Traffic_Sign_Classifier

July 11, 2017

1 Self-Driving Car Engineer Nanodegree

1.1 Deep Learning

1.2 Project: Build a Traffic Sign Recognition Classifier

In this notebook, a template is provided for you to implement your functionality in stages, which is required to successfully complete this project. If additional code is required that cannot be included in the notebook, be sure that the Python code is successfully imported and included in your submission if necessary.

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the iPython Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to ", "**File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

In addition to implementing code, there is a writeup to complete. The writeup should be completed in a separate file, which can be either a markdown file or a pdf document. There is a write up template that can be used to guide the writing process. Completing the code template and writeup template will cover all of the rubric points for this project.

The rubric contains "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. The stand out suggestions are optional. If you decide to pursue the "stand out suggestions", you can include the code in this Ipython notebook and also discuss the results in the writeup file.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. In addition, Markdown cells can be edited by typically double-clicking the cell to enter edit mode.

1.3 Step 0: Load The Data

```
In [1]: # Load pickled data
        import pickle
```

```
# TODO: Fill this in based on where you saved the training and testing data

training_file = "train.p"
validation_file = "valid.p"
testing_file = "test.p"

with open(training_file, mode='rb') as f:
    train = pickle.load(f)
with open(validation_file, mode='rb') as f:
    valid = pickle.load(f)
with open(testing_file, mode='rb') as f:
    test = pickle.load(f)

X_train, y_train = train['features'], train['labels']
X_valid, y_valid = valid['features'], valid['labels']
X_test, y_test = test['features'], test['labels']
```

1.4 Step 1: Dataset Summary & Exploration

The pickled data is a dictionary with 4 key/value pairs:

- `'features'` is a 4D array containing raw pixel data of the traffic sign images, (num examples, width, height, channels).
- `'labels'` is a 1D array containing the label/class id of the traffic sign. The file `signnames.csv` contains id -> name mappings for each id.
- `'sizes'` is a list containing tuples, (width, height) representing the original width and height the image.
- `'coords'` is a list containing tuples, (x1, y1, x2, y2) representing coordinates of a bounding box around the sign in the image. **THESE COORDINATES ASSUME THE ORIGINAL IMAGE. THE PICKLED DATA CONTAINS RESIZED VERSIONS (32 by 32) OF THESE IMAGES**

Complete the basic data summary below. Use python, numpy and/or pandas methods to calculate the data summary rather than hard coding the results. For example, the pandas shape method might be useful for calculating some of the summary results.

1.4.1 Provide a Basic Summary of the Data Set Using Python, Numpy and/or Pandas

In [2]:
```
### Replace each question mark with the appropriate value.
### Use python, pandas or numpy methods rather than hard coding the results

# TODO: Number of training examples
n_train = train['features'].shape[0]

# TODO: Number of validation examples
n_validation = valid['features'].shape[0]
```

```
        # TODO: Number of testing examples.
        n_test = test['features'].shape[0]

        # TODO: What's the shape of an traffic sign image?
        image_shape = (train['features'].shape[1], train['features'].shape[2] )

        # TODO: How many unique classes/labels there are in the dataset.
        n_classes = len(set(valid['labels']))

        print("Number of training examples =", n_train)
        print("Number of testing examples =", n_test)
        print("Image data shape =", image_shape)
        print("Number of classes =", n_classes)
```

```
Number of training examples = 34799
Number of testing examples = 12630
Image data shape = (32, 32)
Number of classes = 43
```

1.4.2 Include an exploratory visualization of the dataset

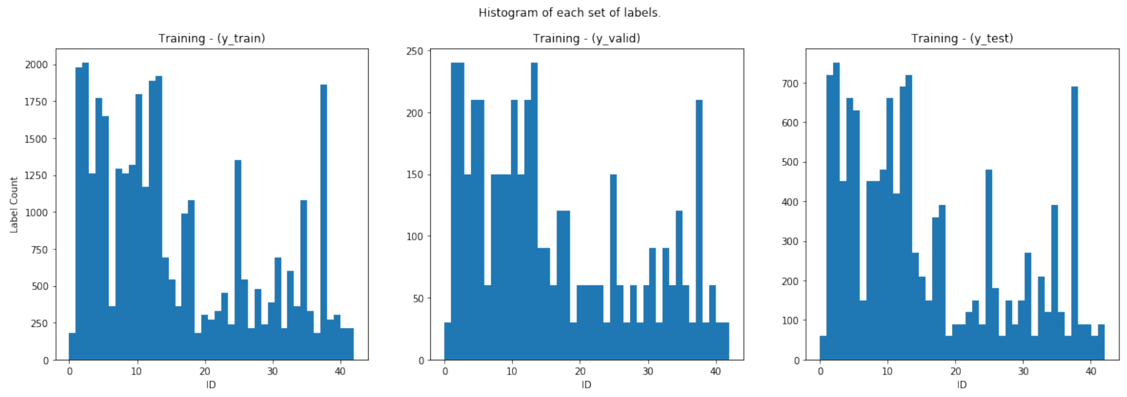
Visualize the German Traffic Signs Dataset using the pickled file(s). This is open ended, suggestions include: plotting traffic sign images, plotting the count of each sign, etc.

The Matplotlib examples and gallery pages are a great resource for doing visualizations in Python.

NOTE: It's recommended you start with something simple first. If you wish to do more, come back to it after you've completed the rest of the sections. It can be interesting to look at the distribution of classes in the training, validation and test set. Is the distribution the same? Are there more examples of some classes than others?

```
In [3]: ### Data exploration visualization code goes here.
        ### Feel free to use as many code cells as needed.
        import matplotlib.pyplot as plt
        # Visualizations will be shown in the notebook.
        %matplotlib inline

        fig, (ax_train, ax_valid, ax_test) = plt.subplots(ncols=3, figsize=(20, 6))
        num_bins = n_classes

        # Training
        x = y_train
        n, bins, pathces = ax_train.hist(x, num_bins)
        ax_train.set_xlabel('ID')
        ax_train.set_ylabel('Label Count')
        ax_train.set_title(r'Training - (y_train)')

        # Valid
```

```
x = y_valid
n, bins, pathces = ax_valid.hist(x, num_bins)
ax_valid.set_xlabel('ID')
ax_valid.set_title(r'Training - (y_valid)')

# Test
x = y_test
n, bins, pathces = ax_test.hist(x, num_bins)
ax_test.set_xlabel('ID')
ax_test.set_title(r'Training - (y_test)')

# Plot
fig.suptitle("Histogram of each set of labels.")
plt.show()
```



1.4.3 Display Three Example Signs

Here I show three (3) example images as a prelude to the processing step.

```
In [4]: import numpy as np
        import random

        num_ex_images = 3
        images = []
        for i in range(num_ex_images):
            idx = random.randint(0, len(X_train))
            images.append(X_train[idx])

        fig, (ax0, ax1, ax2) = plt.subplots(ncols=3, figsize=(8,8))
        ax0.imshow(images[0])
        ax1.imshow(images[1])
        ax1.axis('off')
        ax2.imshow(images[2])
```

```
ax2.axis('off')
plt.show()
```

1.5 Step 2: Design and Test a Model Architecture

Design and implement a deep learning model that learns to recognize traffic signs. Train and test your model on the German Traffic Sign Dataset.

The LeNet-5 implementation shown in the classroom at the end of the CNN lesson is a solid starting point. You'll have to change the number of classes and possibly the preprocessing, but aside from that it's plug and play!

With the LeNet-5 solution from the lecture, you should expect a validation set accuracy of about 0.89. To meet specifications, the validation set accuracy will need to be at least 0.93. It is possible to get an even higher accuracy, but 0.93 is the minimum for a successful project submission.

There are various aspects to consider when thinking about this problem:

- Neural network architecture (is the network over or underfitting?)
- Play around preprocessing techniques (normalization, rgb to grayscale, etc)
- Number of examples per label (some have more than others).
- Generate fake data.

Here is an example of a published baseline model on this problem. It's not required to be familiar with the approach used in the paper but, it's good practice to try to read papers like these.

1.5.1 Pre-process the Data Set (normalization, grayscale, etc.)

Minimally, the image data should be normalized so that the data has mean zero and equal variance. For image data, `(pixel - 128)/ 128` is a quick way to approximately normalize the data and can be used in this project.

Other pre-processing steps are optional. You can try different techniques to see if it improves performance.

Use the code cell (or multiple code cells, if necessary) to implement the first step of your project.

1 | Shuffle Training Data

```
In [5]: from sklearn.utils import shuffle
        X_train, y_train = shuffle(X_train, y_train)
```

2 | Normalization

```
In [6]: ### Preprocess the data here. It is required to normalize the data. Other preprocessing
        ### converting to grayscale, etc.
        ### Feel free to use as many code cells as needed.
        def preprocess_normalize(X_data):
            X_normalized = np.array(X_data, dtype=np.float32)
            n = len(X_normalized)
            n_layers = X_normalized.shape[-1]
            for k in range(n):
                for layer in range(n_layers):
                    X_normalized[k][:, :, layer] = (X_normalized[k][:, :, layer] - np.mean(X_nor
            print("Normalized {} entries.".format(n))
            return X_normalized
        X_train_n = preprocess_normalize(X_train)
        X_valid_n = preprocess_normalize(X_valid)
        X_test_n = preprocess_normalize(X_test)
```

```
Normalized 34799 entries.
Normalized 4410 entries.
Normalized 12630 entries.
```

3 | Model Architecture

Input
I'll use the LeNet architecture which accepts a 32x32 image composed of 3 color layers (RGB).

Architecture ** Layer 1: Convolutional ** The output shape will be 28x28x6.
** Activation ** ReLU
** Pooling ** Output shape is 14x14x6.
** Layer 2: Convolutional ** Output shape is 10x10x16
** Activation ** ReLU
** Pooling ** Output shape is 5x5x16
** Flatten ** Flatten the output shape of the final pooling layer such that it's 1D instead of 3D.
** Layer 3: Fully Connected ** This should have "X" outputs.
** Activation ** ReLU
** Layer 4: Fully Connected ** This should have "X" outputs.
** Activation ** ReLU
** Layer 5: Fully Connected (Logits) ** This should have "X" outputs.
Output
Return the result of the 2nd fully connected layer.

```
In [7]: ### Define your architecture here.
        ### Feel free to use as many code cells as needed.
```

```python
import tensorflow as tf
from tensorflow.contrib.layers import flatten

# Define epoch and batch size
EPOCHS = 300
BATCH_SIZE = 128
learning_rate = 0.00025

def LeNet(x):
    mu = 0.
    sigma = 0.1

    ## LAYER 1. 32x32x3 -> 28x28x6
    in_depth = 3
    out_depth = 6
    weights = tf.Variable(tf.truncated_normal([5, 5, in_depth, out_depth], mu, sigma))
    biases = tf.Variable(tf.truncated_normal([out_depth], mu, sigma))
    strides = [1, 1, 1, 1]
    padding = 'VALID'
    conv_1 = tf.nn.conv2d(x, weights, strides, padding)
    conv_1 = tf.nn.bias_add(conv_1, biases)
    conv_1 = tf.nn.relu(conv_1)
    # Pooling 28x28x6 -> 14x14x6
    strides = [1, 2, 2, 1]
    ksize = [1, 2, 2, 1]
    layer_1_output = tf.nn.max_pool(conv_1, ksize, strides, padding)
    ## LAYER 2. 14x14x6 -> 10x10x16
    in_depth = 6
    out_depth = 16
    weights = tf.Variable(tf.truncated_normal([4, 4, in_depth, out_depth], mu, sigma))
    biases = tf.Variable(tf.truncated_normal([out_depth], mu, sigma))
    strides = [1, 1, 1, 1]
    padding = 'VALID'
    conv_2 = tf.nn.conv2d(layer_1_output, weights, strides, padding)
    conv_2 = tf.nn.bias_add(conv_2, biases)
    conv_2 = tf.nn.relu(conv_2)
    # Pooling 10x10x16 -> 5x5x16
    strides = [1, 2, 2, 1]
    ksize = [1, 2, 2, 1]
    layer_2 = tf.nn.max_pool(conv_2, ksize, strides, padding)
    # Flatten 5x5x16 -> 400
    layer_2_output = tf.contrib.layers.flatten(layer_2)
    ## LAYER 3. Fully Connected 400 -> 120
    weights = tf.Variable(tf.truncated_normal([400, 120], mu, sigma))
    biases = tf.Variable(tf.truncated_normal([120], mu, sigma))
    layer_3 = tf.add(tf.matmul(layer_2_output, weights), biases)
    layer_3_output = tf.nn.relu(layer_3)
    ## LAYER 4. Fully Connected 120 -> 84
    weights = tf.Variable(tf.truncated_normal([120, 84], mu, sigma))
    biases = tf.Variable(tf.truncated_normal([84], mu, sigma))
    layer_4 = tf.add(tf.matmul(layer_3_output, weights), biases)
    layer_4_output = tf.nn.relu(layer_4)
    layer_4_output = tf.nn.dropout(layer_4_output, 0.50)
    ## LAYER 5. Fully Connected 84 -> n_classes (43)
    weights = tf.Variable(tf.truncated_normal([84, n_classes], mu, sigma))
    biases = tf.Variable(tf.truncated_normal([n_classes], mu, sigma))
    logits = tf.add(tf.matmul(layer_4_output, weights), biases)
    return logits
```

1.5.2 Train, Validate and Test the Model

A validation set can be used to assess how well the model is performing. A low accuracy on the training and validation sets imply underfitting. A high accuracy on the training set but low accuracy on the validation set implies overfitting.

```
In [8]: ### Train your model here.
        ### Calculate and report the accuracy on the training and validation set.
        ### Once a final model architecture is selected,
        ### the accuracy on the test set should be calculated and reported as well.
        ### Feel free to use as many code cells as needed.

        x = tf.placeholder(tf.float32, (None, 32, 32, 3))
        y = tf.placeholder(tf.uint8, (None))
        one_hot_y = tf.one_hot(y, n_classes)
```

1.5.3 Training Pipeline

```
In [9]: logits = LeNet(x)
        cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y, logits=logits)
        loss_operation = tf.reduce_mean(cross_entropy)
        optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
        training_operation = optimizer.minimize(loss_operation)
```

1.5.4 Calculate Accuracy

```
In [10]: correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_y, 1))
         accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
         saver = tf.train.Saver()

         def evaluate(X_data, y_data):
             num_examples = len(X_data)
             total_accuracy = 0
             sess = tf.get_default_session()
             for offset in range(0, num_examples, BATCH_SIZE):
                 batch_x, batch_y = X_data[offset:offset+BATCH_SIZE], y_data[offset:offset+BATCH
                 accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batch_y})
```

```
                total_accuracy += (accuracy * len(batch_x))
            return total_accuracy / num_examples
```

1.5.5 Train the Model

```
In [11]: with tf.Session() as sess:
             sess.run(tf.global_variables_initializer())
             num_examples = len(X_train_n)

             print("Training...\n")
             for i in range(EPOCHS):
                 X_train_n, y_train = shuffle(X_train_n, y_train)
                 for offset in range(0, num_examples, BATCH_SIZE):
                     end = offset + BATCH_SIZE
                     batch_x, batch_y = X_train_n[offset:end], y_train[offset:end]
                     sess.run(training_operation, feed_dict={x: batch_x, y: batch_y})

                 validation_accuracy = evaluate(X_valid_n, y_valid)
                 if (i+1)%25 == 0:
                     print("EPOCH {0} of {1}...".format(i+1, EPOCHS))
                     print("Validation Accuracy = {:.3f}\n".format(validation_accuracy))

             saver.save(sess, './traffic_sign_lenet')
             print("Model saved.")
```

```
Training...

EPOCH 25 of 300...
Validation Accuracy = 0.898

EPOCH 50 of 300...
Validation Accuracy = 0.919

EPOCH 75 of 300...
Validation Accuracy = 0.929

EPOCH 100 of 300...
Validation Accuracy = 0.939

EPOCH 125 of 300...
Validation Accuracy = 0.937

EPOCH 150 of 300...
Validation Accuracy = 0.945

EPOCH 175 of 300...
Validation Accuracy = 0.946
```

```
EPOCH 200 of 300...
Validation Accuracy = 0.943

EPOCH 225 of 300...
Validation Accuracy = 0.936

EPOCH 250 of 300...
Validation Accuracy = 0.941

EPOCH 275 of 300...
Validation Accuracy = 0.942

EPOCH 300 of 300...
Validation Accuracy = 0.947

Model saved.
```

1.5.6 Test the Model

```
In [12]: with tf.Session() as sess:
             sess.run(tf.global_variables_initializer())
             new_saver = tf.train.import_meta_graph('./traffic_sign_lenet.meta')
             new_saver.restore(sess, tf.train.latest_checkpoint('.'))
             print(X_test_n.shape)
             test_accuracy = evaluate(X_test_n, y_test)
             print("Test Accuracy = {:.3f}".format(test_accuracy))
```

```
(12630, 32, 32, 3)
Test Accuracy = 0.937
```

1.6 Step 3: Test a Model on New Images

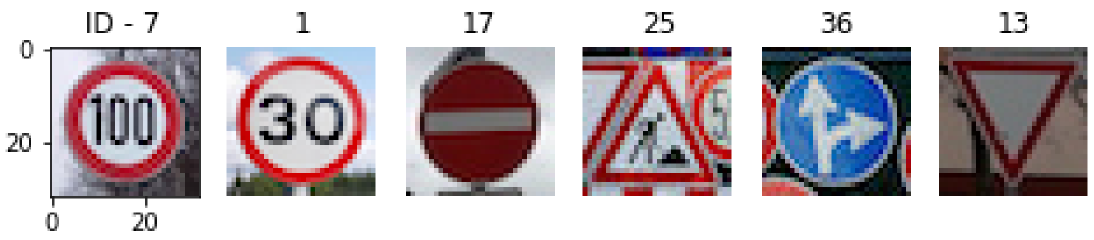
To give yourself more insight into how your model is working, download at least five pictures of German traffic signs from the web and use your model to predict the traffic sign type.

You may find `signnames.csv` useful as it contains mappings from the class id (integer) to the actual sign name.

1.6.1 Load and Output the Images

```
In [13]: ### Credit to David Clark at https://becominghuman.ai/classifying-traffic-signs-728744d
         ### and his github repo (https://github.com/SealedSaint/CarND-Term1-P2) for the 32x32 s
         ### I used the 7 images he has available for testing my model.
         import matplotlib.image as mpimg
         import numpy as np
```

```
test_images = np.array([
    mpimg.imread('1 - 100 speed.png'),
    mpimg.imread('2 - 30 speed.png'),
    mpimg.imread('3 - do not enter.png'),
    mpimg.imread('5 - road work.png'),
    mpimg.imread('6 - straight or right.png'),
    mpimg.imread('7 - yield.png')
])
test_images_labels = np.array([
    7,
    1,
    17,
    25,
    36,
    13
])
fig, (ax0, ax1, ax2, ax3, ax4, ax5) = plt.subplots(ncols=6, figsize=(8,8))
ax0.imshow(test_images[0])
ax0.set_title('ID - 7')
ax1.imshow(test_images[1])
ax1.axis('off')
ax1.set_title('1')
ax2.imshow(test_images[2])
ax2.axis('off')
ax2.set_title('17')
ax3.imshow(test_images[3])
ax3.axis('off')
ax3.set_title('25')
ax4.imshow(test_images[4])
ax4.axis('off')
ax4.set_title('36')
ax5.imshow(test_images[5])
ax5.axis('off')
ax5.set_title('13')
plt.show()
```



1.6.2 Normalize Test Data

```
In [14]: test_images_n = preprocess_normalize(test_images)
```

```
Normalized 6 entries.
```

1.6.3 Accuracy of Test Data

```
In [15]: with tf.Session() as sess:
             sess.run(tf.global_variables_initializer())
             new_saver = tf.train.import_meta_graph('./traffic_sign_lenet.meta')
             new_saver.restore(sess, tf.train.latest_checkpoint('.'))
             test_accuracy = evaluate(test_images_n, test_images_labels)
             print("Test Data Accuracy = {:.3f}".format(test_accuracy))
```

```
Test Data Accuracy = 0.667
```

1.6.4 Output Top 5 Softmax Probabilities For Each Image Found on the Web

For each of the new images, print out the model's softmax probabilities to show the **certainty** of the model's predictions (limit the output to the top 5 probabilities for each image). `tf.nn.top_k` could prove helpful here.

The example below demonstrates how tf.nn.top_k can be used to find the top k predictions for each image.

`tf.nn.top_k` will return the values and indices (class ids) of the top k predictions. So if k=3, for each sign, it'll return the 3 largest probabilities (out of a possible 43) and the correspoding class ids.

Take this numpy array as an example. The values in the array represent predictions. The array contains softmax probabilities for five candidate images with six possible classes. `tk.nn.top_k` is used to choose the three classes with the highest probability:

```
# (5, 6) array
a = np.array([[ 0.24879643,  0.07032244,  0.12641572,  0.34763842,  0.07893497,
         0.12789202],
       [ 0.28086119,  0.27569815,  0.08594638,  0.0178669 ,  0.18063401,
         0.15899337],
       [ 0.26076848,  0.23664738,  0.08020603,  0.07001922,  0.1134371 ,
         0.23892179],
       [ 0.11943333,  0.29198961,  0.02605103,  0.26234032,  0.1351348 ,
         0.16505091],
       [ 0.09561176,  0.34396535,  0.0643941 ,  0.16240774,  0.24206137,
         0.09155967]])
```

Running it through `sess.run(tf.nn.top_k(tf.constant(a), k=3))` produces:

```
TopKV2(values=array([[ 0.34763842,  0.24879643,  0.12789202],
       [ 0.28086119,  0.27569815,  0.18063401],
       [ 0.26076848,  0.23892179,  0.23664738],
       [ 0.29198961,  0.26234032,  0.16505091],
       [ 0.34396535,  0.24206137,  0.16240774]]), indices=array([[3, 0, 5],
       [0, 1, 4],
       [0, 5, 1],
       [1, 3, 5],
       [1, 4, 3]], dtype=int32))
```

Looking just at the first row we get `[ 0.34763842,  0.24879643,  0.12789202]`, you can confirm these are the 3 largest probabilities in a. You'll also notice `[3, 0, 5]` are the corresponding indices.

```
In [18]: ### Print out the top five softmax probabilities for the predictions on the German traf
         ### Feel free to use as many code cells as needed.

         softmax = tf.nn.softmax(logits)
         top_predictions = tf.nn.top_k(softmax, k=5)

         with tf.Session() as sess:
             new_saver = tf.train.import_meta_graph('./traffic_sign_lenet.meta')
             new_saver.restore(sess, tf.train.latest_checkpoint('.'))
             predictions = sess.run(top_predictions, feed_dict={x: test_images_n})
         for k in range(6):
             print('--= Image {0} =--'.format(k+1))
             print('Probabilities - {}'.format(predictions.values[k]))
             print('Indices - {}'.format(predictions.indices[k]))
```

```
--= Image 1 =--
Probabilities - [  1.00000000e+00   4.96904740e-10   1.28131238e-11   5.09963353e-24
   6.03904223e-28]
Indices - [7 5 8 2 1]
--= Image 2 =--
Probabilities - [ 0.33413783  0.30592906  0.23992415  0.03507989  0.0278321 ]
Indices - [11 10  2 40 12]
--= Image 3 =--
Probabilities - [  9.99998927e-01   5.88499972e-07   3.17071311e-07   1.23908805e-07
   1.76504944e-08]
Indices - [17  9 34 14 26]
--= Image 4 =--
Probabilities - [  1.00000000e+00   1.26362280e-36   6.64336937e-38   4.28843096e-38
   0.00000000e+00]
Indices - [25 20 18  1  0]
--= Image 5 =--
Probabilities - [  9.99763072e-01   2.14393571e-04   2.10554917e-05   1.30286264e-06
   1.37908273e-07]
Indices - [25 36 22 26 18]
--= Image 6 =--
Probabilities - [  1.00000000e+00   3.08083587e-25   0.00000000e+00   0.00000000e+00
```

```
   0.00000000e+00]
Indices - [13 25  0  1  2]
```

1.6.5 Project Writeup

Once you have completed the code implementation, document your results in a project writeup using this template as a guide. The writeup can be in a markdown or pdf file.

> **Note**: Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the menu above and navigating to ", "**File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

1.7 Step 4 (Optional): Visualize the Neural Network's State with Test Images

This Section is not required to complete but acts as an additional excersise for understaning the output of a neural network's weights. While neural networks can be a great learning device they are often referred to as a black box. We can understand what the weights of a neural network look like better by plotting their feature maps. After successfully training your neural network you can see what it's feature maps look like by plotting the output of the network's weight layers in response to a test stimuli image. From these plotted feature maps, it's possible to see what characteristics of an image the network finds interesting. For a sign, maybe the inner network feature maps react with high activation to the sign's boundary outline or to the contrast in the sign's painted symbol.

Provided for you below is the function code that allows you to get the visualization output of any tensorflow weight layer you want. The inputs to the function should be a stimuli image, one used during training or a new one you provided, and then the tensorflow variable name that represents the layer's state during the training process, for instance if you wanted to see what the LeNet lab's feature maps looked like for it's second convolutional layer you could enter conv2 as the tf_activation variable.

For an example of what feature map outputs look like, check out NVIDIA's results in their paper End-to-End Deep Learning for Self-Driving Cars in the section Visualization of internal CNN State. NVIDIA was able to show that their network's inner weights had high activations to road boundary lines by comparing feature maps from an image with a clear path to one without. Try experimenting with a similar test to show that your trained network's weights are looking for interesting features, whether it's looking at differences in feature maps from images with or without a sign, or even what feature maps look like in a trained network vs a completely untrained one on the same sign image.

Your output should look something like this (above)

```
In [17]: ### Visualize your network's feature maps here.
         ### Feel free to use as many code cells as needed.

         # image_input: the test image being fed into the network to produce the feature maps
         # tf_activation: should be a tf variable name used during your training procedure that
         # activation_min/max: can be used to view the activation contrast in more detail, by de
         # plt_num: used to plot out multiple different weight feature map sets on the same bloc

         def outputFeatureMap(image_input, tf_activation, activation_min=-1, activation_max=-1 ,
             # Here make sure to preprocess your image_input in a way your network expects
             # with size, normalization, ect if needed
             # image_input =
             # Note: x should be the same name as your network's tensorflow data placeholder var
             # If you get an error tf_activation is not defined it may be having trouble accessi
             activation = tf_activation.eval(session=sess,feed_dict={x : image_input})
             featuremaps = activation.shape[3]
             plt.figure(plt_num, figsize=(15,15))
             for featuremap in range(featuremaps):
                 plt.subplot(6,8, featuremap+1) # sets the number of feature maps to show on eac
                 plt.title('FeatureMap ' + str(featuremap)) # displays the feature map number
                 if activation_min != -1 & activation_max != -1:
                     plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", vmin =ac
                 elif activation_max != -1:
                     plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", vmax=act
                 elif activation_min !=-1:
                     plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", vmin=act
                 else:
                     plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", cmap="gr
```