Self-Driving Car Engineer Nanodegree

Deep Learning

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 \n", "**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.

Step 0: Load The Data

In [38]:

```
# Load pickled data
import pickle
import pandas as pd
%matplotlib inline
from multiprocessing import Queue
import pickle
import random
import numpy as np
import tensorflow as tf
import time
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import os
```

```
from sklearn.utils import shuffle


# path
training_file   = "/home/chr/Udacity/P2 traffic sign classifier/traffic-signs-
data/train.p"
validation_file = "/home/chr/Udacity/P2 traffic sign classifier/traffic-signs-
data/valid.p"
testing_file    = "/home/chr/Udacity/P2 traffic sign classifier/traffic-signs-
data/test.p"

# open and load files
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)

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

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.

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

In [6]:

```
# Number of training examples
n_train = len(X_train)

# Number of testing examples.
n_test = len(X_test)

# Number of validation examples.
```

```
n_valid = len(X_valid)

# What's the shape of an (the fist) traffic sign image?
image_shape = X_train[0].shape

# TODO: How many unique classes/labels there are in the dataset.
n_classes = len(set(y_train))       #set():unordered collection with no
duplicate elements

print("Number of training examples =", n_train)
print("Number of validation examples =", n_valid)
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 validation examples = 4410
Number of testing examples = 12630
Image data shape = (32, 32, 3)
Number of classes = 43
```

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.

In [3]:

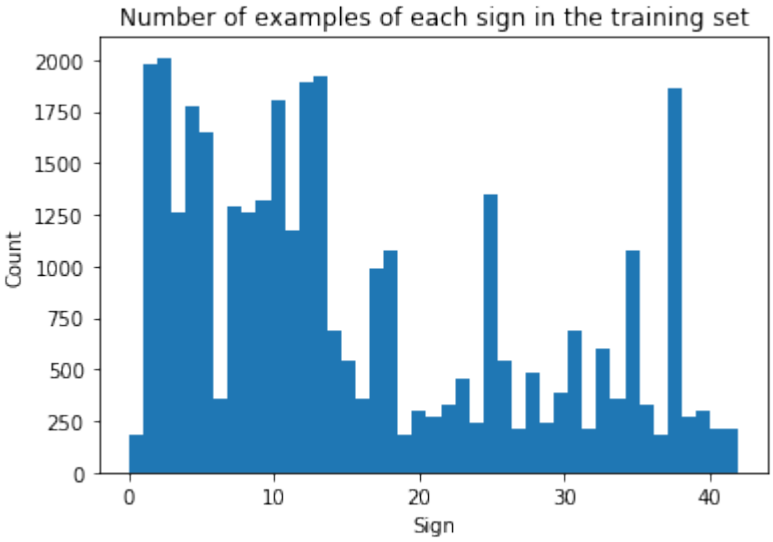
```
# plot 4 random images
print('9 random sample images')
for i in range(9):
    index = random.randint(0, n_train)
    plt.subplot(3,3,i+1)
    plt.imshow(X_train[index])

9 random sample images
```

In [10]:

```
# histogram of labels and count
plt.hist(y_train, bins=n_classes)
plt.title('Number of examples of each sign in the training set')
plt.xlabel('Sign')
plt.ylabel('Count')
plt.plot()
```

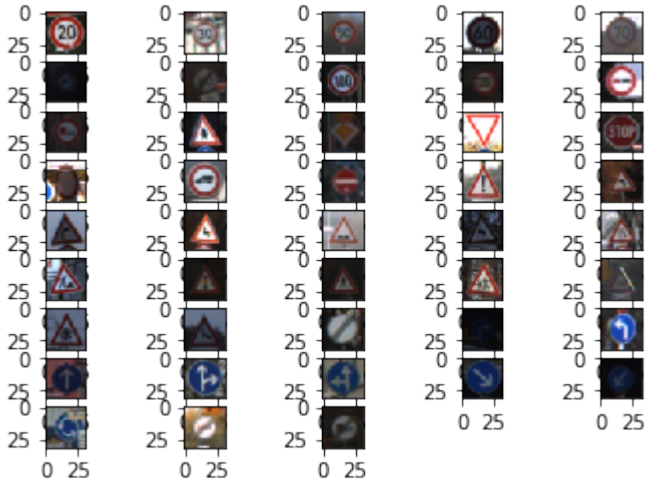
Out[10]:

```
[]
```



In [40]:

```
# plot every sign in labeled order
n=0
for n in range(n_classes):
    i=0
    for i in range(n_train):
        if y_train[i] == n:
            #print(y_train[i])
            plt.subplot(9,5,y_train[i]+1)
            plt.imshow(X_train[i])
            break
```



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.

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

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

In [39]:

```
#shuffle signs
from sklearn.utils import shuffle
X_train, y_train = shuffle(X_train, y_train)
```

In [40]:

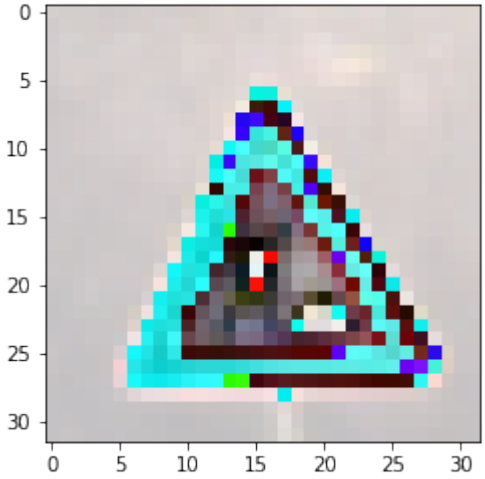
```
#normalize train-, validation- and testimages
def normalize_images(images):
    images = (images - images.mean()) / (np.max(images) - np.min(images))
    return images

X_train= normalize_images(X_train)
X_valid= normalize_images(X_valid)
X_test= normalize_images(X_test)
```

In [91]:

```
#show normalized image
plt.imshow(X_train[500], cmap='gray')
```

Out[91]:

```
<matplotlib.image.AxesImage at 0x7fefdcdb5940>
```



Model Architecture

In [41]:

```
"""derived from LeNet"""
EPOCHS = 15
BATCH_SIZE = 100

from tensorflow.contrib.layers import flatten

def SignNet(x):
    # Arguments used for tf.truncated_normal, randomly defines variables for the
weights and biases for each layer
    mu = 0
    sigma = 0.1
    dropout = 0.75

    # Layer 1: Convolutional. Input = 32x32x3. Output = 28x28x18.
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 3, 18), mean = mu,
stddev = sigma))
    conv1_b = tf.Variable(tf.zeros(18))
    conv1   = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') +
conv1_b

    # Activation.
    conv1 = tf.nn.relu(conv1)
    conv1 = tf.nn.dropout(conv1, dropout)

    # Pooling. Input = 28x28x18. Output = 14x14x18.
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
padding='VALID')

    # Layer 2: Convolutional. Output = 10x10x48.
    conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 18, 48), mean = mu,
stddev = sigma))
    conv2_b = tf.Variable(tf.zeros(48))
    conv2   = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1],
padding='VALID') + conv2_b

    # Activation.
    conv2 = tf.nn.relu(conv2)
    conv2 = tf.nn.dropout(conv2, dropout)

    # Pooling. Input = 10x10x48. Output = 5x5x48.
```

```
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
padding='VALID')

    # Flatten. Input = 5x5x48. Output = 1200.
    fc0   = flatten(conv2)

    # Layer 3: Fully Connected. Input = 1200. Output = 300. #earlier 120 not
1200
    fc1_W = tf.Variable(tf.truncated_normal(shape=(1200, 300), mean = mu, stddev
= sigma))
    fc1_b = tf.Variable(tf.zeros(300))
    fc1   = tf.matmul(fc0, fc1_W) + fc1_b

    # Activation.
    fc1    = tf.nn.relu(fc1)
    fc1 = tf.nn.dropout(fc1, dropout)

    # Layer 4: Fully Connected. Input = 300. Output = 100.
    fc2_W  = tf.Variable(tf.truncated_normal(shape=(300, 100), mean = mu, stddev
= sigma))
    fc2_b  = tf.Variable(tf.zeros(100))
    fc2    = tf.matmul(fc1, fc2_W) + fc2_b

    # Activation.
    fc2    = tf.nn.relu(fc2)
    fc2 = tf.nn.dropout(fc2, dropout)

    # Layer 5: Fully Connected. Input = 100. Output = 43.
    fc3_W  = tf.Variable(tf.truncated_normal(shape=(100, 43), mean = mu, stddev
= sigma))
    fc3_b  = tf.Variable(tf.zeros(43))
    logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits


#Features and Labels
x = tf.placeholder(tf.float32, (None, 32, 32, 3))
y = tf.placeholder(tf.int32, (None))
one_hot_y = tf.one_hot(y, 43)
```

In [42]:

```
rate = 0.001

logits = SignNetcross_entropy(x)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, one_hot_y)
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation)




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()
correct_prediction
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


with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_examples = len(X_train)

    print("Training...")
    print()
    for i in range(EPOCHS):
        X_train, y_train = shuffle(X_train, y_train)
        for offset in range(0, num_examples, BATCH_SIZE):
            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})

        validation_accuracy = evaluate(X_valid, y_valid)
        print("EPOCH {} ...".format(i+1))
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print()

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

```
Training...

EPOCH 1 ...
Validation Accuracy = 0.792

EPOCH 2 ...
Validation Accuracy = 0.859

EPOCH 3 ...
Validation Accuracy = 0.893

EPOCH 4 ...
Validation Accuracy = 0.894

EPOCH 5 ...
Validation Accuracy = 0.916

EPOCH 6 ...
Validation Accuracy = 0.913

EPOCH 7 ...
Validation Accuracy = 0.908

EPOCH 8 ...
Validation Accuracy = 0.920

EPOCH 9 ...
Validation Accuracy = 0.931

EPOCH 10 ...
Validation Accuracy = 0.939

EPOCH 11 ...
Validation Accuracy = 0.937
```

```
EPOCH 12 ...
Validation Accuracy = 0.921

EPOCH 13 ...
Validation Accuracy = 0.928

EPOCH 14 ...
Validation Accuracy = 0.941

EPOCH 15 ...
Validation Accuracy = 0.936

Model saved
```

In [43]:

```
#Test
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))

    test_accuracy = evaluate(X_test, y_test)
    print("Test Accuracy = {:.3f}".format(test_accuracy))
```

```
Test Accuracy = 0.927
```

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.

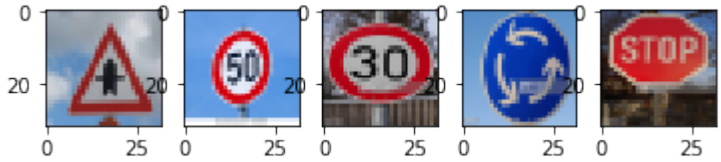
Load and Output the Images

Predict the Sign Type for Each Image

In [125]:

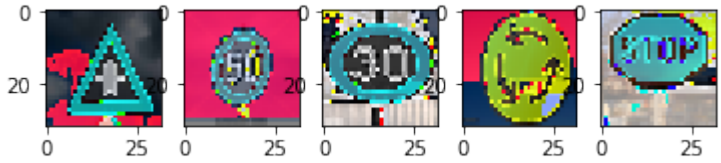
```
#get new images
import cv2
import glob
import numpy as np
import matplotlib.pyplot as plt
files = glob.glob ("/home/chr/Udacity/P2 traffic sign classifier/signs from the
internet/*.jpg")
X_internet = np.zeros((len(files), 32,32,3), dtype=np.uint8)
n_new= len(files)

from io import BytesIO

#read and resize
for i in range(n_new) :
    image = plt.imread(files[i])
    image = cv2.resize(image, (32,32), interpolation=cv2.INTER_AREA)
    X_internet[i] = image
```

```
    plt.subplot(1,5,1+i)
    plt.imshow(X_internet[i])
```



In [126]:

```
#creating an array of the correct labels
y_internet = [11,2,1,40,14]

# normalize images
X_internet= normalize_images(X_internet)
for i in range(n_new):
    plt.subplot(1,5,1+i)
    plt.imshow(X_internet[i])
```



In [171]:

```
#Test images from internet

with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))

    test_accuracy = evaluate(X_internet, y_internet)
    print("Test Accuracy = {:.3f}".format(test_accuracy))
    #--------------------

    for i in range(len(files)):
        top_five = sess.run(tf.nn.top_k(tf.nn.softmax(logits), k=5),
feed_dict={x: [X_internet[i]]})
        print(top_five)


Test Accuracy = 0.800
TopKV2(values=array([[  1.00000000e+00,   3.84700882e-11,   3.23922604e-16,
          4.86102603e-22,   6.46777393e-24]], dtype=float32),
indices=array([[11, 21, 27, 40, 28]], dtype=int32))
TopKV2(values=array([[  4.87712145e-01,   4.42609131e-01,   6.96767271e-02,
          1.54513896e-06,   2.97663348e-07]], dtype=float32), indices=array([[2,
1, 5, 7, 3]], dtype=int32))
TopKV2(values=array([[  5.74092329e-01,   4.04238105e-01,   2.15539262e-02,
          1.15531249e-04,   1.03745258e-07]], dtype=float32),
indices=array([[ 5,  1,  2,  3, 14]], dtype=int32))
TopKV2(values=array([[  1.00000000e+00,   1.19916105e-10,   1.75992883e-11,
          4.11457896e-13,   3.29353912e-16]], dtype=float32),
indices=array([[40, 38, 41, 42, 36]], dtype=int32))
TopKV2(values=array([[  1.00000000e+00,   1.90360653e-13,   5.80655485e-15,
          5.78263464e-16,   4.66309796e-16]], dtype=float32),
indices=array([[12, 11,  6, 10, 42]], dtype=int32))
```

1. Right of way at next intersection: correct classification with a certainty of 100%

2. Speedimit 50km/h: correct classification with a certainty of 49%, next guesses are speedlimit 30km/h with 44% and speedlimit 80km/h with 7%
3. Speedimit 30km/h: correct classification with a certainty of 57%, next guesses are speedlimit 30km/h with 40% and speedlimit 50km/h with 2%
4. Roundabout mandatory: correct classification with a certainty of 100%
5. Stop: INCORRECT classification with a certainty of 100%. The stop sign is classified as a priority road.

Interpretation: For the model Right of way at next intersection and Roundabout mandatory are very easy to distinguish. The the model is less certain about the speedlimits, nevertheless it classifies correctly. The incorrect prediction with a wrong certainty of 100% is very strange. At first I tought my labeling could be wrong. However I can't finde the mistake. Maybe its due to the uncut image in contrast to the cut training images.

Step 4: 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.

In [ ]:

```
### 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 represents the calculated state of a specific weight layer
# activation_min/max: can be used to view the activation contrast in more
detail, by default matplot sets min and max to the actual min and max values of
the output
# plt_num: used to plot out multiple different weight feature map sets on the
same block, just extend the plt number for each new feature map entry

def outputFeatureMap(image_input, tf_activation, activation_min=-1,
activation_max=-1 ,plt_num=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 variable
    # If you get an error tf_activation is not defined it maybe having trouble
accessing the variable from inside a function
    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 each row and column
        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 =activation_min, vmax=activation_max, cmap="gray")
        elif activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest",
vmax=activation_max, cmap="gray")
        elif activation_min !=-1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest",
vmin=activation_min, cmap="gray")
        else:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest",
cmap="gray")
```

Question 9

Discuss how you used the visual output of your trained network's feature maps to show that it had learned to look for interesting characteristics in traffic sign images

Answer:

> **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 \n", "**File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

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.