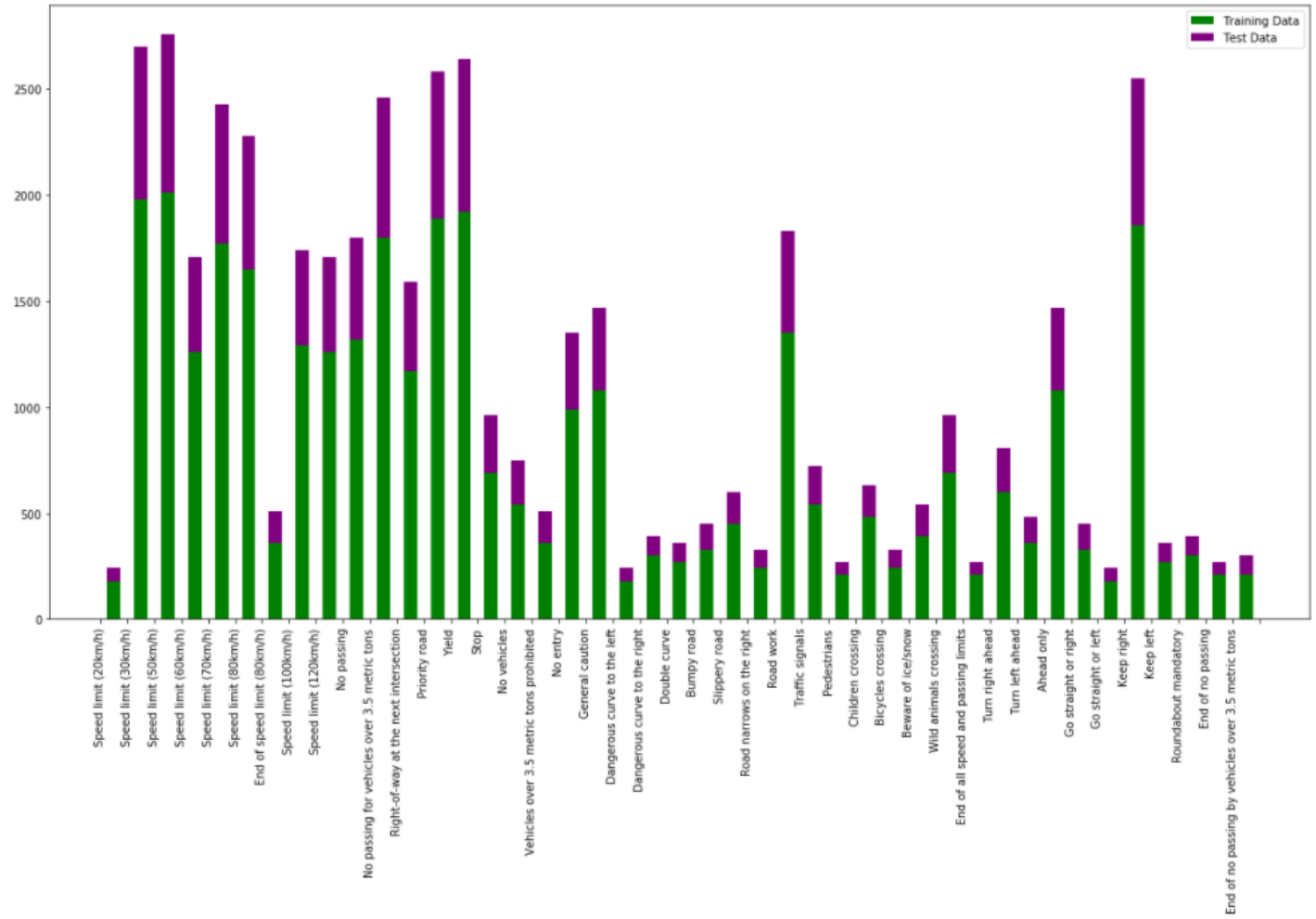
Data Set Summary and Exploration

Summary of data shown below:

```
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, 3)
Number of classes = 43
```

A histogram is plotted to show the label distribution:



Design and Test a Model Architecture

The steps taken to preprocess the images are:

1. Grayscale via averaging the pixel color data from all three layers:

```
import numpy as np
X_train_gray = np.sum(X_train/3, axis = 3, keepdims = True)
X_test_gray = np.sum(X_test/3, axis = 3, keepdims = True)

print("gray train shape is", X_train_gray.shape)
print("gray test shape is", X_test_gray.shape)
```

```
gray train shape is (34799, 32, 32, 1)
gray test shape is (12630, 32, 32, 1)
```

2. Normalize via the method suggested by the comment:

```
#normalizing grayscaled images
X_train_gray = (X_train_gray - 128)/128
X_test_gray = (X_test_gray - 128)/128

#re-assigning gray to training data (in order to keep subsequent code identical)
X_train = X_train_gray
X_test = X_test_gray
```

Subsequently, grayscaling and normalization of the test and validation data sets are done using the grayscale and normalize function:

```
def grayscale_normalize(x):
    result = np.sum(x/3, axis = 3, keepdims = True)
    result = (result - 128)/128
    return result
```

Images are grayscaled and normalized because it rids of the added factors of colors, which may result in overfitting the model to training data set.

Taking the color away is a way to force the model to learn redundancy when it comes to learning.

Normalizing the data shifts the mean towards zero. This reduces error.

Architecture info (LeNet)

```
x = tf.nn.bias_add(tf.nn.conv2d(x, weights['wc1'], strides = [1, 1, 1, 1], padding='VALID'), biases['bc1'])


# TODO: Activation.
x = tf.nn.relu(x)


# TODO: Pooling. Input = 28x28x6. Output = 14x14x6.
x = tf.nn.max_pool(x, ksize = [1, 2, 2, 1], strides = [1, 2, 2, 1], padding = 'VALID')
# TODO: Layer 2: Convolutional. Output = 10x10x16.
x = tf.nn.bias_add(tf.nn.conv2d(x, weights['wc2'], strides = [1, 1, 1, 1], padding = 'VALID'), biases['bc2'])
# TODO: Activation.
x = tf.nn.relu(x)
# TODO: Pooling. Input = 10x10x16. Output = 5x5x16.
x = tf.nn.max_pool(x, ksize = [1, 2, 2, 1], strides = [1, 2, 2, 1], padding = 'VALID')
# TODO: Flatten. Input = 5x5x16. Output = 400.
x = flatten(x)
# TODO: Layer 3: Fully Connected. Input = 400. Output = 120.
x = tf.nn.bias_add(tf.matmul(x, weights['wd3']), biases['bd3'])
# TODO: Activation.
x = tf.nn.relu(x)
# TODO: Layer 4: Fully Connected. Input = 120. Output = 84.
x = tf.nn.bias_add(tf.matmul(x, weights['wd4']), biases['bd4'])
# TODO: Activation.
x = tf.nn.relu(x)
# TODO: Layer 5: Fully Connected. Input = 84. Output = 43.
x = tf.nn.bias_add(tf.matmul(x, weights['wd5']), biases['bd5'])
logits = x
return logits
```

Hyperparameters include:

- Mean and standard deviation of the random weights at first forward prop.
- Number of EPOCHs
- Learn rate
- Batch size

Mean and standard deviations were unchanged as it had very little to do with model accuracy after the first forward prop (which is mostly luck anyway).

The EPOCHs, learn rate and batch size were adjusted through largely trial and error and the best were found to be 30, 0.001, and 130 respectively.

The road sign prediction accuracies were 0.8