ECE 763-Project 2

April 8, 2018

Contents

1 Preprocessing Dataset

The dataset used was a combination of the AFLW, FDDB and LFW datasets. The total number of training images was 10,000 and the number of testing images was 1000, per class. Since this is a binary classification task, the total size of the training set was 20,000 and the total size of the testing set was 2000.
The dimensions of the cropped faces and non-faces were taken to be 64*64.
The glob library was used for easy reading of files. The process is slow on the first run as it builds a list of paths in the RAM, but for subsequent runs, it is extremely fast.

2 Babysitting Neural Network

This section describes the different architectures that were used for the face and non-face dataset and the accuracy and loss curves for them.
A short section describing their performance on a few separate test images and a live video stream is also discussed.
Framework: Tensorflow
Build: From Source with GPU support via CUDA 9.1 and CuDNN 7.1
Platform:Ubuntu 16.04

2.1 Architectures

The Adam Optimizer was used for all the models along with softmax with cross entropy as the loss due to the one-hot encoding of the labels.

2.1.1 Multi-Layer Perceptron

The first architecture tested was a simple MLP with 2 hidden layers.
The training hyperparameters and the code for the model are shown below.

```python
# Parameters
learning_rate = 0.001
training_epochs = 300
batch_size = 128
display_step = 10
logs_path ='./logs/basic_net/'

# Network Parameters
n_hidden_1 = 2048 # 1st layer number of neurons
n_hidden_2 = 1024 # 2nd layer number of neurons
n_input = patch_size*patch_size*3 # data input (img shape: 64*64)
n_classes = 2

# tf Graph input
X = tf.placeholder("float", [None, n_input],name="X")
Y = tf.placeholder("float", [None, n_classes],name="Y")
print(X.name)
print(Y.name)
# Store layers weight & bias
with tf.name_scope('weights'):
    weights = {
        'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
        'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
        'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
    }

with tf.name_scope('biases'):
    biases = {
        'b1': tf.Variable(tf.random_normal([n_hidden_1])),
        'b2': tf.Variable(tf.random_normal([n_hidden_2])),
        'out': tf.Variable(tf.random_normal([n_classes]))
    }



# Create model
def multilayer_perceptron(x):
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1=tf.nn.relu(layer_1)
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_2=tf.nn.relu(layer_2)
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    #print(out_layer.name)
    return out_layer
```

The following figures show the loss and accuracy curves for the basic MLP. Tensorboard was used to generate the graphs in realtime and was instrumental in hyperparameter tuning.
We can see from the figure that the accuracy increases with time and the loss also decreases, but the curves are not very smooth and the performance could be better. This is because we feed the image into the MLP as a **64*64*3** long 1-D vector. There is a loss of spatial information.
Also, a higher batch size produces a smoother loss curve.

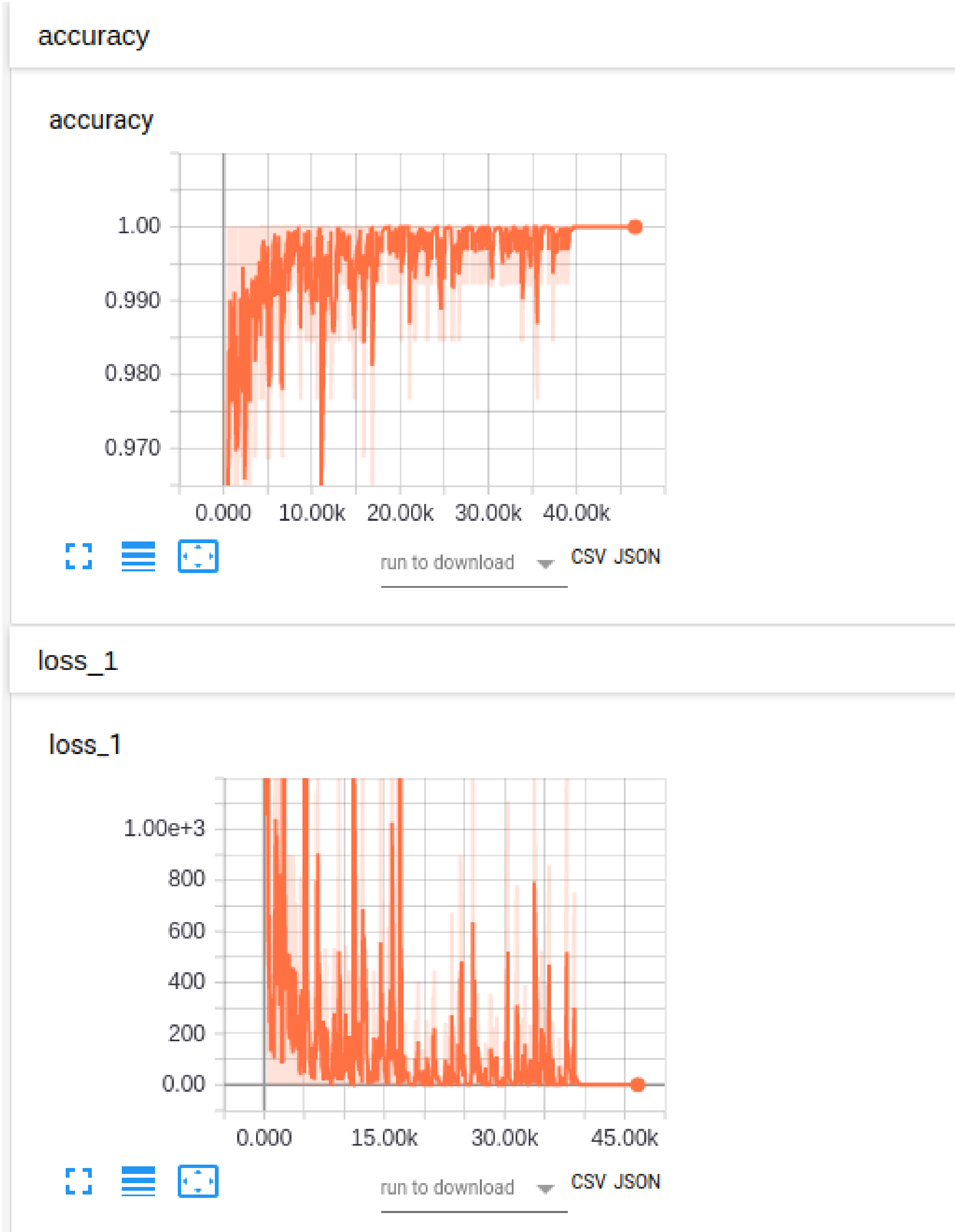

Figure 1: Basic MLP with batch size 128.

Figure 2: Basic MLP with batch size 256.

Though the MLP is a good start, we can definitely do better. This is why we now see the performance of 2 CNN based approaches.
The next architecture is Lenet.

2.1.2 Lenet

The declaration and the hyperparameters are shown in the code section below.

```
# Training Parameters
learning_rate = 0.01
training_epochs = 1000
batch_size = 128
display_step = 10
threshold=0.01
```

```
logs_path='./logs/cnn/'

# Network Parameters
num_input = patch_size*patch_size*3 # data input (img shape: patch_size*patch_size)
num_classes = 2  # total classes
dropout = 0.70  # Dropout, probability to keep units

# tf Graph input
X = tf.placeholder(tf.float32, [None, num_input]) #batch_size, num_input
Y = tf.placeholder(tf.float32, [None, num_classes]) #batch_size, num_classes
keep_prob = tf.placeholder(tf.float32)  # dropout (keep probability)

# Create some wrappers for simplicity
def conv2d(x, W, b, strides=1):
    # Conv2D wrapper, with bias and relu activation
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], padding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)


def maxpool2d(x, k=2):
    # MaxPool2D wrapper
    return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k, 1],
                          padding='SAME')


# create Lenet model
def Lenet(x,weights,biases,dropout):
    #reshaping
    x=tf.reshape(x,shape=[-1,patch_size,patch_size,3])
    #convolution layer 1
    conv1=conv2d(x,weights['wc1'],biases['bc1'])
    # Max Pooling (down-sampling)
    conv1 = maxpool2d(conv1, k=2)

    # Convolution Layer
    conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])
    # Max Pooling (down-sampling)
    conv2 = maxpool2d(conv2, k=2)

    # Fully connected layer
    # Reshape conv2 output to fit fully connected layer input
    fc1 = tf.reshape(conv2, [-1, weights['wd1'].get_shape().as_list()[0]])
    #fc1=flatten(conv2)
    fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])
    fc1 = tf.nn.relu(fc1)
    # Apply Dropout
    fc1 = tf.nn.dropout(fc1, dropout)

    fc2= tf.add(tf.matmul(fc1,weights['wd2']),biases['bd2'])
    fc2=tf.nn.relu(fc2)

    fc3 = tf.add(tf.matmul(fc2, weights['wd3']), biases['bd3'])
    fc3 = tf.nn.relu(fc3)
    # Output, class prediction
    out = tf.add(tf.matmul(fc3, weights['out']), biases['out'])
    return out

    """
    Input
    x->contains the images in (patch_size,patch_size,3) format
    The LeNet architecture accepts a 32x32xC image as input, where C is the number
    of color channels.
    Modifying to accept 64x64 images

    Architecture
    Layer 1: Convolutional. The output shape should be 28x28x6.

    Activation. Your choice of activation function.

    Pooling. The output shape should be 14x14x6.

    Layer 2: Convolutional. The output shape should be 10x10x16.
```

```

    Activation. Your choice of activation function.

    Pooling. The output shape should be 5x5x16.

    Flatten. Flatten the output shape of the final pooling layer such that it's 1D
    instead of 3D. The easiest way to do is by using tf.contrib.layers.flatten,
    which is already imported for you.

    Layer 3: Fully Connected. This should have 120 outputs.

    Activation. Your choice of activation function.

    Layer 4: Fully Connected. This should have 84 outputs.

    Activation. Your choice of activation function.

    Layer 5: Fully Connected (Logits). This should have 10 outputs.
    """


#Lenet
# Store layers weight & bias
lenet_weights = {
    # 5x5 conv, 1 input, 32 outputs
    'wc1': tf.Variable(tf.random_normal([5, 5, 3, 6])),
    # 5x5 conv, 6 inputs, 16 outputs
    'wc2': tf.Variable(tf.random_normal([5, 5, 6, 16])),
    # fully connected, 16*16*16 inputs, 400 outputs
    'wd1': tf.Variable(tf.random_normal([(16)*(16)*16, 400])),
    # fully connected 400 inputs, 120 outputs
    'wd2': tf.Variable(tf.random_normal([400,120])),
    #fully connected 120 inputs, 84 outputs
    'wd3': tf.Variable(tf.random_normal([120,84])),
    #fully connected 84 inputs, 2 outputs
    'out': tf.Variable(tf.random_normal([84,num_classes]))
}

lenet_biases = {
    'bc1': tf.Variable(tf.random_normal([6])),
    'bc2': tf.Variable(tf.random_normal([16])),
    'bd1': tf.Variable(tf.random_normal([400])),
    'bd2': tf.Variable(tf.random_normal([120])),
    'bd3': tf.Variable(tf.random_normal([84])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}
```

The following figures show the loss and accuracy curves for Lenet. Tensorboard was used here as well to generate the graphs in realtime and was instrumental in hyperparameter tuning.
We can see from the figure that the accuracy increases with time and the loss also decreases, the curves are smooth and convergence is quick. The training time is also faster than the MLP, due to the reduced number of parameters.

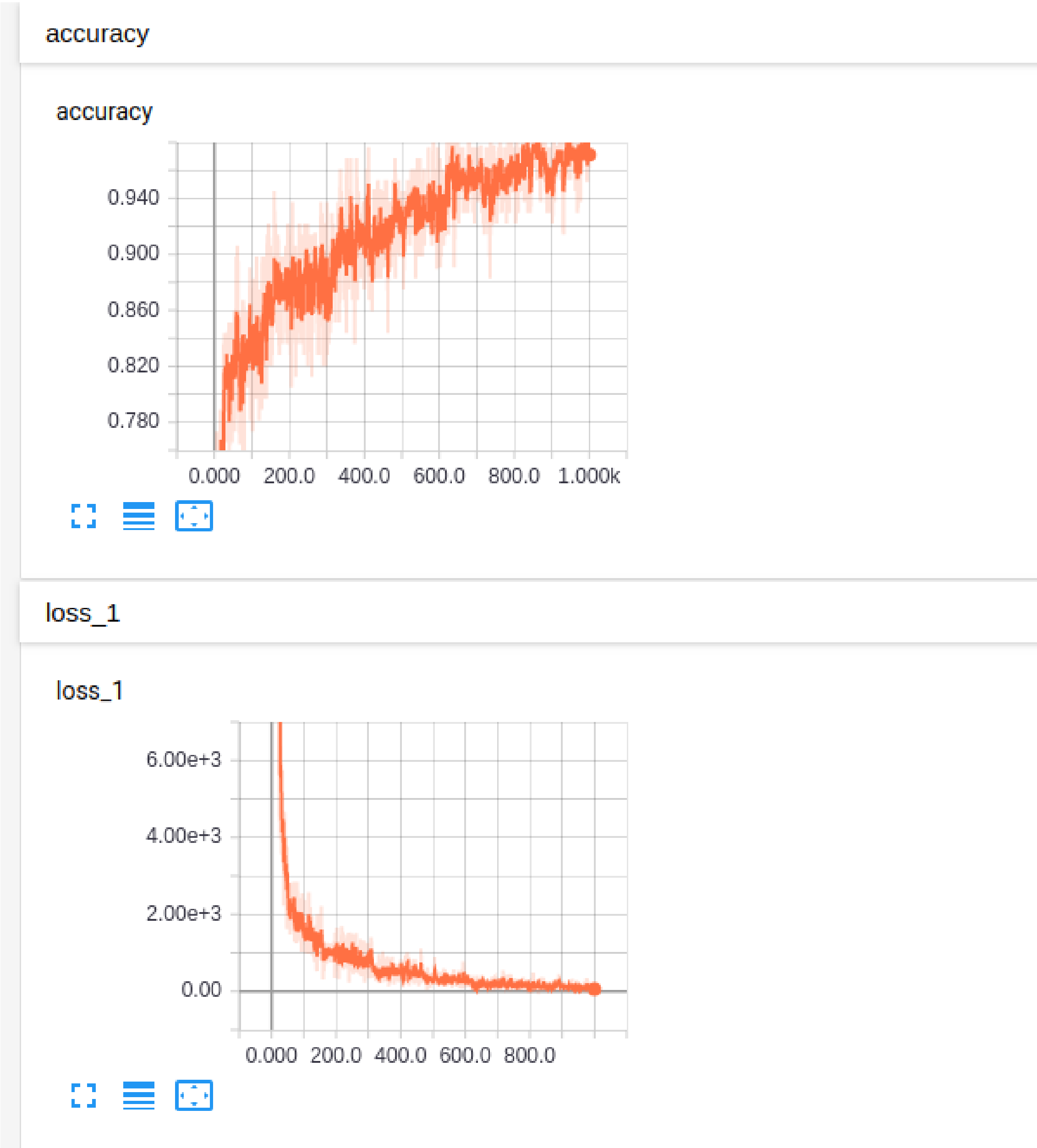

Figure 3: Lenet with batch size 128, 1000 epochs

2.1.3 Own Architecture

The declaration is shown in the code section below.

```
}


# Create model
def conv_net(x, weights, biases, dropout):
    # Reshape to match picture format [Height x Width x Channel]
    # Tensor input become 4-D: [Batch Size, Height, Width, Channel]
    x = tf.reshape(x, shape=[-1, patch_size, patch_size, 3])

    # Convolution Layer
    conv1 = conv2d(x, weights['wc1'], biases['bc1'])
    # Max Pooling (down-sampling)
    conv1 = maxpool2d(conv1, k=2)

```

```
    # Convolution Layer
    conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])
    # Max Pooling (down-sampling)
    conv2 = maxpool2d(conv2, k=2)

    # Fully connected layer
    # Reshape conv2 output to fit fully connected layer input
    fc1 = tf.reshape(conv2, [-1, weights['wd1'].get_shape().as_list()[0]])
    fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])
    fc1 = tf.nn.relu(fc1)
    # Apply Dropout
    fc1 = tf.nn.dropout(fc1, dropout)

    # Output, class prediction
    out = tf.add(tf.matmul(fc1, weights['out']), biases['out'])
    return out
```

The following figures show the loss and accuracy curves for the custom architecture. Tensorboard was used here as well to generate the graphs in realtime and was instrumental in hyperparameter tuning.

We can see from the figure that the accuracy increases with time and the loss also decreases, the curves are smooth and convergence is quick. The training time is higher than Lenet as the number of parameters are higher. The depth of the convolution layer is higher here, 32 compared to 6 in Lenet. This is for more detailed feature extraction.

As expected, this net produces the best results.

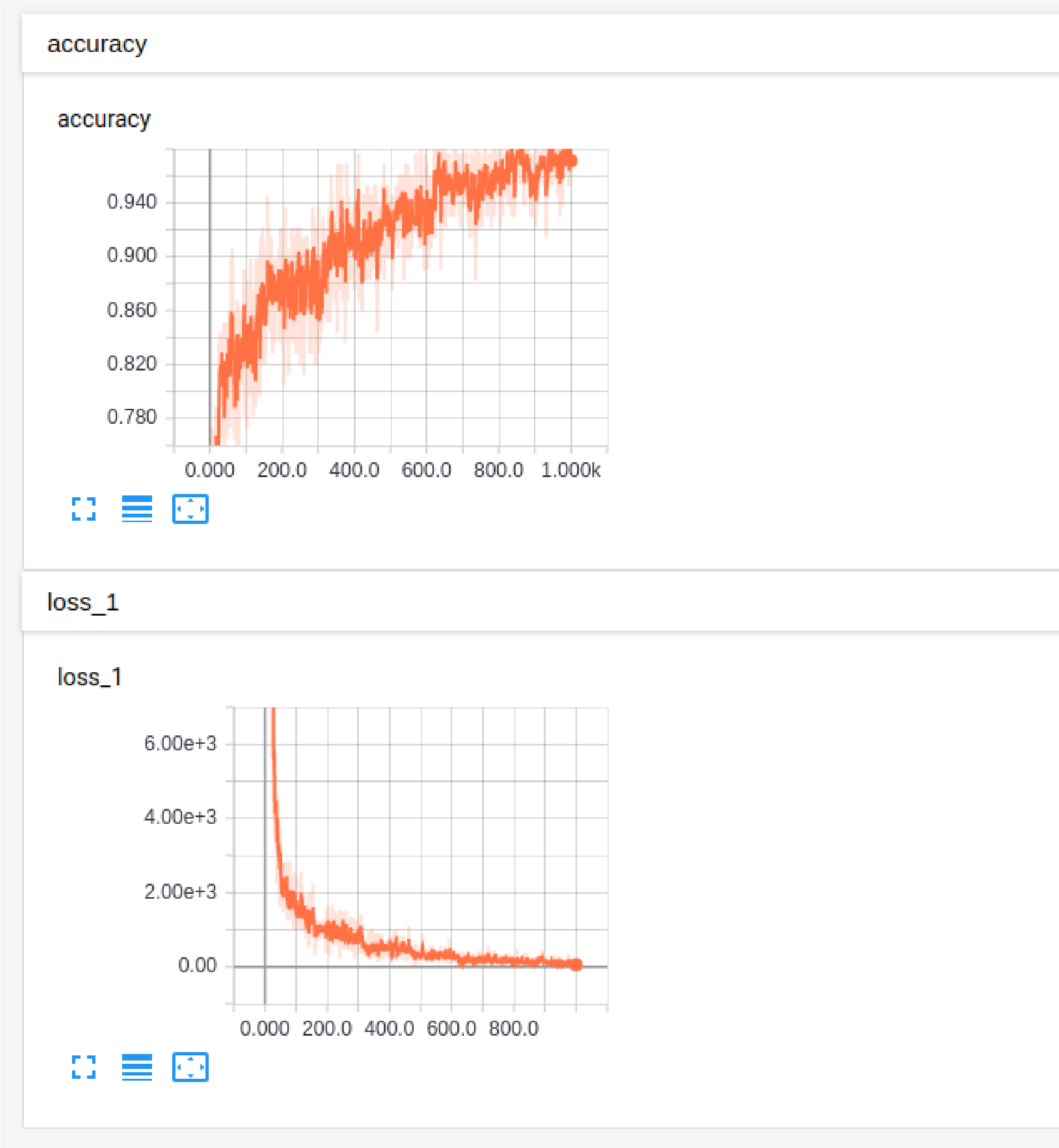

Figure 4: Custom CNN with batch size 128.

3 Conclusion

Overall we can see that the custom CNN architecture performed really well on the face detection task. This was a good primer for the final project of the course.

4 GitHub Repo for Code

The code can be found at https://github.com/hari0920/ECE-763-Final-Project.