Udacity Self-Driving Car Nanodegree

Project 2- Traffic Sign Identifier

Author: Johannes Betz

Date: 06/25/2017

Table of Contents

1. Introduction

The whole code described on the next pages can be found in the python file:

Traffic_Sign_Classifier.py

With the described packages and python 3.5.2 the code can be run.

2. Step 1 – Loading the data

The first step was to load the three different data files. There were 3 variables provided, for each variable the path of the data files has to be provided. The folder of the the data files was lying in the same direcetory so the providing the path was just giving the subfolder which looks like this:

```
training_file = "./traffic-signs-data/train.p"
validation_file = "./traffic-signs-data/valid.p"
testing_file = "./traffic-signs-data/test.p"
```

3. Dataset Exploration

In the second step, the dataset should be exploarde, summarized and visualized. 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.

First, a basic evaluation gives and overview of the parameters:

Number of training examples = 34799
Number of validation examples = 4410
Number of testing examples = 12630
Image data shape = (32, 32, 3)
Number of classes = 43

The code which produces this results as listed as seen below:

```
n_train = len(X_train)

n_test = len(X_test)

image_shape = X_train[0].shape

n_classes = len(np.unique(y_train))
```

In the second step, a visualization of the dataset is done. This visualization of the dataset is done with the matplot.lib package which can plot the traffic sign images or the count of each traffic sign.

In my code I did the following:

1. Visualizing 5 random pictures

2. Providing the histogram of each dataset

The Code for visualizing 5 random pictures looks like this:

```
fig, axs = plt.subplots(1,5, figsize=(15, 6))
fig.subplots_adjust(hspace = .01, wspace=.5)
axs = axs.ravel()
for i in range(5):
    index = random.randint(0, len(X_train))
    image = X_train[index]
    axs[i].axis('off')
    axs[i].imshow(image)
    axs[i].set_title(y_train[index])

plt.show()
```

And provides a picture like this

In the second step, a histogram of each dataset provides a visualization of the distribution of each data. The code for doing this looks like this:

```
hist, bins = np.histogram(y_train, bins=n_classes)
width = 0.7 * (bins[1] - bins[0])
center = (bins[:-1] + bins[1:]) / 2
plt.bar(center, hist, align='center', width=width)
plt.show()

hist1, bins1 = np.histogram(y_valid , bins=n_classes)
width = 0.7 * (bins1[1] - bins1[0])
center = (bins1[:-1] + bins1[1:]) / 2
plt.bar(center, hist1, align='center', width=width)
plt.show()

hist2, bins2 = np.histogram(y_test, bins=n_classes)
width = 0.7 * (bins2[1] - bins2[0])
center = (bins2[:-1] + bins2[1:]) / 2
plt.bar(center, hist2, align='center', width=width)
plt.show()
```

and produces pictures like this.

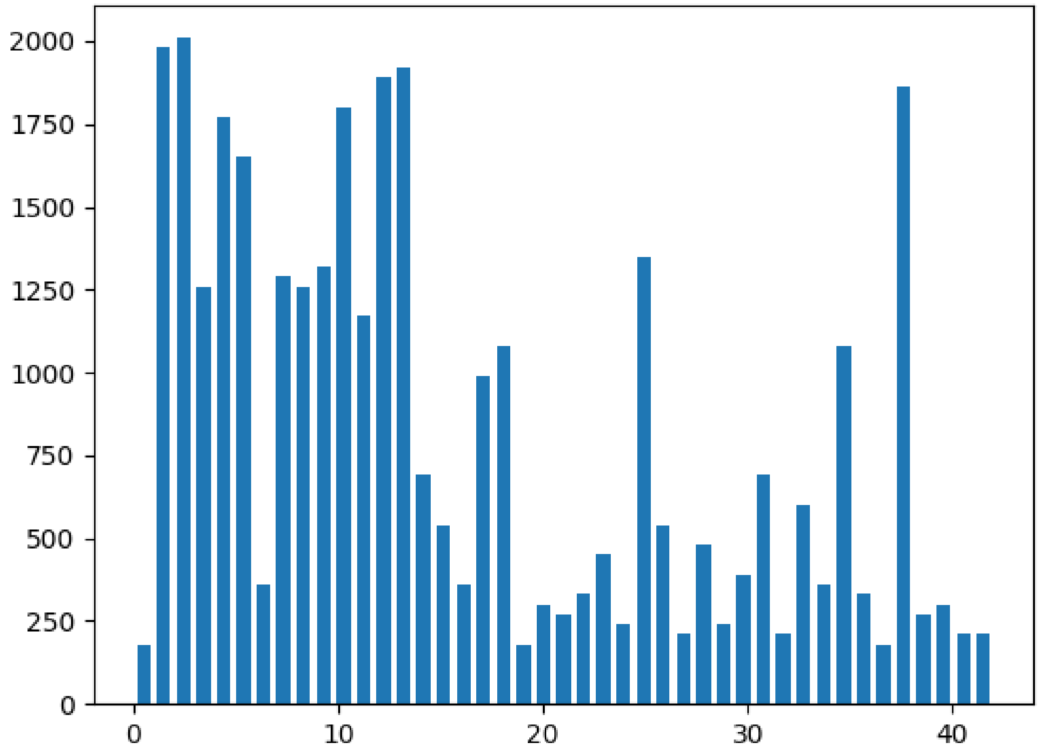

FIgure 1: Histogramm of Train dataset

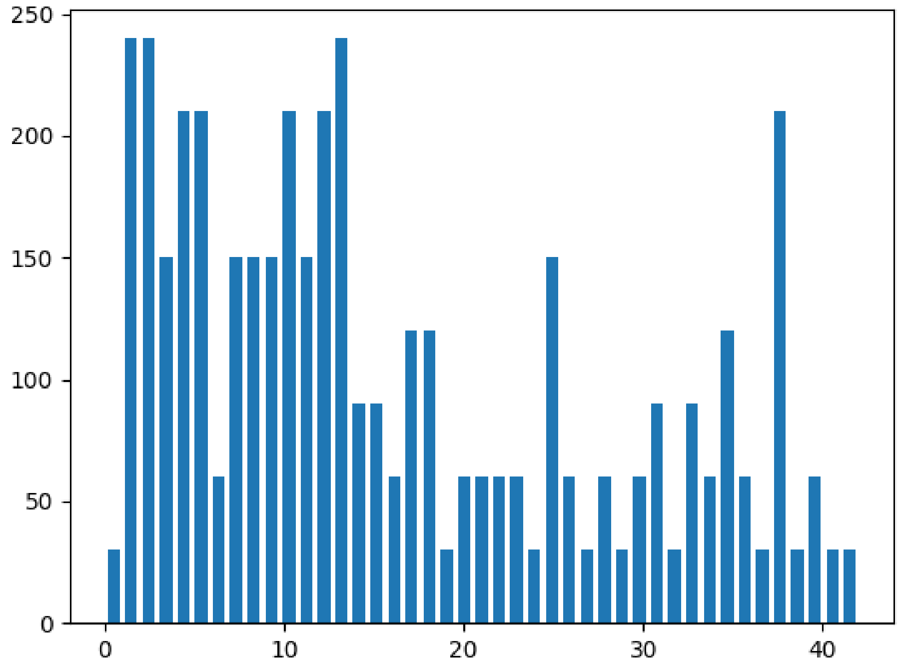

FIgure 2: Histogramm of Valid dataset

4. Design, and Train a NN Model Architecture

In the third step, the goal was to design, train and test a model architecture of a deep learning model that learns to recognize traffic signs. The model is trained and tested on the German Traffic Sign Dataset. For the development of the model we use the LeNet-5 and want to get an set accuracy of about 0.93.

To improve the set accuracy we have to consider various aspects:

- 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.

First, the all the pictures from all the datasets are turned into greyscale, which is indentified as a tuning parameter for reducing the training time.

```
X_train_rgb = X_train
X_train_gry = np.sum(X_train/3, axis=3, keepdims=True)

X_valid_rgb = X_valid
X_valid_gry = np.sum(X_valid/3, axis=3, keepdims=True)

X_test_rgb = X_test
X_test_gry = np.sum(X_test/3, axis=3, keepdims=True)


X_train = X_train_gry
X_test = X_test_gry
X_valid = X_valid_gry

print('RGB shape:', X_train_rgb.shape)
print('Grayscale shape:', X_train_gry.shape)
print('done')
```

Second, all the pictures are normalized. An overall explanation why we should do this can be found here (https://stats.stackexchange.com/questions/185853/why-do-we-need-to-normalize-the-images-before-we-put-them-into-cnn). The normalization has mostly do to with that We'd like in this process for each feature to have a similar range so that our gradients don't go out of control. The normalization is done with this code:

```
print('Mean value of train dataset before normalization:',
np.mean(X_train))
print('Mean value of valid dataset before normalizaion:', np.mean(X_valid))
print('Mean value of test dataset before normalizaion:', np.mean(X_test))

X_train_normalized = (X_train - 128)/128
X_valid_normalized = (X_valid - 128)/128
X_test_normalized = (X_test - 128)/128

print('\nMean value of train dataset before normalizaion:',
np.mean(X_train_normalized))
print('Mean value of valid dataset before normalizaion:',
np.mean(X_valid_normalized))
```

```
print('Mean value of test dataset before normalizaion:',
np.mean(X_test_normalized))
```

At least, the data is shuffled:

```
X_train, y_train = shuffle(X_train, y_train)
```

The next step is implementing the LeNet Architecture, which is the following steps:

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 43 outputs.

The Code for the Model Architecture looks like the following:

```
# Defining the NN paramters, The EPOCH and BATCH_SIZE values affect the
training speed and model accuracy.
EPOCHS = 60             # how many times the training data should be runned
through the network, more epochs, more accuracy, longer time
BATCH_SIZE = 100        # how many Training images should be run through
the NN at a time


def LeNet(x):

    #Hyperparamters: Arguments used for tf.truncated_normal, randomly
defines variables for the weights and biases for each layer
    mu = 0
    sigma = 0.1

    # SOLUTION: Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 6), mean=mu,
stddev=sigma))
    conv1_b = tf.Variable(tf.zeros(6))
    conv1 = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID')
+ conv1_b

    # SOLUTION: Activation.
    conv1 = tf.nn.relu(conv1)

    # SOLUTION: Pooling. Input = 28x28x6. Output = 14x14x6.
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
padding='VALID')

    # SOLUTION: Layer 2: Convolutional. Output = 10x10x16.
    conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16), mean=mu,
stddev=sigma))
```

```
    conv2_b = tf.Variable(tf.zeros(16))
    conv2 = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1],
padding='VALID') + conv2_b

    # SOLUTION: Activation.
    conv2 = tf.nn.relu(conv2)

    # SOLUTION: Pooling. Input = 10x10x16. Output = 5x5x16.
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
padding='VALID')

    # SOLUTION: Flatten. Input = 5x5x16. Output = 400.
    fc0 = flatten(conv2)

    # SOLUTION: Layer 3: Fully Connected. Input = 400. Output = 120.
    fc1_W = tf.Variable(tf.truncated_normal(shape=(400, 120), mean=mu,
stddev=sigma))
    fc1_b = tf.Variable(tf.zeros(120))
    fc1 = tf.matmul(fc0, fc1_W) + fc1_b

    # SOLUTION: Activation.
    fc1 = tf.nn.relu(fc1)

      # Additional Dropout, preventing Overfitting
    fc1 = tf.nn.dropout(fc1, keep_prob)


    # SOLUTION: Layer 4: Fully Connected. Input = 120. Output = 84.
    fc2_W = tf.Variable(tf.truncated_normal(shape=(120, 84), mean=mu,
stddev=sigma))
    fc2_b = tf.Variable(tf.zeros(84))
    fc2 = tf.matmul(fc1, fc2_W) + fc2_b

    # SOLUTION: Activation.
    fc2 = tf.nn.relu(fc2)

    # SOLUTION: Layer 5: Fully Connected. Input = 84. Output = 43.
    fc3_W = tf.Variable(tf.truncated_normal(shape=(84, 43), mean=mu,
stddev=sigma))
    fc3_b = tf.Variable(tf.zeros(43))
    logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits
```

With this code, the NN can be trained. The Code for the training pipeline is also made from the LeNet Architecture but is additionally labeld with comments for the explanation of each program code line. In addition, the pipepline for the evaluation is included, too

```
####################                   3.3 Training Pipeline
###################

# Defining the variables for Tensorflow
x = tf.placeholder(tf.float32, (None, 32, 32, 1))       # Placeholder that
will store the input badges, initilaize badge size to None, image dimension
32x32x2
y = tf.placeholder(tf.int32, (None))                    # Placeholder that
will store labels, Labels are Integers
one_hot_y = tf.one_hot(y, 43)                           # One hot encoding
the labels
```

```
# Setup the Training Pipeline
rate = 0.001
# Learning Rate of the NN, show how quickly to update the weights

logits = LeNet(x)
# Input Data will be passed in the LeNet Function to calculate the logits
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y,
logits=logits)      # Cross-Entropy: Compare the logits to the ground truth
labels - Measurement how different the logits from the ground truth
loss_operation = tf.reduce_mean(cross_entropy)
# Averages the cross entropy
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
# Gradient descent: minimizes the loss function
training_operation = optimizer.minimize(loss_operation)
# Run the minimizer to the optimizer: Backpropagation to reduce the
trainings loss

####################                   3.4 Evaluation Pipeline
###################

# Evaluate how well the loss and accuracy of the model for a given dataset.
How good is the Model
correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_y,
1))     # Measure if a given prediction is correct: comparing logit
prediction to one hot encoded
accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction,
tf.float32))     # Calculate the models overall accuracy by averaging the
individual prediction accuracy
saver = tf.train.Saver()

# The evaluate function: Takes one dataset as an input, sets initial
variables, batches the dataset and runs it trough the pipeline
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_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y:
batch_y})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

After setting up the Pipline for the Training, the actual Training of the NN can be done. The following code line show the implementation of the training of the NN:

```
####################                   3.5 Training the Model
###################

print("\n4. NN Model: Train the Model")

with tf.Session() as sess:
# Create the Tensorflow Session
    sess.run(tf.global_variables_initializer())
# Initialize the variables
    num_examples = len(X_train)
```

```
    print("Training...")
    print()

    start = time.time()
    for i in range(EPOCHS):
        X_train, y_train = shuffle(X_train, y_train)
# We shuffle the data so it isnt biased by the images
        for offset in range(0, num_examples, BATCH_SIZE):
# The training data is broken into batches and the training is done for
each batch
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train[offset:end], y_train[offset:end]
            sess.run(training_operation, feed_dict={x: batch_x, y: batch_y,
keep_prob: 0.5})

        validation_accuracy = evaluate(X_valid, y_valid)                    #
At the end of each epoch, we evaluate the model with the validation data
        print("EPOCH {} ...".format(i + 1))
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print()

    saver.save(sess, './lenet')
    print("Model saved")
    end = time.time()
    print("Time for Training the model:", (end - start),"s")
```

With this code, different evaluations for different paramters are done. The solution for figuring out an accuracy >0.93 is found in different adjustmenst:

1. Greyscaling the data
2. Normalizing the data
3. Adjusting the paramters:
    a. batch size: 100
    b. epochs: 60
    c. learning rate: 0.0009
    d. mu: 0
    e. sigma: 0.1
4. Integration of an additional dropout in the LeNet function with the following probability:
    a. dropout keep probability: 0.5
5. Using the Adam Optimizer (already described in the LeNet Architecture)

With an epoch Size of 60 the following accuracy increasement could be provided:

EPOCH 1 ...

Validation Accuracy = 0.510

EPOCH 2 ...

Validation Accuracy = 0.790

EPOCH 3 ...

Validation Accuracy = 0.839

EPOCH 4 ...

Validation Accuracy = 0.866

EPOCH 5 ...

Validation Accuracy = 0.893

EPOCH 6 ...

Validation Accuracy = 0.905

EPOCH 7 ...

Validation Accuracy = 0.914

EPOCH 8 ...

Validation Accuracy = 0.915

EPOCH 9 ...

Validation Accuracy = 0.927

EPOCH 10 ...

Validation Accuracy = 0.925

EPOCH 11 ...

Validation Accuracy = 0.931

EPOCH 12 ...

Validation Accuracy = 0.931

EPOCH 13 ...

Validation Accuracy = 0.940

EPOCH 14 ...

Validation Accuracy = 0.940

EPOCH 15 ...

Validation Accuracy = 0.939

EPOCH 16 ...

Validation Accuracy = 0.947

EPOCH 17 ...

Validation Accuracy = 0.947

EPOCH 18 ...

Validation Accuracy = 0.950

EPOCH 19 ...

Validation Accuracy = 0.948

EPOCH 20 ...

Validation Accuracy = 0.955

EPOCH 21 ...

Validation Accuracy = 0.961

EPOCH 22 ...

Validation Accuracy = 0.953

EPOCH 23 ...

Validation Accuracy = 0.958

EPOCH 24 ...

Validation Accuracy = 0.956

EPOCH 25 ...

Validation Accuracy = 0.955

EPOCH 26 ...

Validation Accuracy = 0.961

EPOCH 27 ...

Validation Accuracy = 0.963

EPOCH 28 ...

Validation Accuracy = 0.960

EPOCH 29 ...

Validation Accuracy = 0.962

EPOCH 30 ...

Validation Accuracy = 0.956

EPOCH 31 ...

Validation Accuracy = 0.955

EPOCH 32 ...

Validation Accuracy = 0.953

EPOCH 33 ...

Validation Accuracy = 0.963

EPOCH 34 ...

Validation Accuracy = 0.965

EPOCH 35 ...

Validation Accuracy = 0.963

EPOCH 36 ...

Validation Accuracy = 0.968

EPOCH 37 ...

Validation Accuracy = 0.961

EPOCH 38 ...

Validation Accuracy = 0.964

EPOCH 39 ...

Validation Accuracy = 0.957

EPOCH 40 ...

Validation Accuracy = 0.962

EPOCH 41 ...

Validation Accuracy = 0.963

EPOCH 42 ...

Validation Accuracy = 0.961

EPOCH 43 ...

Validation Accuracy = 0.962

EPOCH 44 ...

Validation Accuracy = 0.959

EPOCH 45 ...

Validation Accuracy = 0.961

EPOCH 46 ...

Validation Accuracy = 0.968

EPOCH 47 ...

Validation Accuracy = 0.968

EPOCH 48 ...

Validation Accuracy = 0.964

EPOCH 49 ...

Validation Accuracy = 0.969

EPOCH 50 ...

Validation Accuracy = 0.966

EPOCH 51 ...

Validation Accuracy = 0.962

EPOCH 52 ...

Validation Accuracy = 0.964

EPOCH 53 ...

Validation Accuracy = 0.969

EPOCH 54 ...

Validation Accuracy = 0.968

EPOCH 55 ...

Validation Accuracy = 0.965

EPOCH 56 ...

Validation Accuracy = 0.966

EPOCH 57 ...

Validation Accuracy = 0.961

EPOCH 58 ...

Validation Accuracy = 0.972

EPOCH 59 ...

Validation Accuracy = 0.968

EPOCH 60 ...

Validation Accuracy = 0.969

5. Test a NN Model Architecture

The last step is testing the NN model with the test data set to evaluate how good the model is to new data. The following code uses the provided test data set to get the accuracy.

```
####################                    3.6  Test the Model
###################

print("\n5. NN Model: Test the Model")

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver2 = tf.train.import_meta_graph('./lenet.meta')
    saver2.restore(sess, "./lenet")
    test_accuracy = evaluate(X_test, y_test)
    print("Test Set Accuracy = {:.3f}".format(test_accuracy))
```

With the trained NN in the test data set and accuracy of Test Set Accuracy = 0.942 could be reached

6. Test the trained NN model and new images never seen before

Now we have a trained, validated and tested convolutional NN that can be used to detect and classify traffic signs. We are using this NN now to detect new images

First, new images have to be acquired. These images can be found on the web, in this case I took them from https://github.com/jeremy-shannon/CarND-Traffic-Sign-Classifier-Project/blob/master/Traffic_Sign_Classifier.ipynb to have a better comparison. The code for including the new pictures looks like this and includes the greyscaling and normalizing of the data:

```
my_images = []
my_labels = [3, 11, 1, 12, 38, 34, 25]

for i, img in enumerate(glob.glob('./new-traffic-signs/*x.png')):
    image = cv2.imread(img)
    my_images.append(image)

my_images = np.asarray(my_images)

my_images_gry = np.sum(my_images/3, axis=3, keepdims=True)

my_images_normalized = (my_images_gry - 128)/128
```

After that, an accuracy prediction test of the new images is made with the following code:

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver3 = tf.train.import_meta_graph('./lenet.meta')
    saver3.restore(sess, "./lenet")
    my_accuracy = evaluate(my_images_normalized, my_labels)
    print("New Images - Set Accuracy = {:.3f}".format(my_accuracy))
```

The accuracy of the predicted pictures is at 85%, which is 10% lower then the test-set accuracy of 94%. This can b

The last part of the performance test is comparing the softmax probabilities. For each of the new images, the model's softmax probabilities are printed to show the **certainty** of the model's predictions. The code for doing this is made on the basis of the tf.nn.to_k function and is shown as below.

```
softmax_logits = tf.nn.softmax(logits)
top_k = tf.nn.top_k(softmax_logits, k=3)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver = tf.train.import_meta_graph('./lenet.meta')
    saver.restore(sess, "./lenet")
    my_softmax_logits = sess.run(softmax_logits, feed_dict={x:
my_images_normalized, keep_prob: 1.0})
    my_top_k = sess.run(top_k, feed_dict={x: my_images_normalized,
keep_prob: 1.0})

    fig, axs = plt.subplots(len(my_images), 4, figsize=(12, 14))
    fig.subplots_adjust(hspace=.4, wspace=.2)
    axs = axs.ravel()

    for i, image in enumerate(my_images):
        axs[4 * i].axis('off')
        axs[4 * i].imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
        axs[4 * i].set_title('input')
        guess1 = my_top_k[1][i][0]
        index1 = np.argwhere(y_valid == guess1)[0]
        axs[4 * i + 1].axis('off')
        axs[4 * i + 1].imshow(X_valid[index1].squeeze(), cmap='gray')
        axs[4 * i + 1].set_title('top guess: {} ({:.0f}%)'.format(guess1,
100 * my_top_k[0][i][0]))
        guess2 = my_top_k[1][i][1]
        index2 = np.argwhere(y_valid == guess2)[0]
        axs[4 * i + 2].axis('off')
        axs[4 * i + 2].imshow(X_valid[index2].squeeze(), cmap='gray')
        axs[4 * i + 2].set_title('2nd guess: {} ({:.0f}%)'.format(guess2,
100 * my_top_k[0][i][1]))
        guess3 = my_top_k[1][i][2]
        index3 = np.argwhere(y_valid == guess3)[0]
        axs[4 * i + 3].axis('off')
        axs[4 * i + 3].imshow(X_valid[index3].squeeze(), cmap='gray')
        axs[4 * i + 3].set_title('3rd guess: {} ({:.0f}%)'.format(guess3,
100 * my_top_k[0][i][2]))
    plt.show()
```

As a results the seven new integrated pictures are compared to the top guess, the 2nd guess and the 3rd guess. In addition, the softmax probability is shown

Improvements:

I wonder why my start accuracy of the model is really low. I compared it to other NN and they are all starting at 80 % or more. I think this is the reason, why the accuracy prediction of the new images is just 85%. I Think additional data for some images could help here. I varied the Hyperparameters a lot but couldn't figure out a big improvement in the end so I think the biggest improvement must be in more data.