y) = mini_batch
                # run the graph on a mini batch
                # run the session to excute the optimizer and the cost.
                _, mini_batch_cost = sess.run([opt, cost], feed_dict = {X : mini_batch_X, Y : mini_batch_Y})
                epoch_cost += mini_batch_cost / num_mini_batches
            # print and save the costs
            if print_cost_freq > 0 and (i == 1 or i % print_cost_freq == 0):
                print("current epoch: " + str(i) + ", cost: " + str(epoch_cost))
            if save_cost_freq > 0 and (i == 1 or i % save_cost_freq == 0):
                costs.append(epoch_cost)
```

```
        # plot the cost
        plt.plot(costs)
        plt.ylabel('cost')
        plt.xlabel('epochs * ' + str(save_cost_freq))
        plt.title('learning rate = ' + str(learning_rate))
        plt.show()

        # save the parameters
        params = sess.run(params)
        print('Parameters have been trained!')

        # evaluate the model accuracy

        # Get the correct predictions
        # for each example (i.e., each row), the maximum Zout corresponds to the predicted class
        correct_pred = tf.equal(tf.argmax(Zout, axis = 1), tf.argmax(Y, axis = 1)) # shape is (number of examples,)

        # calculate the accuracy
        accuracy = tf.reduce_mean(tf.cast(correct_pred, "float"))

        print("Training Set Accuracy: " + str(accuracy.eval({X : train_X, Y : train_Y})))
        print("Test Set Accuracy: " + str(accuracy.eval({X : test_X, Y : test_Y})))

    return params
```

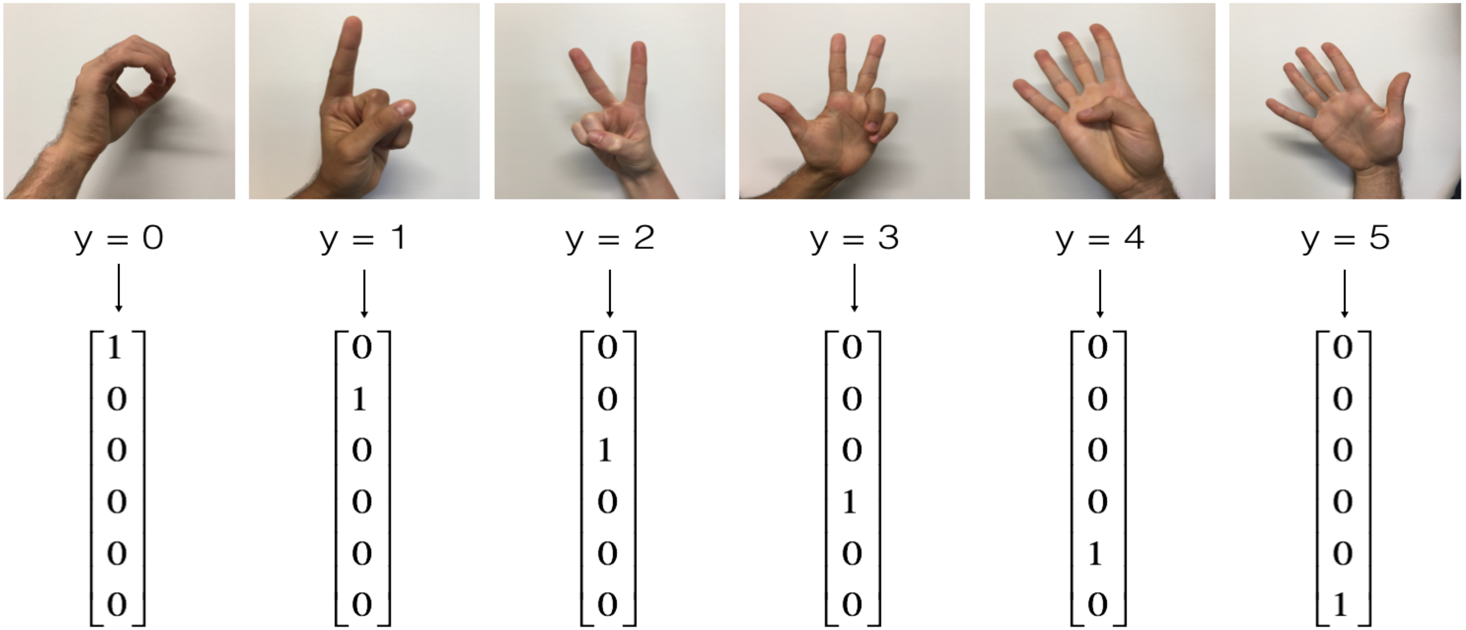
Part II: Example - Sign Language

1. Introduction

One afternoon, we decided to teach our computers to decipher sign language. We spent a few hours taking pictures in front of a white wall and came up with the following dataset：

- **Training set**: 1080 pictures (64 by 64 pixels) of signs representing numbers from 0 to 5 (180 pictures per number).
- **Test set**: 120 pictures (64 by 64 pixels) of signs representing numbers from 0 to 5 (20 pictures per number).

Here are examples for each number, and how we represent the labels. These are the original pictures, before we lowered the image resolutoion to 64 by 64 pixels.



Our goal is to build an algorithm that would facilitate communications from a speech-impaired person to someone who doesn't understand sign language.

2. Load Data

In [8]:
```python
# Load training and test data, and list of classes

def load_data():
    """
    Returns:
    train_x_orig: numpy array, original training set features
    train_y_orig: numpy array, original training set labels
    test_x_orig: numpy array, original test set features
    test_y_orig: numpy array, original test set labels
    classes: numpy array, list of classes
    """
    # training set
    train_data = h5py.File('data/train_signs.h5', 'r')
    train_x_orig = np.array(train_data['train_set_x']) # training set features
    train_y_orig = np.array(train_data['train_set_y']) # training set labels
    # test set
    test_data = h5py.File('data/test_signs.h5', 'r')
    test_x_orig = np.array(test_data['test_set_x']) # test set features
    test_y_orig = np.array(test_data['test_set_y']) # test set labels
    # list of classes
    classes = np.array(test_data['list_classes'])
    # reshape the labels, make sure the dimension is (1, number of examples)
    train_y_orig = train_y_orig.reshape((1, train_y_orig.shape[0]))
    test_y_orig = test_y_orig.reshape((1, test_y_orig.shape[0]))

    return train_x_orig, train_y_orig, test_x_orig, test_y_orig, classes
```

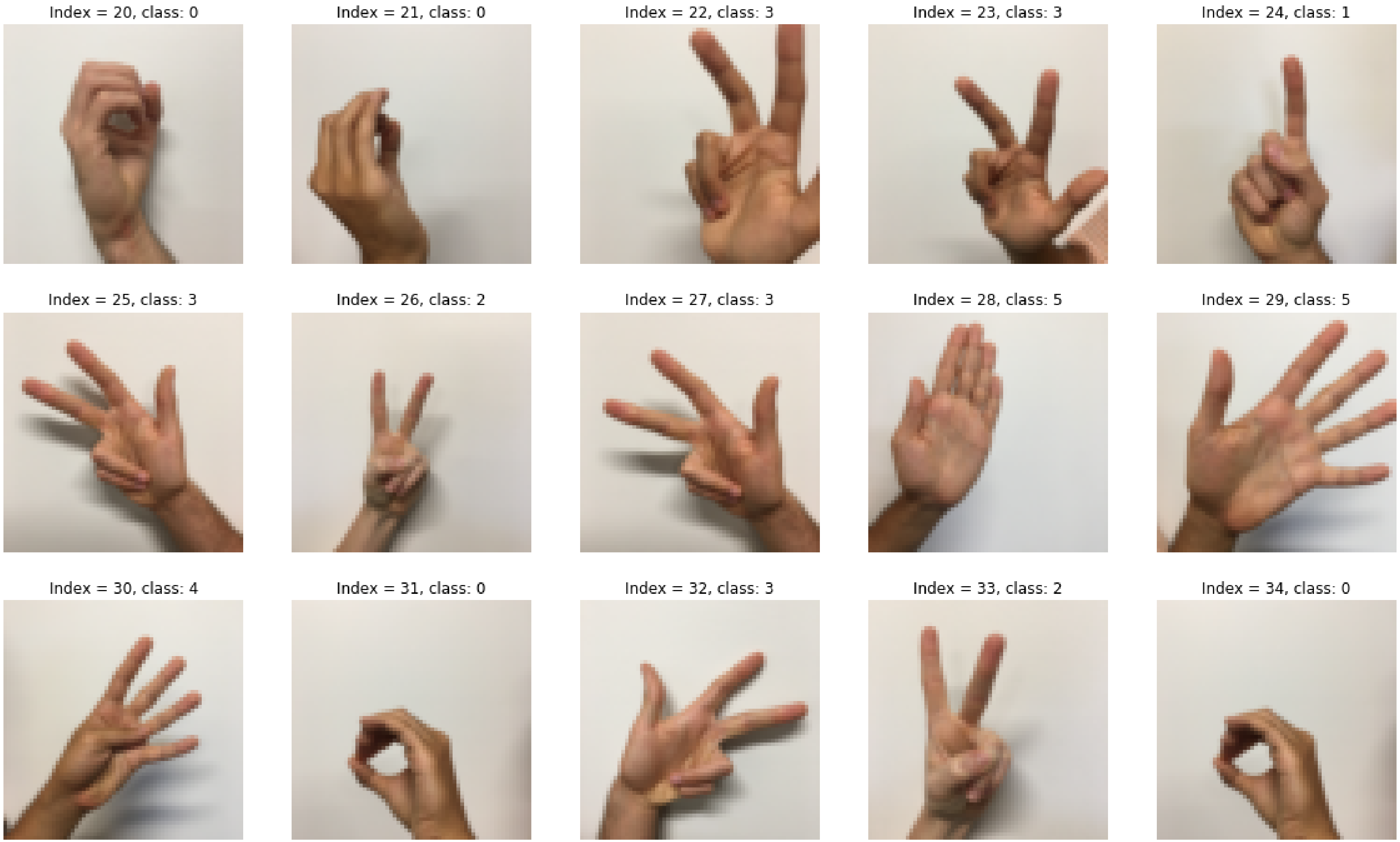
In [9]:
```
# Load the data
train_x_orig, train_y_orig, test_x_orig, test_y_orig, classes = load_data()
print("Total number of training examples: " + str(train_x_orig.shape[0]))
print("Total number of test examples: " + str(test_x_orig.shape[0]))
print("Size of each image: " + str(train_x_orig[0].shape))
print("All classes: " + str(classes))
print("train_x_orig shape: " + str(train_x_orig.shape))
print("train_y_orig shape: " + str(train_y_orig.shape))
print("test_x_orig shape: " + str(test_x_orig.shape))
print("test_y_orig shape: " + str(test_y_orig.shape))
```

```
Total number of training examples: 1080
Total number of test examples: 120
Size of each image: (64, 64, 3)
All classes: [0 1 2 3 4 5]
train_x_orig shape: (1080, 64, 64, 3)
train_y_orig shape: (1, 1080)
test_x_orig shape: (120, 64, 64, 3)
test_y_orig shape: (1, 120)
```

In [10]:
```
# show some examples of the images in the training set

def example(indices, X, Y, classes):
    """
    Arguments:
    indices: list of the indices of X to be shown
    X: image fearues, with the shape of (number of examples, num_px, num_px, 3)
    Y: image classes, with the shape of (1, number of examples)
    classes: numpy array, list of classes
    """

    num = len(indices)
    columns = 5 # the number of columns to arrange the figures
    plt.figure(figsize = (20, 12))

    for i in range(num):
        plt.subplot(math.ceil(num / columns), columns, i + 1)
        plt.imshow(X[indices[i]])
        plt.axis('off')
        plt.title("Index = " + str(indices[i]) + ", class: " + str(classes[Y[0, indices[i]]]))
```

In [11]:

```
indices = [i for i in range(20, 35, 1)]
example(indices, train_x_orig, train_y_orig, classes)
```



3. Data Pre-Processing

We flatten the image dataset, then normalize it by dividing by 255. On top of that, we convert each label to a one-hot vector.

In [12]:
```
# convert Y to one-hot vectors

def convert_to_one_hot(Y, C):
    """
    Argument:
    Y: labels, with the dimension of (1, number of examples)
    C: number of classes

    Returns:
    Y_converted: one-hot representation of Y, with the dimension of (C, number of examples)
    """
    # numpy.eye(): Return a 2-D array with ones on the diagonal and zeros elsewhere.
    # note that the ith column is exactly the one-hot representation of the ith class
    Y_converted = np.eye(C)[:, Y.reshape(-1)]

    return Y_converted
```

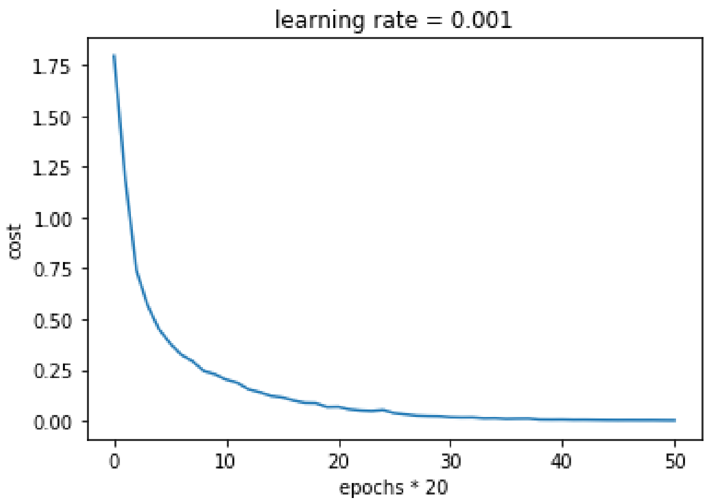
In [13]:
```
# pre-processing the features

# standardize, so the values are between 0 and 1.
train_x = train_x_orig / 255
test_x = test_x_orig / 255

# convert labels to one-hot vectors, and make the dimension (number of examples, number of classes)
train_y = convert_to_one_hot(train_y_orig, len(classes)).T
test_y = convert_to_one_hot(test_y_orig, len(classes)).T


print("dimension of train_x: " + str(train_x.shape))
print("dimension of test_x: " + str(test_x.shape))
print("dimension of train_y: " + str(train_y.shape))
print("dimension of test_y: " + str(test_y.shape))
```

```
dimension of train_x: (1080, 64, 64, 3)
dimension of test_x: (120, 64, 64, 3)
dimension of train_y: (1080, 6)
dimension of test_y: (120, 6)
```

4. Model Training and Evaluation

In [14]:
```
conv2D_filter_dims = [[4, 4, 3, 8], [2, 2, 8, 16]]

params = conv_model(train_x, train_y, test_x, test_y, conv2D_filter_dims, params_seed = 1, mini_batch_size = 64,
                    optimizer = "adam", learning_rate = 0.001, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8,
                    epochs = 1000, print_cost_freq = 200, save_cost_freq = 20, initialization = "xavier")
```

```
WARNING: Logging before flag parsing goes to stderr.
W0819 21:56:05.694703 10280 lazy_loader.py:50]
The TensorFlow contrib module will not be included in TensorFlow 2.0.
For more information, please see:
  * https://github.com/tensorflow/community/blob/master/rfcs/20180907-contrib-sunset.md
  * https://github.com/tensorflow/addons
  * https://github.com/tensorflow/io (for I/O related ops)
If you depend on functionality not listed there, please file an issue.

W0819 21:56:05.756707 10280 deprecation.py:323] From D:\Python\Anaconda3\envs\py3\lib\site-packages\tensorflow\contrib\layers\pyt
hon\layers\layers.py:1634: flatten (from tensorflow.python.layers.core) is deprecated and will be removed in a future version.
Instructions for updating:
Use keras.layers.flatten instead.

current epoch: 1, cost: 1.7945187372319837
current epoch: 200, cost: 0.20280148395720649
current epoch: 400, cost: 0.06857917435905513
current epoch: 600, cost: 0.017066438326283413
current epoch: 800, cost: 0.005564909935107125
current epoch: 1000, cost: 0.001731089230708997
```




```
Parameters have been trained!
Training Set Accuracy: 1.0
Test Set Accuracy: 0.85833335
```