bedu2806final

September 17, 2024

1 Beckett's Final Project INFO 4604

1.1 Autonomous Traffic Light Detection System (ATLDS)

1.2 Using TF Multi-layer and LR

[5]:
```
import random # for shuffling
import glob # for importing images by file name
import numpy as np # for making arrays
import matplotlib.pyplot as plt # for plotting
import matplotlib.image as mpimg # for loading in images
import tensorflow as tf # for algorithm 1 multi layer

%matplotlib inline
```

1.2.1 1. EDA

[6]:
```
labels = []
traffic = []

# defining two lists to store images and labels
```

[7]:
```
import cv2
def standardize_input(image):
    image_crop = np.copy(image)
    row_crop = 7
    col_crop = 8
    image_crop = image[row_crop:-row_crop, col_crop:-col_crop, :]
    ## TODO: Resize image and pre-process so that all "standard" images are the␣
 ↪same size
    standard_im = cv2.resize(image_crop, (32, 32))
    return standard_im

# defining an image standardizer to crop the images to a standard of 32x32
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[7], line 1
```

```
----> 1 import cv2
      2 def standardize_input(image):
      3     image_crop = np.copy(image)

ModuleNotFoundError: No module named 'cv2'
```

[4]:
```
for file in glob.glob("trafficLightImages/training/green/*"):

    # Read in the image
    im = mpimg.imread(file)
    std_img = standardize_input(im)
    label = 0

    # Append the image to the green image list
    traffic.append(std_img)
    labels.append(label)
# using glob, this iterates through my files,
```

[5]:
```
for file in glob.glob("trafficLightImages/training/yellow/*"):

    # Read in the image
    im = mpimg.imread(file)
    std_img = standardize_input(im)
    label = 1

    # Append the image to the yellow image list
    traffic.append(std_img)
    labels.append(label)
```

[6]:
```
for file in glob.glob("trafficLightImages/training/red/*"):

    # Read in the image
    im = mpimg.imread(file)
    std_img = standardize_input(im)
    label = 2

    # Append the image to the red image list
    traffic.append(std_img)
    labels.append(label)
```

[7]:
```
class_names = ['green','yellow','red']
# assign class names for later use in predictions
```

1.2.2 2. Preprocessing

[8]:
```
from sklearn.model_selection import train_test_split

image_train, image_test, labels_train, labels_test = train_test_split(traffic,␣
 ↪labels, test_size=0.2,
                                                                              ␣
 ↪random_state=42, stratify=labels)

# from scikit, I split the images into train and test sets, 80/20 split.
```

[9]:
```
image_train = np.array(image_train)
image_test = np.array(image_test)
labels_train = np.array(labels_train)
labels_test = np.array(labels_test)
# transforming the lists into numpy arrays
```

[10]:
```
image_train.shape
# 959 refers to the amount of images
# 32,32 are the height and length of the images cropped by the standardizer
# 3 stores the rgb information, 3 for red, green, and blue
```

[10]:
```
(959, 32, 32, 3)
```

[11]:
```
image_test.shape
```

[11]:
```
(240, 32, 32, 3)
```

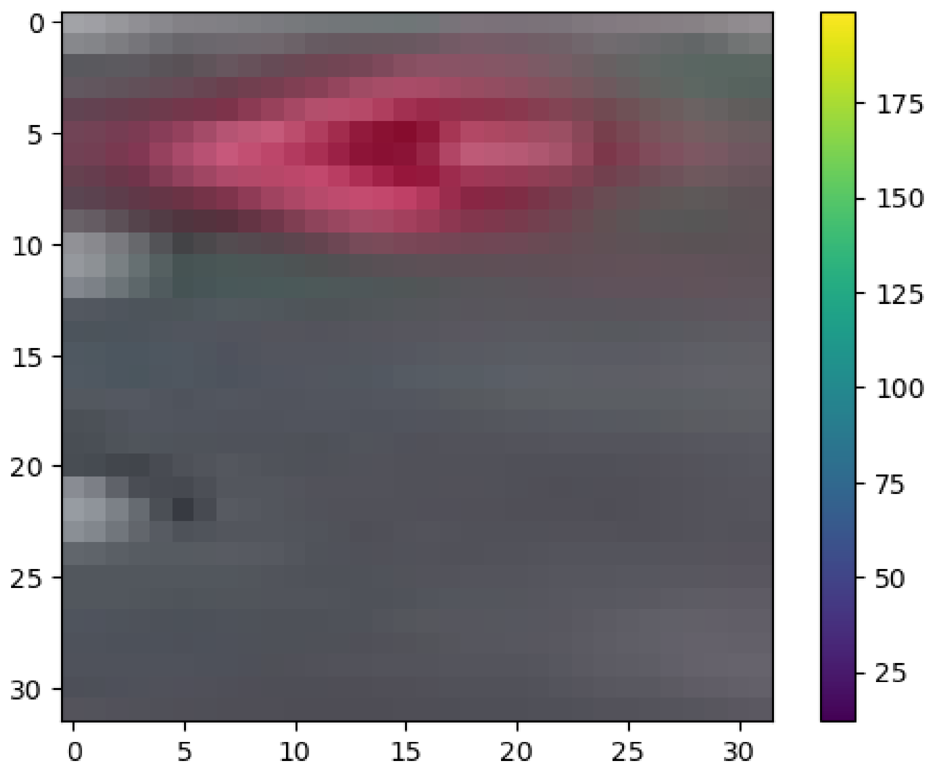
[12]:
```
labels_train.shape
# 1 label for every image
```

[12]:
```
(959,)
```

[13]:
```
labels_test.shape
```

[13]:
```
(240,)
```

[14]:
```
plt.figure()
plt.imshow(image_train[0])
plt.colorbar()
plt.grid(False)
plt.show()
```



[15]:
```
image_train = image_train / 200.0
image_test = image_test / 200.0

# the images fall within 200 pixels so the train and test sets are processed␣
↪the same way to ensure uniformity
```

1.3 TF Multi-layer

1.3.1 3. Build/Train the Model

[16]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(32, 32, 3)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(3)
])

# for tensorflow, the model flattens the arrays into the 32x32 pixel size and␣
↪keeps the rgb data
# the first layer of neurons in the network are set at 128 and activation␣
↪specifys the activation function to be rectified linear unit
# the second is set to 3 because there are 3 classes so 0-3 or 0,1,2.
```

[17]:
```
model.compile(optimizer='adam',
              loss=tf.keras.losses.
 ↪SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

# optimizer specifys how the model is updated
# loss specifys the loss function to tell the model if it is classifying␣
 ↪correctly
# metrics specifys the accuracy score as a metric for performance
```

[18]:
```
training_history = model.fit(image_train, labels_train, epochs=50)
# the model is fit and the training sets are inputted, epochs set to 50, but␣
 ↪will be tested later on
```

```
Epoch 1/50
30/30 [==============================] - 1s 6ms/step - loss: 0.5352 - accuracy:
0.8488
Epoch 2/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0893 - accuracy:
0.9708
Epoch 3/50
30/30 [==============================] - 0s 9ms/step - loss: 0.0511 - accuracy:
0.9885
Epoch 4/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0454 - accuracy:
0.9885
Epoch 5/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0710 - accuracy:
0.9781
Epoch 6/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0446 - accuracy:
0.9854
Epoch 7/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0391 - accuracy:
0.9885
Epoch 8/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0311 - accuracy:
0.9885
Epoch 9/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0296 - accuracy:
0.9927
Epoch 10/50
30/30 [==============================] - 0s 6ms/step - loss: 0.0315 - accuracy:
0.9906
Epoch 11/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0201 - accuracy:
0.9948
```

```
Epoch 12/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0401 - accuracy:
0.9896
Epoch 13/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0209 - accuracy:
0.9927
Epoch 14/50
30/30 [==============================] - 0s 6ms/step - loss: 0.0350 - accuracy:
0.9927
Epoch 15/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0412 - accuracy:
0.9864
Epoch 16/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0547 - accuracy:
0.9791
Epoch 17/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0373 - accuracy:
0.9844
Epoch 18/50
30/30 [==============================] - 0s 8ms/step - loss: 0.0182 - accuracy:
0.9958
Epoch 19/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0135 - accuracy:
0.9948
Epoch 20/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0287 - accuracy:
0.9864
Epoch 21/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0100 - accuracy:
0.9969
Epoch 22/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0120 - accuracy:
0.9958
Epoch 23/50
30/30 [==============================] - 0s 9ms/step - loss: 0.0098 - accuracy:
0.9969
Epoch 24/50
30/30 [==============================] - 0s 5ms/step - loss: 0.0080 - accuracy:
0.9990
Epoch 25/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0150 - accuracy:
0.9937
Epoch 26/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0355 - accuracy:
0.9823
Epoch 27/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0223 - accuracy:
0.9917
```

```
Epoch 28/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0089 - accuracy:
0.9979
Epoch 29/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0534 - accuracy:
0.9791
Epoch 30/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0343 - accuracy:
0.9906
Epoch 31/50
30/30 [==============================] - 0s 6ms/step - loss: 0.0099 - accuracy:
0.9969
Epoch 32/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0154 - accuracy:
0.9969
Epoch 33/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0054 - accuracy:
0.9979
Epoch 34/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0512 - accuracy:
0.9812
Epoch 35/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0520 - accuracy:
0.9833
Epoch 36/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0606 - accuracy:
0.9812
Epoch 37/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0114 - accuracy:
0.9937
Epoch 38/50
30/30 [==============================] - 0s 6ms/step - loss: 0.0084 - accuracy:
0.9958
Epoch 39/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0068 - accuracy:
0.9990
Epoch 40/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0197 - accuracy:
0.9885
Epoch 41/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0401 - accuracy:
0.9937
Epoch 42/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0251 - accuracy:
0.9937
Epoch 43/50
30/30 [==============================] - 0s 6ms/step - loss: 0.0418 - accuracy:
0.9896
```

```
Epoch 44/50
30/30 [==============================] - 0s 5ms/step - loss: 0.0078 - accuracy:
0.9969
Epoch 45/50
30/30 [==============================] - 0s 6ms/step - loss: 0.0166 - accuracy:
0.9937
Epoch 46/50
30/30 [==============================] - 0s 6ms/step - loss: 0.0050 - accuracy:
0.9979
Epoch 47/50
30/30 [==============================] - 0s 6ms/step - loss: 0.0048 - accuracy:
0.9979
Epoch 48/50
30/30 [==============================] - 0s 7ms/step - loss: 0.0022 - accuracy:
1.0000
Epoch 49/50
30/30 [==============================] - 0s 6ms/step - loss: 0.0164 - accuracy:
0.9937
Epoch 50/50
30/30 [==============================] - 0s 7ms/step - loss: 0.1133 - accuracy:
0.9708
```

[19]:
```python
print(training_history.history['loss'])
# prints the loss of the model on the train set
```

```
[0.5352107882499695, 0.08925800025463104, 0.05105925723910332,
0.04540086165070534, 0.07095669955015182, 0.044625215232372284,
0.039148248732089996, 0.031144781038165092, 0.029597699642181396,
0.03154650330543518, 0.020143283531069756, 0.040138084441423416,
0.020931633189320564, 0.03495061770081520, 0.04117148742079735,
0.05465039610862732, 0.03732357174158096, 0.018182838335633278,
0.013505736365914345, 0.028727306053042412, 0.009960971772670746,
0.011979969218373299, 0.009802144020795822, 0.008016565814614296,
0.014961236156523228, 0.035508155822753906, 0.022323209792375565,
0.008857224136590958, 0.05337311699986458, 0.03433821722865105,
0.009882161393761635, 0.015360146760940552, 0.005412007682025433,
0.051208604127168655, 0.051978934556245804, 0.06055993586778641,
0.011386849917471409, 0.008400123566389084, 0.006759406998753548,
0.019696107134222984, 0.04008736088871956, 0.025146545842289925,
0.041785482317209244, 0.007830962538719177, 0.016635408625006676,
0.0050080507062375546, 0.00478401081636548, 0.002201923867687583,
0.01644516922533512, 0.11325343698263168]
```

1.3.2 4. Evaluate Performance

[20]:
```
test_loss, test_acc = model.evaluate(image_test,  labels_test, verbose=2)

print('\nTest accuracy:', test_acc)
# metrics using accuracy score from Tensorflow on the test set
```

```
8/8 - 0s - loss: 0.0511 - accuracy: 0.9792 - 156ms/epoch - 19ms/step

Test accuracy: 0.9791666865348816
```

1.3.3 5. Make/Verify predictions

[21]:
```
probability_model = tf.keras.Sequential([model, tf.keras.layers.Softmax()])
# soft max is the last layer of the neural network, the relu function lies in␣
↪the hidden layers, softmax makes the descions to classify the images
```

[22]:
```
predictions = probability_model.predict(image_test)
# the model is instructed to predict the label of each image in the test set
```

```
8/8 [==============================] - 0s 2ms/step
```

[23]:
```
predictions[0]
```

[23]:
```
array([1.0000000e+00, 1.3423411e-10, 5.9319916e-10], dtype=float32)
```

[24]:
```
np.argmax(predictions[0])
# index 0 of image prediction
```

[24]:
```
0
```

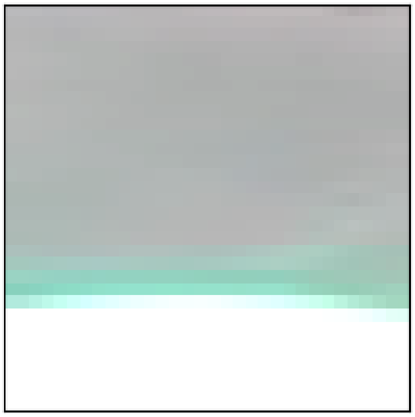
[25]:
```
labels_test[0]
# index 0 of image test, it sees it as a 0 or green so they match
```

[25]:
```
0
```

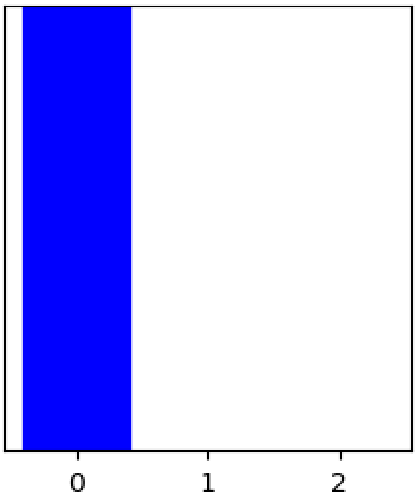
[26]:
```
def plot_image(i, predictions_array, true_label, img):
  true_label, img = true_label[i], img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img, cmap=plt.cm.binary)

  predicted_label = np.argmax(predictions_array)
  if predicted_label == true_label:
    color = 'blue'
  else:
    color = 'red'

  plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
  true_label = true_label[i]
  plt.grid(False)
  plt.xticks(range(3))
  plt.yticks([])
  thisplot = plt.bar(range(3), predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)

  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')

# prediction plotting function learned in class, essentially it takes in the␣
 ↪image and its true label,
# compares it to the predicted label and assigns a blue color to the text of a␣
 ↪correct prediction and red to a false
# the array follows the same idea but instead of showing the image, plots a bar␣
 ↪graph in the column of the predicted class
```

[27]:
```
i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], labels_test, image_test)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  labels_test)
plt.show()
# correct prediction of index 0
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

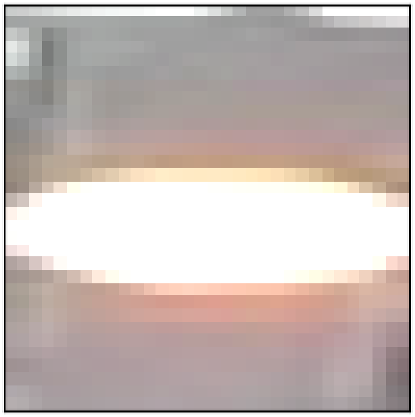


green 100% (green)

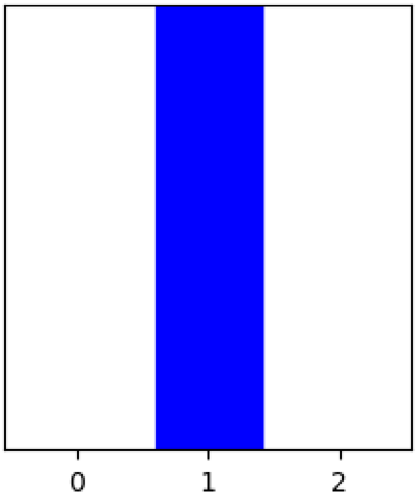


[28]:
```
i = 23
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], labels_test, image_test)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  labels_test)
plt.show()
# correct prediction of index 23
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

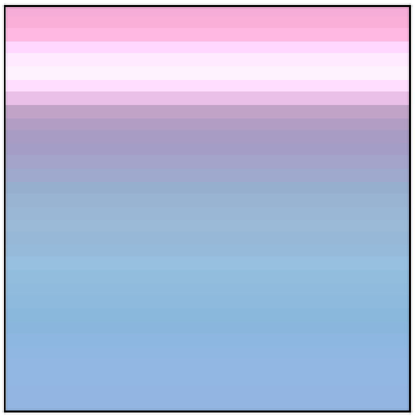


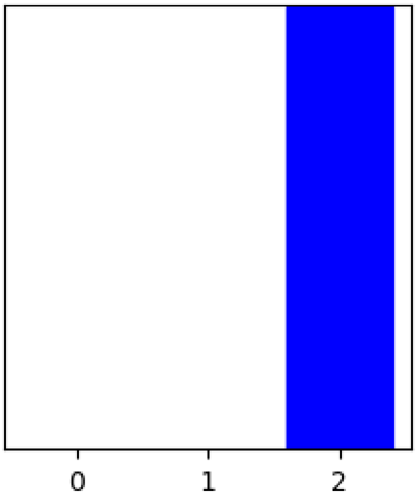
yellow 100% (yellow)



```
[29]: i = 5
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], labels_test, image_test)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  labels_test)
plt.show()
# correct prediction of index 5
```

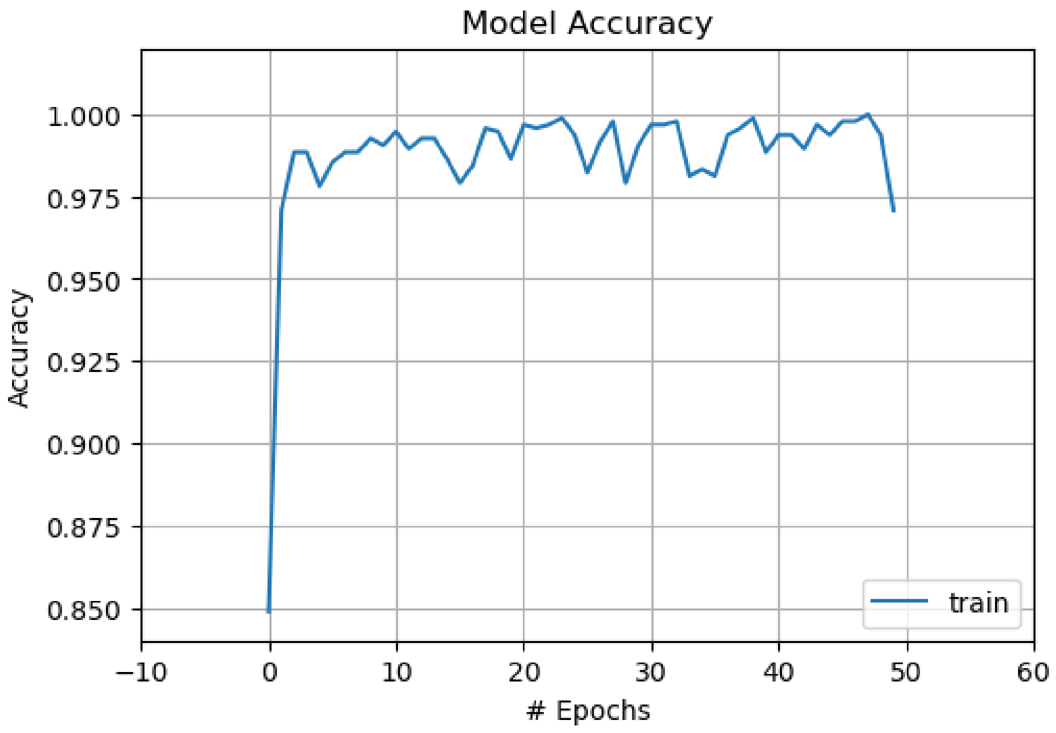
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

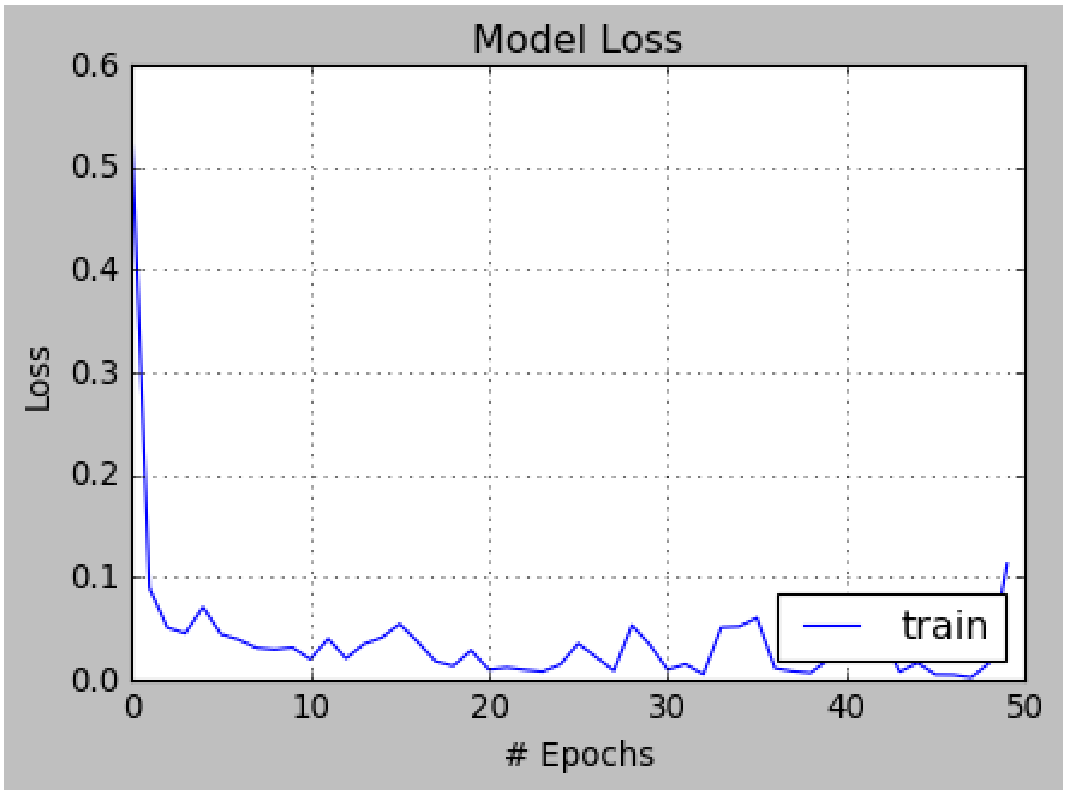




1.3.4 6. Model Accuracy and Loss Graphs

```
[30]: epoch = len(training_history.history.get('loss',[]))

# Draw Model Accuracy
plt.figure(2,figsize=(6,4))
plt.plot(range(epoch),training_history.history.get('accuracy'))
#plt.plot(range(epoch),training_history.history.get('val_acc'))
plt.xlabel('# Epochs')
plt.ylabel('Accuracy')
plt.title('Model Accuracy')
plt.grid(True)
plt.legend(['train','validation'],loc=4)
plt.style.use(['classic'])

# Draw Model Loss
plt.figure(1,figsize=(6,4))
plt.plot(range(epoch),training_history.history.get('loss'))
#plt.plot(range(epoch),training_history.history.get('val_loss'))
plt.xlabel('# Epochs')
plt.ylabel('Loss')
plt.title('Model Loss')
plt.grid(True)
plt.legend(['train','validation'], loc=4)
plt.style.use(['classic'])
```





1.3.5 7. Summarize results and enumerate conclusions

After 8 Epochs the model reached 99% accuracy and a loss of below 0.1

After 49 Epochs the model reached 100% accuracay

As expected the Multi-layer neural network performed very well when working with images

MLP works well for accurately classifying low resolution images, but for hd images I would opt for CNN because of its spatial mapping

1.4 LR

1.4.1 3. Build/Train the Model

[31]:
```
labels_train.shape
# check the shape of labels to see if tensorflow affected them
```

[31]:
```
(959,)
```

[32]:
```
image_train.shape
# check the shape of images to see if tensorflow affected them
```

[32]:
```
(959, 32, 32, 3)
```

[33]:
```
image_train = image_train.reshape(959,1024,-1)
length, width, color = image_train.shape
image_train = image_train.reshape((length,width*color))
image_train = image_train.transpose(0,1)

image_test = image_test.reshape(240,1024,-1)
length, width, color = image_test.shape
image_test = image_test.reshape((length,width*color))
image_test = image_test.transpose(0,1)

# unlike tf, logistic regression expects two dimensions instead of the 3 in tf
# to remedy this I reshaped the labels into the amount of images, the area of␣
 ↪the photo, and -1 to tell the array to fill in the rest
# then I reshape them as the length and width with color
# lastly transpose them to ensure that the amount of images is first in the␣
 ↪array
```

[34]:
```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(C=100, random_state=1,␣
 ↪class_weight='balanced',solver='lbfgs', max_iter=3000)

# C = 0.1 for regulaization strength, more regularization means less overfitting
# random_state = 1 as per usual
# solver is Limited-memory BFGS that works with multi class data like green,␣
 ↪yellow, red
# max_iter is 3,000 to make sure it can go past the defualt bottleneck of 100␣
 ↪iterations
# ovr fits a binary problem to each label
```

[35]:
```
lr.fit(image_train, labels_train)
X_combined_std = np.vstack((image_train, image_test))
y_combined = np.hstack((labels_train, labels_test))

# trains the data using the train data as arguments to help balance the weights
# then I combine the train/test data back together in stacked numpy arrays
```

1.4.2 4. Evaluate Performance

[36]:
```
from sklearn import metrics

y_pred = lr.predict(image_test)
print(metrics.accuracy_score(labels_test, y_pred))

# metrics for accuracy score of the model
```

```
1.0
```

```
[37]: from sklearn.metrics import accuracy_score

      print('Accuracy: %.3f' % accuracy_score(y_true=labels_test, y_pred=y_pred))

      # scikit acuracy score to evaluate the model
```

```
Accuracy: 1.000
```

1.4.3 5. Make/Verify predictions

```
[38]: predictions = y_pred
      # same as before with tf but with lr
```

```
[39]: predictions[20]
```

```
[39]: 2
```

```
[40]: np.argmax(predictions[0])
```

```
[40]: 0
```

```
[41]: labels_test[20]
```

```
[41]: 2
```

```
[42]: image_test.shape
```

```
[42]: (240, 3072)
```

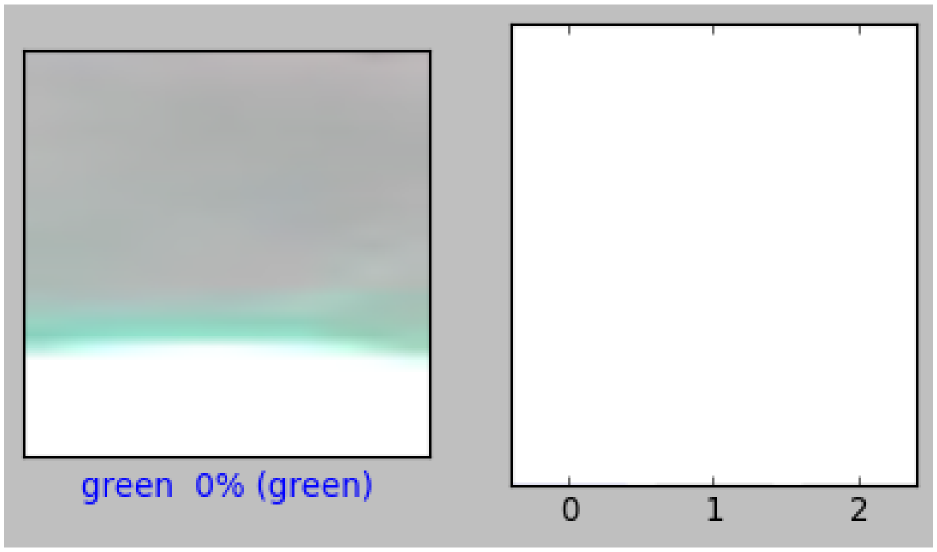
```
[43]: labels_test.shape
```

```
[43]: (240,)
```

```
[44]: image_test = image_test.reshape(240,32,32,3)
```

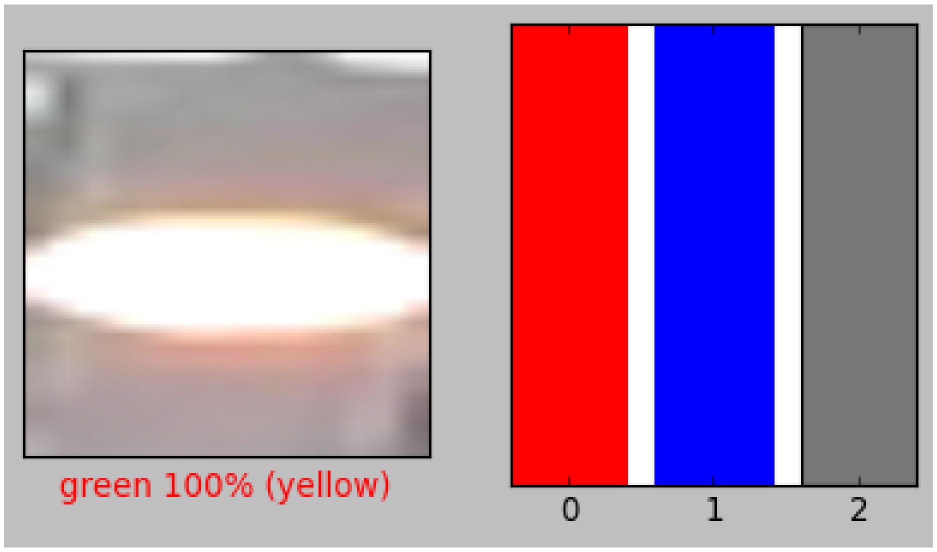
```
[45]: i = 0
      plt.figure(figsize=(6,3))
      plt.subplot(1,2,1)
      plot_image(i, predictions[i], labels_test, image_test)
      plt.subplot(1,2,2)
      plot_value_array(i, predictions[i],  labels_test)
      plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



```
[46]: i = 23
      plt.figure(figsize=(6,3))
      plt.subplot(1,2,1)
      plot_image(i, predictions[i], labels_test, image_test)
      plt.subplot(1,2,2)
      plot_value_array(i, predictions[i],  labels_test)
      plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



```
[47]: predictions[23]
```

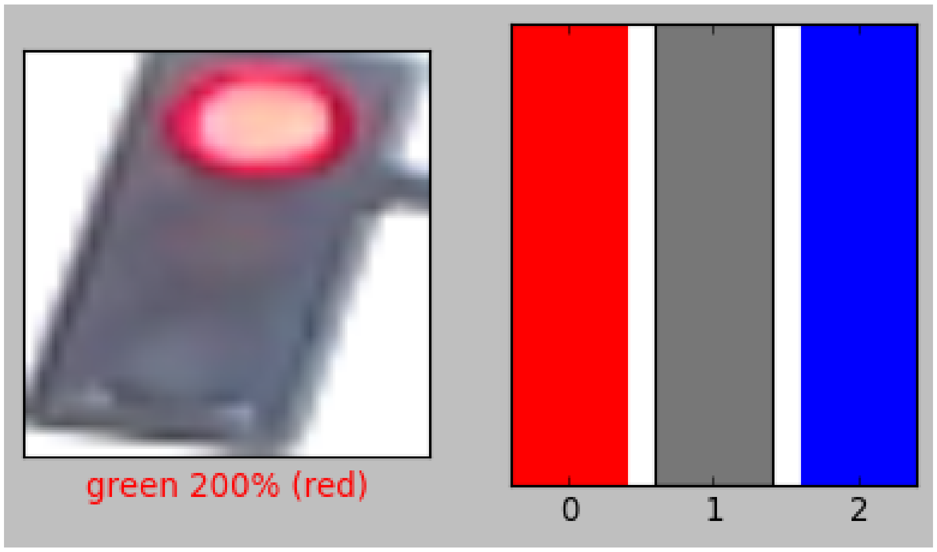
```
[47]: 1
```

```
[48]: labels_test[23]
```

```
[48]: 1
```

```
[49]: i = 80
      plt.figure(figsize=(6,3))
      plt.subplot(1,2,1)
      plot_image(i, predictions[i], labels_test, image_test)
      plt.subplot(1,2,2)
      plot_value_array(i, predictions[i],  labels_test)
      plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



```
[50]: predictions[80]
```

```
[50]: 2
```

```
[51]: labels_test[80]
```

```
[51]: 2
```

```
[52]: image_test = image_test.reshape(240,1024,-1)
      length, width, color = image_test.shape
```

```
image_test = image_test.reshape((length,width*color))
image_test = image_test.transpose(0,1)
```

```
[53]: from sklearn.metrics import confusion_matrix

confmat = confusion_matrix(y_true=labels_test, y_pred=y_pred)

print(confmat)
print('Test Accuracy: %.3f' % lr.score(image_test, labels_test))

# from scikit this is the confusion matrix for the predicted and test data
# there are no missclassifications of green, yellow, or red lights
# The test accuracy is 100%
```

```
[[ 87   0   0]
 [  0   8   0]
 [  0   0 145]]
Test Accuracy: 1.000
```

As you can see above the plot image function was failing for some reason, but as I have shown with the predictions vs labels of the same indexes, the lr model isn't misidentifying any} of the images

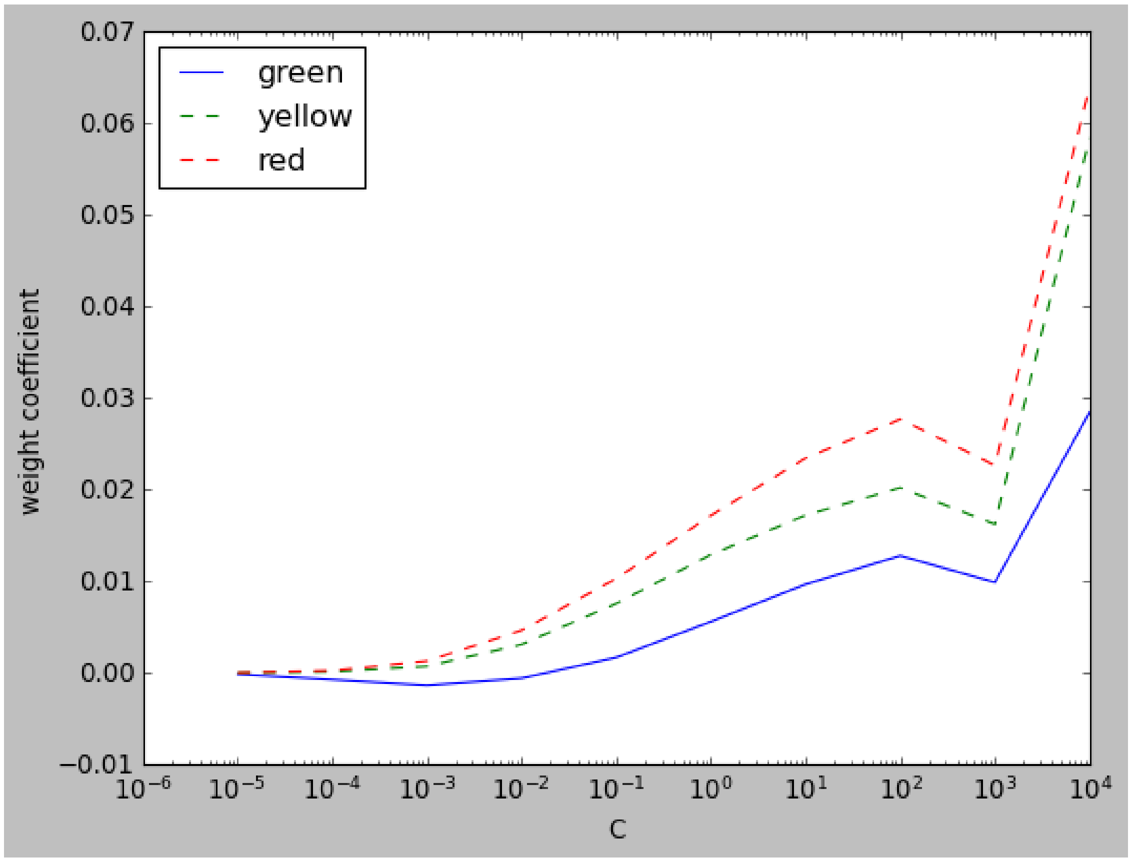
I tested svm, knn, and lr as second algorithms, knn and svm got scores below 0.96 in accuracy with misidentification of reds and greens

This is why I chose lr

1.4.4 6. Model Accuracy and Loss Graphs

```
[54]: weights, params = [], []
for c in np.arange(-5, 5):
    lr = LogisticRegression(C=10.**c, random_state=1,␣
 ↪class_weight='balanced',solver='lbfgs',max_iter=3000)
    lr.fit(image_train, labels_train)
    weights.append(lr.coef_[1])
    params.append(10.**c)

weights = np.array(weights)
plt.plot(params, weights[:, 0],
         label='green')
plt.plot(params, weights[:, 1], linestyle='--',
         label='yellow')
plt.plot(params, weights[:, 2], linestyle='--',
         label='red')
plt.ylabel('weight coefficient')
plt.xlabel('C')
plt.legend(loc='upper left')
plt.xscale('log')
plt.show()
```

```
# loss graph from class looking at overfitting by changing the c or␣
 ↪regularization value of the model
```



1.4.5 7. Summarize results and enumerate conclusions

Logistic Regression worked well for images, far better than the other models I tested. The model was 100% accurate on the test set.

2 Conclusions

2.1 At the end of the day, Neural Networks will always outperform more binary algorithms like LR because of its layers and nodes, but I was happy to see LR(my favorite) perform well with the image data.