VW581 HW2 report

I run the whole program in the Google Colab.

P2. Data Exploration

In Data Exploration process, first I output the number of samples in each set, shape of the traffic image and number of classed/labels.

Code:

```
# ====== i.Number of training set ======
n_train = X_train.shape[0]

# ====== Number of validation set ======
n_validation = X_valid.shape[0]

# ====== Number of testing set ======
n_test = X_test.shape[0]

# ====== ii.Shape of traffic image ======
image_shape = X_train[0].shape

# ====== iii.Number of classes/labels ======
n_classes = len(np.unique(y_train))
```

Results:

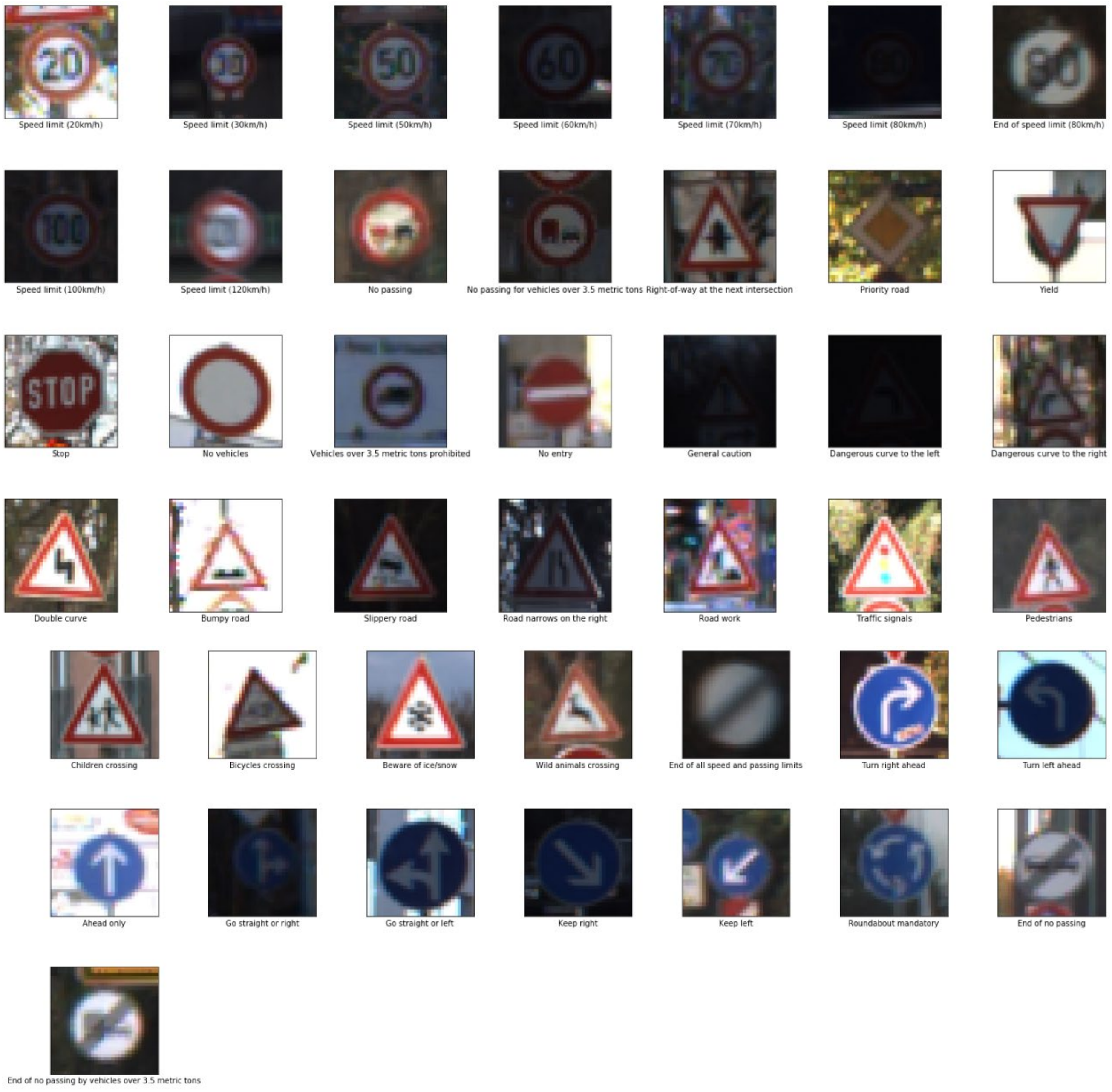
```
Number of training set:  34799
Number of validation set:  4410
Number of testing set:  12630
Image data shape = (32, 32, 3)
Number of classes = 43
```

Then, I plot a sample image for each label chosen from training set.

Code:

```
# ====== i.Plot a sample image for each class/label ======
def list_images(dataset, dataset_y, ylabel="", cmap=None):
    plt.figure(figsize=(20, 20))
    for i in range(43):
        plt.subplot(7, 7, i+1)
        indx = -1
        while True:
            indx = random.randint(0, len(dataset)-1)
            if dataset_y[indx]==i:
                cmap = 'gray' if len(dataset[indx].shape) == 2 else cmap
                plt.imshow(dataset[indx], cmap = cmap)
                plt.xlabel(signs[dataset_y[indx]])
                plt.xticks([])
                plt.yticks([])
            break
    plt.tight_layout(pad=0, h_pad=0, w_pad=0)
    plt.show()

print('These sample images for each class are chosen in training set')
list_images(X_train, y_train, "Training example")
```

Results:



P3. Design and Test a Classifier

a. Preprocess.

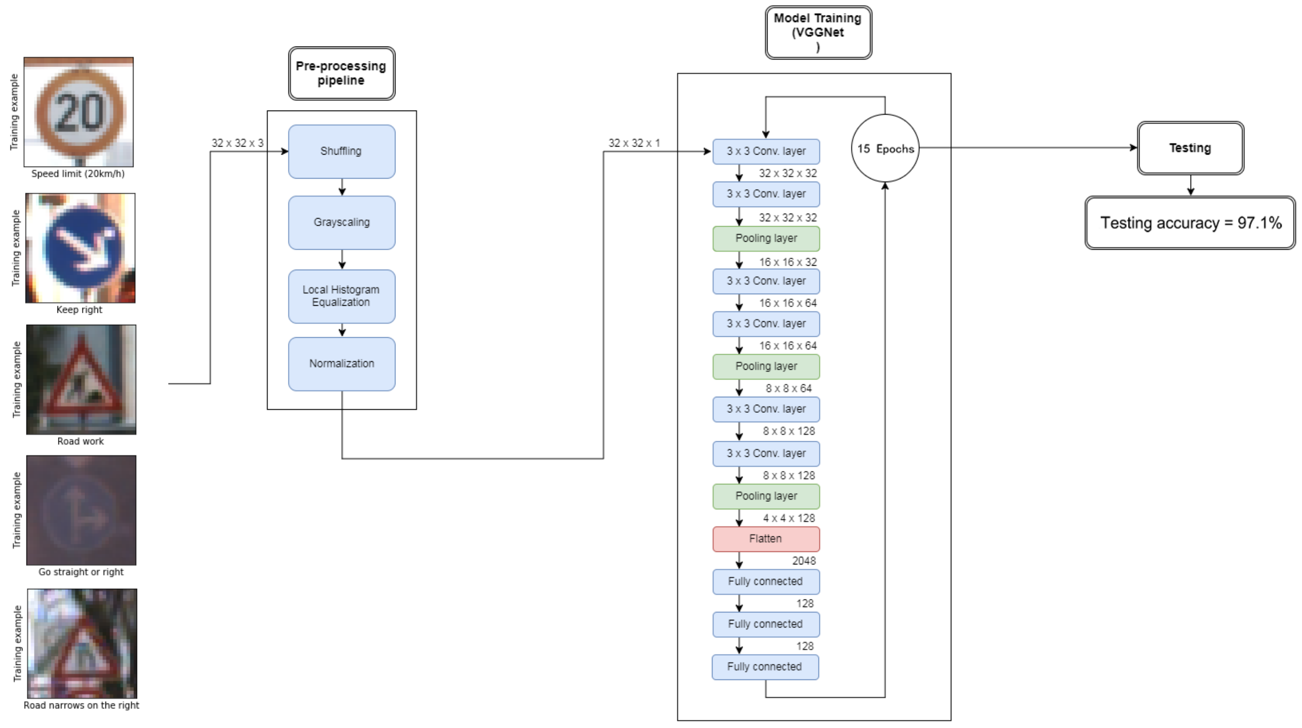
We will use the following preprocessing techniques and these are reasons:

- Shuffling: In general, we shuffle the training data to increase randomness and variety in training dataset, in order for the model to be more stable.

- Grayscaling: Using grayscale images instead of color improves the ConvNet's accuracy.
- Local Histogram Equalization: This technique simply spreads out the most frequent intensity values in an image, resulting in enhancing images with low contrast. Applying this technique will be very helpful in case since the dataset in hand has real world images, and many of them has low contrast.
- Normalization: Normalization is a process that changes the range of pixel intensity values. Usually the image data should be normalized so that the data has mean zero and equal variance.

b. Design model and show its architecture

I use VGGNet as model, and below is a simple schematic diagram. I do not show the code in report due to space.



c. Compile the model:

We minimize the loss function using the Adaptive Moment Estimation (Adam) Algorithm and we will run minimize() function on the optimizer which use backpropagation to update the network and minimize our training loss.

```
self.loss_operation = tf.reduce_mean(self.cross_entropy)
self.optimizer = tf.train.AdamOptimizer(learning_rate = learning_rate)
self.training_operation = self.optimizer.minimize(self.loss_operation)

EPOCHS = 30
BATCH_SIZE = 64
```

d. Train your model, and plot your training history

Figure below is the training history:

```
Training...

EPOCH 1 : Validation Accuracy = 16.825%
EPOCH 2 : Validation Accuracy = 44.036%
EPOCH 3 : Validation Accuracy = 70.771%
EPOCH 4 : Validation Accuracy = 86.145%
EPOCH 5 : Validation Accuracy = 90.658%
EPOCH 6 : Validation Accuracy = 94.399%
EPOCH 7 : Validation Accuracy = 95.714%
EPOCH 8 : Validation Accuracy = 97.279%
EPOCH 9 : Validation Accuracy = 97.823%
EPOCH 10 : Validation Accuracy = 98.118%
EPOCH 11 : Validation Accuracy = 98.277%
EPOCH 12 : Validation Accuracy = 97.664%
EPOCH 13 : Validation Accuracy = 98.322%
EPOCH 14 : Validation Accuracy = 98.617%
EPOCH 15 : Validation Accuracy = 98.844%
```

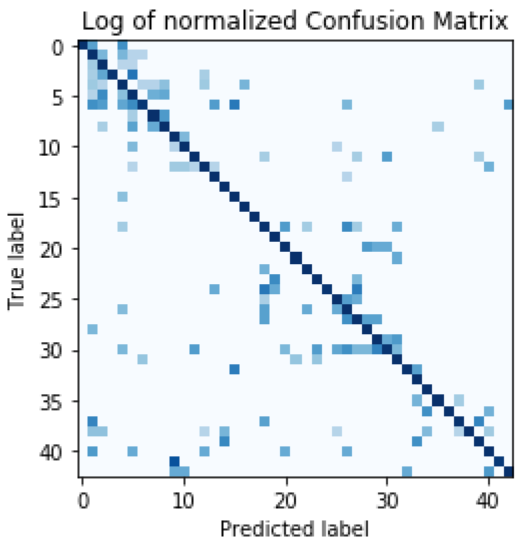
e. Tune your model until validation accuracy higher than 95%

Using VGGNet, we've been able to reach a maximum validation accuracy of 98.844%.

f. Evaluate model's performance and make predictions on test images.

Now, we'll use the testing set to measure the accuracy of the model over unknown examples. The test accuracy equals to 96.7% and is a pretty good result. And I plot the confusion matrix to see where the model actually fails.

```
test Accuracy = 96.7%
```



We can observe some clusters in the confusion matrix above. It turns out that the various speed limits are sometimes misclassified among themselves. Similarly, traffic signs with triangular shape are misclassified among themselves. We can further improve on the model using hierarchical CNNs to first identify broader groups (like speed signs) and then have CNNs to classify finer features (such as the actual speed limit).

P4. Test Your Classifier on New Images

a&b:

Code:

In this step, we will use the model to predict traffic signs type of 10 random images of German traffic signs from the web our model's performance on these images.

```
# Loading and resizing new test images
new_test_images = []
path = '/content/drive/My Drive/new_images/'
for image in os.listdir(path):
    print(image)
    img = cv2.imread(path + image)
    img = cv2.resize(img, (32,32))
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    new_test_images.append(img)
new_IDs = [27,14,17,2,13,38,33,34,15,35]
print("Number of new testing examples: ", len(new_test_images))
```

Displaying the new testing examples, with their respective ground-truth labels:

```
plt.figure(figsize=(20,20))
for i in range(len(new_test_images)):
    plt.subplot(2, 5, i+1)
    plt.imshow(new_test_images[i])
    plt.xlabel(signs[new_IDs[i]])
    plt.ylabel("New testing image")
    plt.xticks([])
    plt.yticks([])
plt.tight_layout(pad=0, h_pad=0, w_pad=0)
plt.show()
```

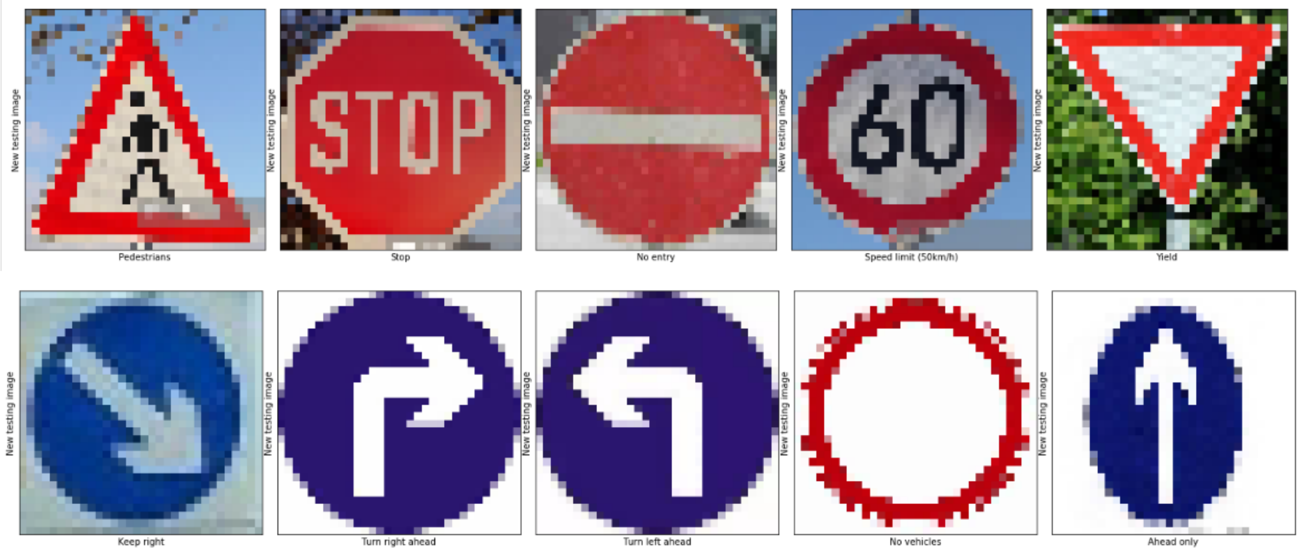
These test images include some easy to predict signs, and other signs are considered hard for the model to predict.

For instance, we have easy to predict signs like the "Stop" and the "No entry". The two signs are clear and belong to classes where the model can predict with high accuracy.

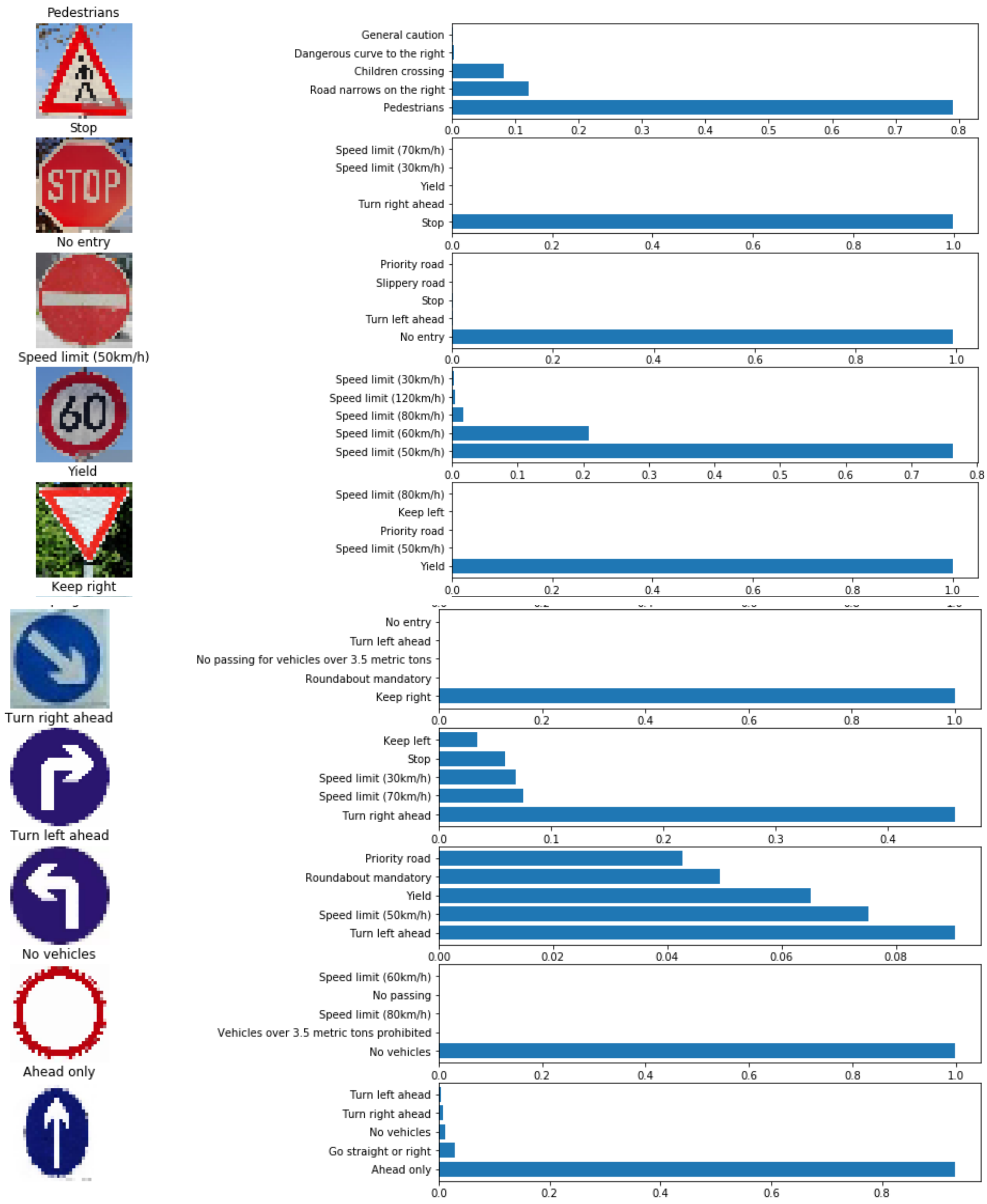
On the other hand, we have signs belong to classes where has poor accuracy, like the "Speed limit" sign, because as stated above it turns out that the various speed limits are sometimes misclassified among themselves, and the "Pedestrians" sign, because traffic signs with triangular shape are misclassified among themselves.

```
new_test_images_preprocessed = preprocess(np.asarray(new_test_images))

def y_predict_model(Input_data, top_k=5):
    num_examples = len(Input_data)
    y_pred = np.zeros((num_examples, top_k), dtype=np.int32)
    y_prob = np.zeros((num_examples, top_k))
    with tf.Session() as sess:
        VGGNet_Model.saver.restore(sess, os.path.join(DIR, "VGGNet"))
        y_prob, y_pred =
sess.run(tf.nn.top_k(tf.nn.softmax(VGGNet_Model.logits), k=top_k),
                        feed_dict={x:Input_data, keep_prob:1,
keep_prob_conv:1})
    return y_prob, y_pred

y_prob, y_pred = y_predict_model(new_test_images_preprocessed)
print('length=',len(new_test_images_preprocessed))

test_accuracy = 0
for i in enumerate(new_test_images_preprocessed):
    accu = new_IDs[i[0]] == np.asarray(y_pred[i[0]])[0]
    if accu == True:
        test_accuracy += 0.1
print("New Images Test Accuracy = {:.1f}%".format(test_accuracy*100))

plt.figure(figsize=(20, 20))
for i in range(len(new_test_images_preprocessed)):
    plt.subplot(10, 2, 2*i+1)
    plt.imshow(new_test_images[i])
    plt.title(signs[y_pred[i][0]])
    plt.axis('off')
    plt.subplot(10, 2, 2*i+2)
    plt.barh(np.arange(1, 6, 1), y_prob[i, :])
    labels = [signs[j] for j in y_pred[i]]
    plt.yticks(np.arange(1, 6, 1), labels)
plt.show()
```
Results:

```
Number of new testing examples:  10
```



```
New Images Test Accuracy = 100.0%
```



As we can notice from the top 5 softmax probabilities, the model has very high confidence (100%) when it comes to predict simple signs, like the "Stop" and the "No entry" sign, and even high confidence when predicting simple triangular signs in a very clear image, like the "Yield" sign.

P5. Not finished