CSE 574: Introduction to Machine Learning
Fall 2017

Project 4: Introduction to Deep Learning

Instructor: Dr. Sargur N. Srihari

Prasad Salvi
Person #: 50207353

Veerappan Saravanan
Person #:50247314

Sarah Mullin
Person #: 34508498

Table of Contents

1. Project Synopsis

The objective of this project is to implement a convolutional neural network to determine whether a person is wearing glasses or not. This is a binary classification problem using the CelebA dataset. We have chosen to implement the convolutional neural network using Tensorflow. In the project below, we have taken a subset of the training data and the whole training dataset, tested models with varying layers (7,9, and 11 layers), used dropout for regularization, and changed the resolution of the images for a one channel model. For a three channel model, we used an optimal architecture for classification based on the CIFAR-10 model implementation for object classification.

2. Data Partition and Feature Extraction

We extracted feature values in two ways, such that RGB color (3 channels) was retained and image reduction in the form of extracting a grayscale image (1 channel).

One Channel (Grayscale):

```
def load_data(attr_name, sz):
    print("Loading Data.................")
    data = []
    label = []
    attribute = []
    path_to_data = "./img_align_celeba/"
    img_list = os.listdir(path_to_data)
#    sz = (27, 33)
    for name in sorted(img_list):
        if '.jpg' in name:
            img = cv2.imread(path_to_data + name)
            img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            resized_img = resize_and_scale(img, sz)
            data.append(resized_img.flatten())
            #data.append(img.flatten())
    data = np.array(data)
    print("Loading Label.................")
    f = open('./list_attr_celeba.txt', 'r')
    text = f.readlines()
    text = np.array(text)
    attr2idx = dict()
    for i, attr in enumerate(text[1].split()):
        attr2idx[attr] = i
    attr_index = attr2idx[attr_name]#'Eyeglasses'
    for i in text[2:]:
        value = i.split()
        attribute.append(value[attr_index + 1]) #First index is image name
    attribute = np.array(attribute,dtype= np.float32)
    print("Converting Label.................")
    for i in range(0,len(attribute)):
        if (attribute[i] == 1):
            label.append([1, 0])
        else:
            label.append([0, 1])
    label = np.array(label)
    #usps_dataset, usps_label = reformat_tf(usps_data, usps_label)
    return data, label
```

Three Channel (Color):

```
files_path = "./img_align_celeba/"

glasses_files_path = os.path.join(files_path, '*.jpg')

glasses_files = sorted(glob(glasses_files_path))

n_files = len(glasses_files)
#print(n_files)
#size_image1 =178
#size_image2=218
size_image1=27
size_image2=33
allX = np.zeros((n_files, size_image1, size_image2, 3), dtype='float32')
ally = np.zeros(n_files)
count = 0
for f in glasses_files:
    try:
        img = io.imread(f)
        new_img = skimage.transform.resize(img, (27, 33, 3))
        allX[count] = np.array(new_img)
        ally[count] = 0
        count += 1
    except:
        continue
attribute=[]
g = open('./list_attr_celeba.txt', 'r')
text = g.readlines()
text = np.array(text)
attr2idx = dict()
for i, attr in enumerate(text[1].split()):
    attr2idx[attr] = i
attr_index = attr2idx['Eyeglasses']#'Eyeglasses'
for i in text[2:]:
    value = i.split()
    attribute.append(value[attr_index + 1]) #First index is image name
attribute = np.array(attribute,dtype= np.float32)
#print("Converting Label.................")
for i in range(0,len(attribute)):
    if (attribute[i] == 1):
        ally[i]=1
    else:
        ally[i]=0
ally = np.array(ally)
```

Data was then partitioned such that 80% was used for training, 10% validation, and 10% testing. A subset of the training set was taken such that 50% (81,039) of the training dataset was used initially to choose an optimal model (Project Instructions #3).

```
########break up data into training, validation, and test sets
train_limit = int(math.floor(0.8 * len(allX)))
validate_limit = int(math.floor(0.1*len(allX)))
#print (train_limit, validate_limit)


X = allX[0:train_limit,:,]
X_validation = allX[(train_limit+1):(train_limit+validate_limit),:,]
X_test = allX[(train_limit+validate_limit+1):,:,]

Y = ally[0:train_limit]
Y_validation = ally[(train_limit+1):(train_limit+validate_limit)]
Y_test = ally[(train_limit+validate_limit+1):]

# encode the Ys
Y = to_categorical(Y, 2)
Y_test = to_categorical(Y_test, 2)
Y_validation = to_categorical(Y_validation, 2)

#take a subset of training dataset to find parameters
x_sm = int(math.floor(0.8 * len(allX))*0.5)
print (x_sm)
X_sm = allX[0:x_sm,:,]
Y_sm = ally[0:x_sm]
Y_sm=to_categorical(Y_sm, 2)
```

Data were initially reduced to the dimension: 27 x 33.

Weights and bias parameters were randomly initiated.

3. Models: Number of Layers, Filter Sizes, and Number of Fully Connected Nodes

Model 1:

# layer 1: convolution layer with 32 filters
# layer 2: Max pooling layer
# layer 3: Convolution layer with 64 filters
# layer 4: Max pooling layer
# layer 5: Fully connected 1024 node layer
# layer 6: Dropout layer (removes neurons randomly to combat overfitting)
# layer 7: Fully connected layer with two outputs (glasses or no glasses)

Model 2:

# layer 1: convolution layer with 32 filters
# layer 2: Max pooling layer
# layer 3: Convolution layer with 32 filters
# layer 4: Max pooling layer
# layer 5: Another convolution layer with 64 filters
# layer 6: Max pooling layer
# layer 7: Fully connected 512 node layer
# layer 8: Dropout layer (removes neurons randomly to combat overfitting)
# layer 9: Fully connected layer with two outputs (glasses or no glasses

Model 3:

# layer 1: convolution layer with 32 filters

# layer 2: Max pooling layer
# layer 3: Convolution layer with 64 filters
# layer 4: Max pooling layer
# layer 5: Another convolution layer with 64 filters
# layer 6: Max pooling layer
# layer 7: Another convolution layer with 128 filters
# layer 8: Max pooling layer
# layer 9: Fully connected 1024 node layer
# layer 10: Dropout layer (removes neurons randomly to combat overfitting)
# layer 11: Fully connected layer with two outputs (glasses or no glasses)

Filter Size was chosen to be 3x3x1 (grayscale) or 3x3x3 (RGB) since we want to filter small details from the images. Stride was chosen to be 1. For pooling layers, we used max pooling with a filter size of 2x2. Our batch size was set at 50. Padding was set to "same" to ensure that the output was the same size as the input. 20,000 Epochs were run.
Weights and bias parameters were randomly initiated.

```
#Generate random weights based on shape
def generateWeigths(shape):
    weight = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(weight)

#Generate bias weights based on shape
def generateBias(shape):
    bias = tf.constant(0.1, shape=shape)
    return tf.Variable(bias)

#Apply convolution to image x with filter W and move the strides 1 pixel
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

#Max pooling
def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
```

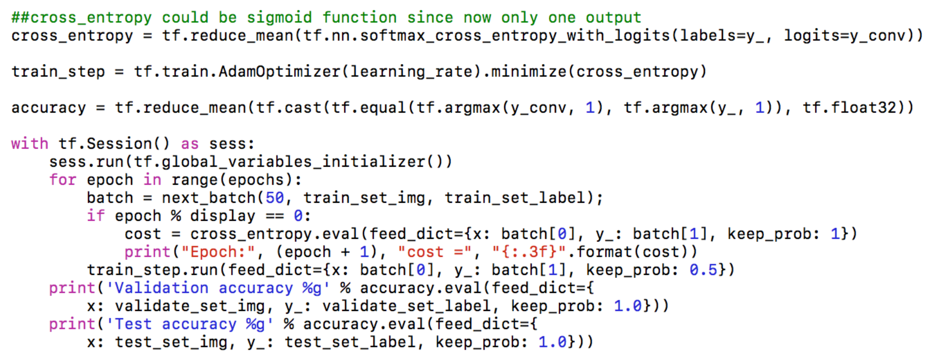
```
def flatten_layer(layer):
    # Get the shape of the input layer.
    layer_shape = layer.get_shape()

    # The shape of the input layer is assumed to be:
    # layer_shape == [num_images, img_height, img_width, num_channels]

    # The number of features is: img_height * img_width * num_channels
    # We can use a function from TensorFlow to calculate this.
    num_features = layer_shape[1:4].num_elements()

    # Reshape the layer to [num_images, num_features].
    # Note that we just set the size of the second dimension
    # to num_features and the size of the first dimension to -1
    # which means the size in that dimension is calculated
    # so the total size of the tensor is unchanged from the reshaping.
    layer_flat = tf.reshape(layer, [-1, num_features])

    # The shape of the flattened layer is now:
    # [num_images, img_height * img_width * num_channels]

    # Return both the flattened layer and the number of features.
    return layer_flat, num_features
```

Dropout regularization was applied to all intermediate layers of our model. The final dropout rate was 0.5 (see Validation Set: Tuning Hyperparameters). Below is the code for the architecture of the model (layers) including the dropout layer.

```
#Convolution Layer 1, finds 32 features for each 5x5
#Weights: patch_size, patch_size, i/p_channel, o/p_channel
w_conv1 = generateWeigths([filter_size1, filter_size1, num_channels, num_filters1])
b_conv1 = generateBias([num_filters1])

#Reshape to shape any, width, height, colour_channel
x_image = tf.reshape(x, [-1, img_shape[0], img_shape[1], num_channels])

h_conv1 = tf.nn.relu(conv2d(x_image, w_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)


#Convolution Layer 2, finds 64 features for each 5x5
#Weights: patch_size, patch_size, i/p_channel, o/p_channel
# 32 channels: 32*64 filters: Take 1,1 2,1.. 32,1 and apply to 1st pixel of each channel and sum to
    calculate on pixel. Repeat for depth 64
w_conv2 =  generateWeigths([filter_size2, filter_size2, num_filters1, num_filters2])
b_conv2 = generateBias([num_filters2])

h_conv2 = tf.nn.relu(conv2d(h_pool1, w_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

#Convolution Layer 3,
#w_conv3 =  generateWeigths([filter_size3, filter_size3, num_filters2, num_filters3])
#b_conv3 = generateBias([num_filters3])

#h_conv3 = tf.nn.relu(conv2d(h_pool2, w_conv3) + b_conv3)
#h_pool3 = max_pool_2x2(h_conv3)


layer_flat, num_features = flatten_layer(h_pool2)

W_fc1 = generateWeigths([num_features, fc_size])
b_fc1 = generateBias([fc_size])

h_fc1 = tf.nn.relu(tf.matmul(layer_flat, W_fc1) + b_fc1)

#Drop out
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

W_fc2 = generateWeigths([fc_size, num_classes])
b_fc2 = generateBias([num_classes])

y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

In addition, we used Rectified Linear Unit ($f(x)=max(0,x)$) activation function as opposed to sigmoid activation function, since it is computationally simple.

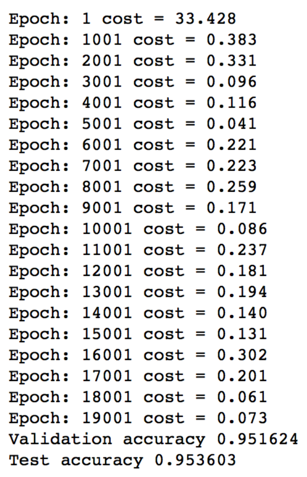
The model was then trained in the following way:



```
##cross_entropy could be sigmoid function since now only one output
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))

train_step = tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)

accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1)), tf.float32))

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch in range(epochs):
        batch = next_batch(50, train_set_img, train_set_label);
        if epoch % display == 0:
            cost = cross_entropy.eval(feed_dict={x: batch[0], y_: batch[1], keep_prob: 1})
            print("Epoch:", (epoch + 1), "cost =", "{:.3f}".format(cost))
        train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
    print('Validation accuracy %g' % accuracy.eval(feed_dict={
        x: validate_set_img, y_: validate_set_label, keep_prob: 1.0}))
    print('Test accuracy %g' % accuracy.eval(feed_dict={
        x: test_set_img, y_: test_set_label, keep_prob: 1.0}))
```

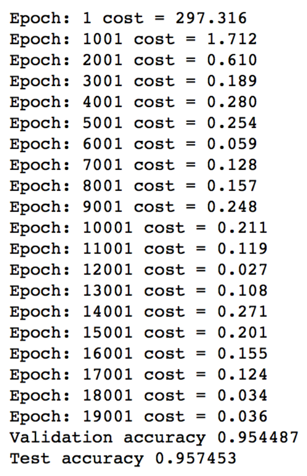
4. Validation Set: Tuning Hyperparameters

We used our validation set to tune the number of layers and filters (architectural structure). Stride, filter sizes, fully-connected nodes=1024 remained constant. By convention (like AlexNet), filter sizes were chosen to be 32, 64, and 128. Below, shows the output for Model 1, Model 2, and Model 3, respectively. Validation accuracy improved slightly for the more complicated models (Model 2 and Model 3). However, in favor of a more parsimonious model and since binary classification is not a complex problem, we have chosen to retain model 1.



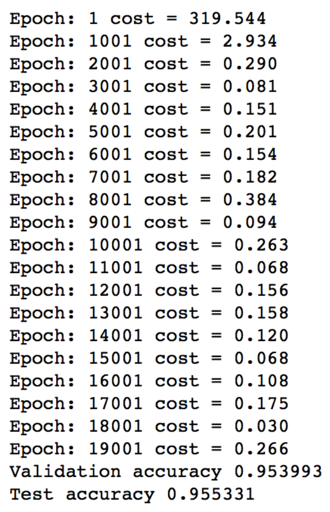
```
Epoch: 1 cost = 33.428
Epoch: 1001 cost = 0.383
Epoch: 2001 cost = 0.331
Epoch: 3001 cost = 0.096
Epoch: 4001 cost = 0.116
Epoch: 5001 cost = 0.041
Epoch: 6001 cost = 0.221
Epoch: 7001 cost = 0.223
Epoch: 8001 cost = 0.259
Epoch: 9001 cost = 0.171
Epoch: 10001 cost = 0.086
Epoch: 11001 cost = 0.237
Epoch: 12001 cost = 0.181
Epoch: 13001 cost = 0.194
Epoch: 14001 cost = 0.140
Epoch: 15001 cost = 0.131
Epoch: 16001 cost = 0.302
Epoch: 17001 cost = 0.201
Epoch: 18001 cost = 0.061
Epoch: 19001 cost = 0.073
Validation accuracy 0.951624
Test accuracy 0.953603
```

Model 1



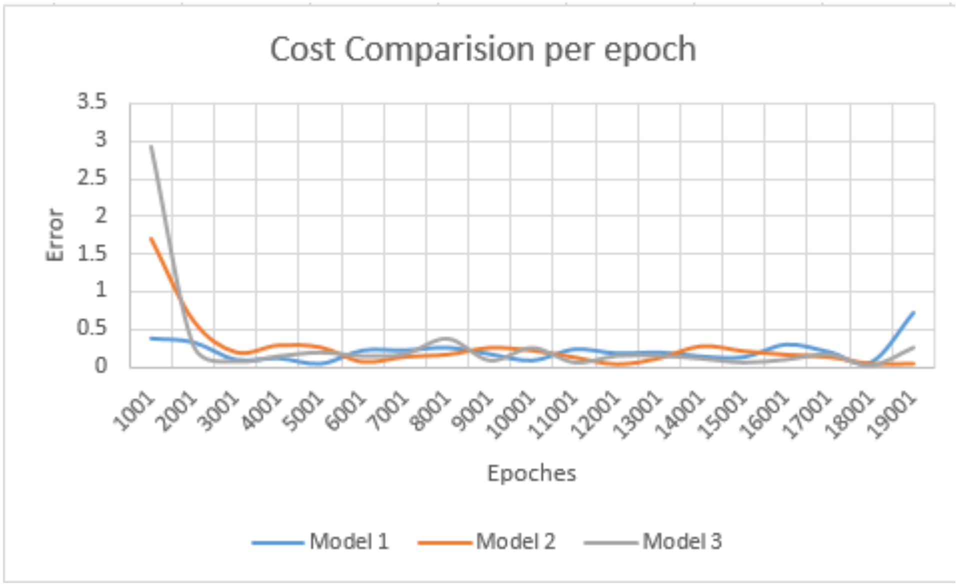
```
Epoch: 1 cost = 297.316
Epoch: 1001 cost = 1.712
Epoch: 2001 cost = 0.610
Epoch: 3001 cost = 0.189
Epoch: 4001 cost = 0.280
Epoch: 5001 cost = 0.254
Epoch: 6001 cost = 0.059
Epoch: 7001 cost = 0.128
Epoch: 8001 cost = 0.157
Epoch: 9001 cost = 0.248
Epoch: 10001 cost = 0.211
Epoch: 11001 cost = 0.119
Epoch: 12001 cost = 0.027
Epoch: 13001 cost = 0.108
Epoch: 14001 cost = 0.271
Epoch: 15001 cost = 0.201
Epoch: 16001 cost = 0.155
Epoch: 17001 cost = 0.124
Epoch: 18001 cost = 0.034
Epoch: 19001 cost = 0.036
Validation accuracy 0.954487
Test accuracy 0.957453
```

Model 2

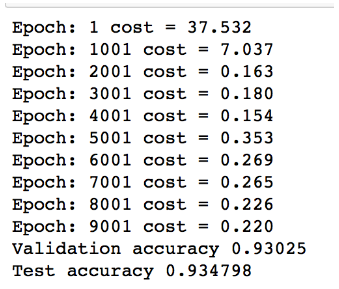


```
Epoch: 1 cost = 319.544
Epoch: 1001 cost = 2.934
Epoch: 2001 cost = 0.290
Epoch: 3001 cost = 0.081
Epoch: 4001 cost = 0.151
Epoch: 5001 cost = 0.201
Epoch: 6001 cost = 0.154
Epoch: 7001 cost = 0.182
Epoch: 8001 cost = 0.384
Epoch: 9001 cost = 0.094
Epoch: 10001 cost = 0.263
Epoch: 11001 cost = 0.068
Epoch: 12001 cost = 0.156
Epoch: 13001 cost = 0.158
Epoch: 14001 cost = 0.120
Epoch: 15001 cost = 0.068
Epoch: 16001 cost = 0.108
Epoch: 17001 cost = 0.175
Epoch: 18001 cost = 0.030
Epoch: 19001 cost = 0.266
Validation accuracy 0.953993
Test accuracy 0.955331
```

Model 3

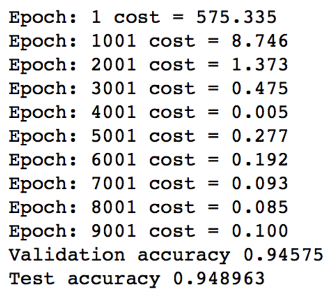


The dropout rate was also tested using the validation set and the range of values 0.1, 0.5, 0.9 on Model 1 with resolution 27x33. For dropout rate=0.5, Validation Accuracy was 0.945. For dropout rate=0.1, Validation Accuracy was 0.930. For dropout rate-0.9, Validation Accuracy was 0.932.



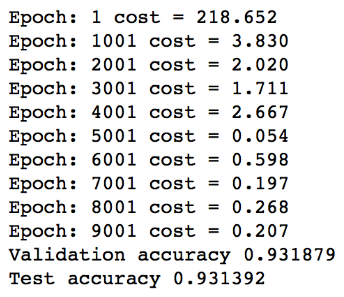
```
Epoch: 1 cost = 37.532
Epoch: 1001 cost = 7.037
Epoch: 2001 cost = 0.163
Epoch: 3001 cost = 0.180
Epoch: 4001 cost = 0.154
Epoch: 5001 cost = 0.353
Epoch: 6001 cost = 0.269
Epoch: 7001 cost = 0.265
Epoch: 8001 cost = 0.226
Epoch: 9001 cost = 0.220
Validation accuracy 0.93025
Test accuracy 0.934798
```

Dropout 0.9



```
Epoch: 1 cost = 575.335
Epoch: 1001 cost = 8.746
Epoch: 2001 cost = 1.373
Epoch: 3001 cost = 0.475
Epoch: 4001 cost = 0.005
Epoch: 5001 cost = 0.277
Epoch: 6001 cost = 0.192
Epoch: 7001 cost = 0.093
Epoch: 8001 cost = 0.085
Epoch: 9001 cost = 0.100
Validation accuracy 0.94575
Test accuracy 0.948963
```

Dropout 0.5



```
Epoch: 1 cost = 218.652
Epoch: 1001 cost = 3.830
Epoch: 2001 cost = 2.020
Epoch: 3001 cost = 1.711
Epoch: 4001 cost = 2.667
Epoch: 5001 cost = 0.054
Epoch: 6001 cost = 0.598
Epoch: 7001 cost = 0.197
Epoch: 8001 cost = 0.268
Epoch: 9001 cost = 0.207
Validation accuracy 0.931879
Test accuracy 0.931392
```

Dropout 0.1

5. Retrain Model Using Higher Resolution Using Full Training Set

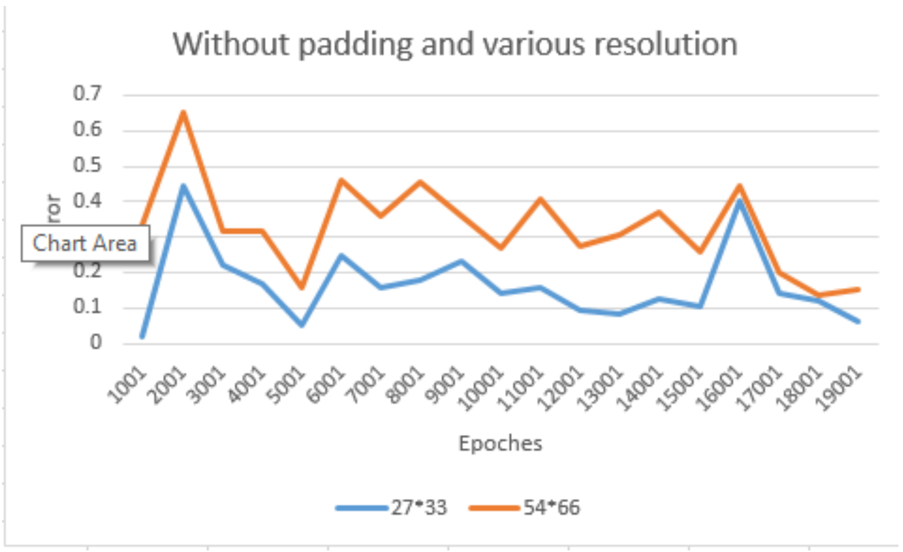
Many different resolutions were run. Below, we compare (27, 33) to the higher resolution (54,66) trained on all of the training set data. Notice that the test accuracy does not improve greatly (0.958 vs. 0.964).

```
Epoch: 1 cost = 606.299
Epoch: 1001 cost = 0.017
Epoch: 2001 cost = 0.443
Epoch: 3001 cost = 0.222
Epoch: 4001 cost = 0.167
Epoch: 5001 cost = 0.051
Epoch: 6001 cost = 0.246
Epoch: 7001 cost = 0.158
Epoch: 8001 cost = 0.177
Epoch: 9001 cost = 0.232
Epoch: 10001 cost = 0.144
Epoch: 11001 cost = 0.160
Epoch: 12001 cost = 0.096
Epoch: 13001 cost = 0.081
Epoch: 14001 cost = 0.126
Epoch: 15001 cost = 0.102
Epoch: 16001 cost = 0.404
Epoch: 17001 cost = 0.142
Epoch: 18001 cost = 0.121
Epoch: 19001 cost = 0.063
Validation accuracy 0.95582
Test accuracy 0.958193
```

Image resized to 27, 33

```
Epoch: 1 cost = 68.590
Epoch: 1001 cost = 0.317
Epoch: 2001 cost = 0.208
Epoch: 3001 cost = 0.097
Epoch: 4001 cost = 0.149
Epoch: 5001 cost = 0.109
Epoch: 6001 cost = 0.212
Epoch: 7001 cost = 0.202
Epoch: 8001 cost = 0.278
Epoch: 9001 cost = 0.126
Epoch: 10001 cost = 0.126
Epoch: 11001 cost = 0.247
Epoch: 12001 cost = 0.177
Epoch: 13001 cost = 0.227
Epoch: 14001 cost = 0.246
Epoch: 15001 cost = 0.159
Epoch: 16001 cost = 0.039
Epoch: 17001 cost = 0.056
Epoch: 18001 cost = 0.015
Epoch: 19001 cost = 0.090
Validation accuracy 0.960411
Test accuracy 0.964314
```

Image resized to 54, 66



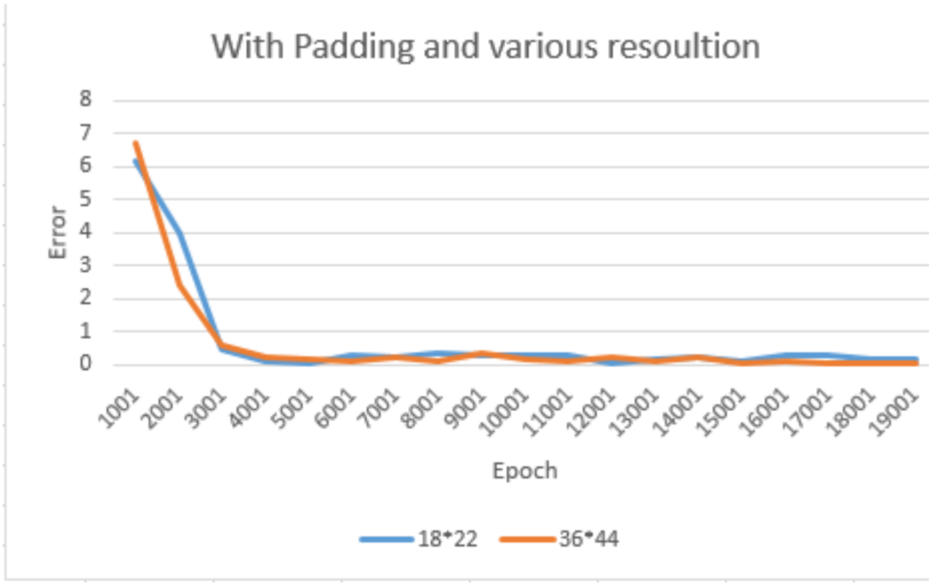
Also, we tried padding the image and resizing the image by preserving the aspect ratio of the image. Below, we compare (18, 22) to the higher resolution (36, 44) trained on all of the training set data. Notice that the test accuracy does not improve greatly (0.953 vs. 0.95).

```
Epoch: 1 cost = 17.828
Epoch: 1001 cost = 6.154
Epoch: 2001 cost = 3.998
Epoch: 3001 cost = 0.488
Epoch: 4001 cost = 0.085
Epoch: 5001 cost = 0.065
Epoch: 6001 cost = 0.275
Epoch: 7001 cost = 0.214
Epoch: 8001 cost = 0.349
Epoch: 9001 cost = 0.276
Epoch: 10001 cost = 0.253
Epoch: 11001 cost = 0.270
Epoch: 12001 cost = 0.044
Epoch: 13001 cost = 0.147
Epoch: 14001 cost = 0.204
Epoch: 15001 cost = 0.075
Epoch: 16001 cost = 0.252
Epoch: 17001 cost = 0.309
Epoch: 18001 cost = 0.156
Epoch: 19001 cost = 0.156
Validation accuracy 0.950094
Test accuracy 0.953406
```

Padded to 180,220 and then resized to 18,22

```
Epoch: 1 cost = 1141.278
Epoch: 1001 cost = 6.694
Epoch: 2001 cost = 2.421
Epoch: 3001 cost = 0.564
Epoch: 4001 cost = 0.200
Epoch: 5001 cost = 0.149
Epoch: 6001 cost = 0.087
Epoch: 7001 cost = 0.189
Epoch: 8001 cost = 0.121
Epoch: 9001 cost = 0.345
Epoch: 10001 cost = 0.143
Epoch: 11001 cost = 0.079
Epoch: 12001 cost = 0.201
Epoch: 13001 cost = 0.097
Epoch: 14001 cost = 0.240
Epoch: 15001 cost = 0.054
Epoch: 16001 cost = 0.119
Epoch: 17001 cost = 0.053
Epoch: 18001 cost = 0.032
Epoch: 19001 cost = 0.047
Validation accuracy 0.951673
Test accuracy 0.955429
```

Padded to 180,220 and then resized to 36,44



This shows the convolutional neural networks are resilient to compression and resolution changes. Note that higher resolutions could have been chosen, but processing those resolutions was not efficient.

6. Model Using Higher Resolution in Terms of Channels

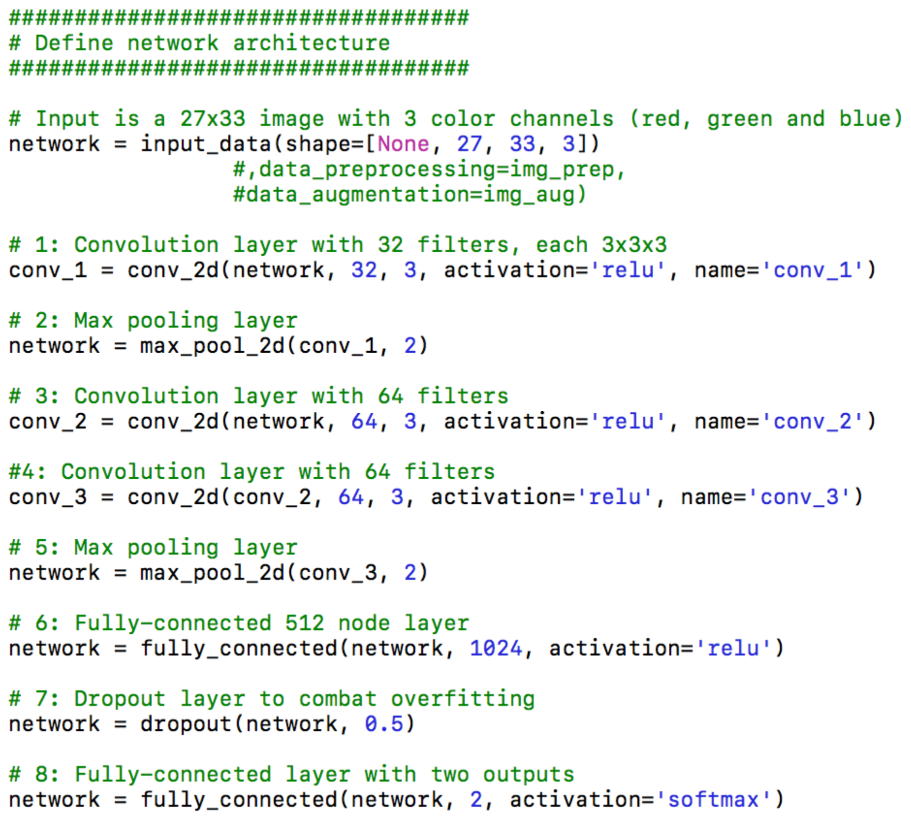
In addition to the lower resolution of using only one channel (grayscale), we also created a model using all three channels (RGB). Since building an architecture for convolutional neural networks is somewhat of an art and takes a lot of time to run, the architecture already implemented by tensorflow to classify objects (CIFAR-10) was used. Like above, we used Rectified Linear Unit (f(x)=max(0,x)) activation function as opposed to sigmoid activation function, since it is computationally simple.

Model Layers and Specifications:

1: Convolution layer with 32 filters, Filter size: 3x3x3
2: Max pooling layer (Filter: 2x2)
3: Convolution layer with 64 filters, Filter size: 3x3x3
4: Convolution layer with 64 filters, Filter size: 3x3x3
5: Max pooling layer (Filter: 2x2)
# 6: Fully-connected 1024 node layer (Note: 1024 was used instead of 512 since we are using regularization in the form of dropout
7: Dropout layer to combat overfitting with dropout rate=0.5
8: Fully-connected layer with two outputs using softmax

Hyperparameters:

We set our learning rate to 0.001, our dropout rate to 0.5, and number of layers to 8. Filter size was chosen to be 3x3x3, since we are trying to depict smaller differences in the form of glasses. Padding was set to "same." Our batch size was set to 50.

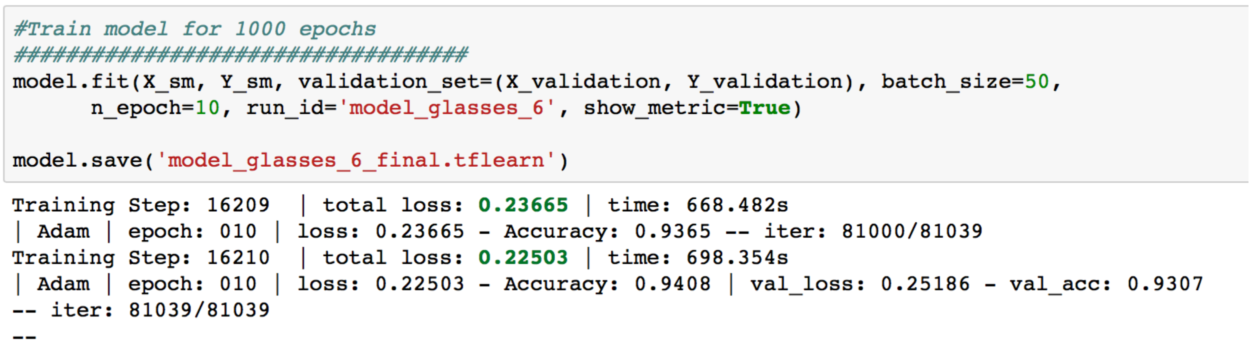


```
###################################
# Define network architecture
###################################

# Input is a 27x33 image with 3 color channels (red, green and blue)
network = input_data(shape=[None, 27, 33, 3])
                 #,data_preprocessing=img_prep,
                 #data_augmentation=img_aug)

# 1: Convolution layer with 32 filters, each 3x3x3
conv_1 = conv_2d(network, 32, 3, activation='relu', name='conv_1')

# 2: Max pooling layer
network = max_pool_2d(conv_1, 2)

# 3: Convolution layer with 64 filters
conv_2 = conv_2d(network, 64, 3, activation='relu', name='conv_2')

#4: Convolution layer with 64 filters
conv_3 = conv_2d(conv_2, 64, 3, activation='relu', name='conv_3')

# 5: Max pooling layer
network = max_pool_2d(conv_3, 2)

# 6: Fully-connected 512 node layer
network = fully_connected(network, 1024, activation='relu')

# 7: Dropout layer to combat overfitting
network = dropout(network, 0.5)

# 8: Fully-connected layer with two outputs
network = fully_connected(network, 2, activation='softmax')
```

The model then was trained in the following way:

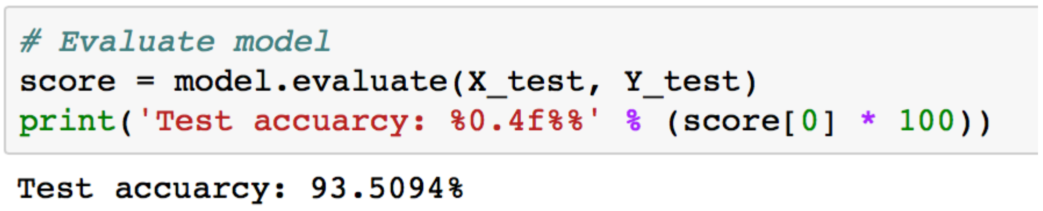
```
# Configure how the network will be trained
acc = Accuracy(name="Accuracy")
network = regression(network, optimizer='adam',
                     loss='categorical_crossentropy',
                     learning_rate=0.001, metric=acc)

# Wrap the network in a model object
model = tflearn.DNN(network, checkpoint_path='model_glasses_6.tflearn', max_checkpoints = 3,
                    tensorboard_verbose = 3, tensorboard_dir='tmp/tflearn_logs/')
###################################
# Train model for 1000 epochs
###################################
model.fit(X_sm, Y_sm, validation_set=(X_validation, Y_validation), batch_size=50,
      n_epoch=1000, run_id='model_glasses_6', show_metric=True)

model.save('model_glasses_6_final.tflearn')

# Evaluate model
score = model.evaluate(X_test, Y_test)
print('Test accuarcy: %0.4f%%' % (score[0] * 100))
```

Since the model is very computationally expensive, we have only run 10 epochs to show how the code works and a potential accuracy. With only 10 epochs, a batch_size=50, and a learning rate=0.001, validation accuracy was 0.9307. Predicting the test set, gave us a test accuracy of 0.935.



```
#Train model for 1000 epochs
###################################
model.fit(X_sm, Y_sm, validation_set=(X_validation, Y_validation), batch_size=50,
      n_epoch=10, run_id='model_glasses_6', show_metric=True)

model.save('model_glasses_6_final.tflearn')
```

```
Training Step: 16209  | total loss: 0.23665 | time: 668.482s
| Adam | epoch: 010 | loss: 0.23665 - Accuracy: 0.9365 -- iter: 81000/81039
Training Step: 16210  | total loss: 0.22503 | time: 698.354s
| Adam | epoch: 010 | loss: 0.22503 - Accuracy: 0.9408 | val_loss: 0.25186 - val_acc: 0.9307
-- iter: 81039/81039
--
```



```
# Evaluate model
score = model.evaluate(X_test, Y_test)
print('Test accuarcy: %0.4f%%' % (score[0] * 100))
```

```
Test accuarcy: 93.5094%
```

In addition, to we tried running other models, but the kernel would die or they are continuing to run:

```
###################################
# Train model for 1000 epochs
###################################
model.fit(X_sm, Y_sm, validation_set=(X_validation, Y_validation), batch_size=100,
      n_epoch=1000, run_id='model_glasses_6', show_metric=True)

model.save('model_glasses_6_final.tflearn')
```

```
Training Step: 25991  | total loss: 0.23612 | time: 14.189s
| Adam | epoch: 033 | loss: 0.23612 - Accuracy: 0.9367 -- iter: 03900/81039
```

7. Conclusion

We trained dataset with different set of parameter and come up with three best models. With these three models, we test dataset with varied dropout ratio and with different resolution and training set sizes. After testing with all these parameters, we found out the best accuracy for image resolution of 54*66 and dropout ratio of 0.5.