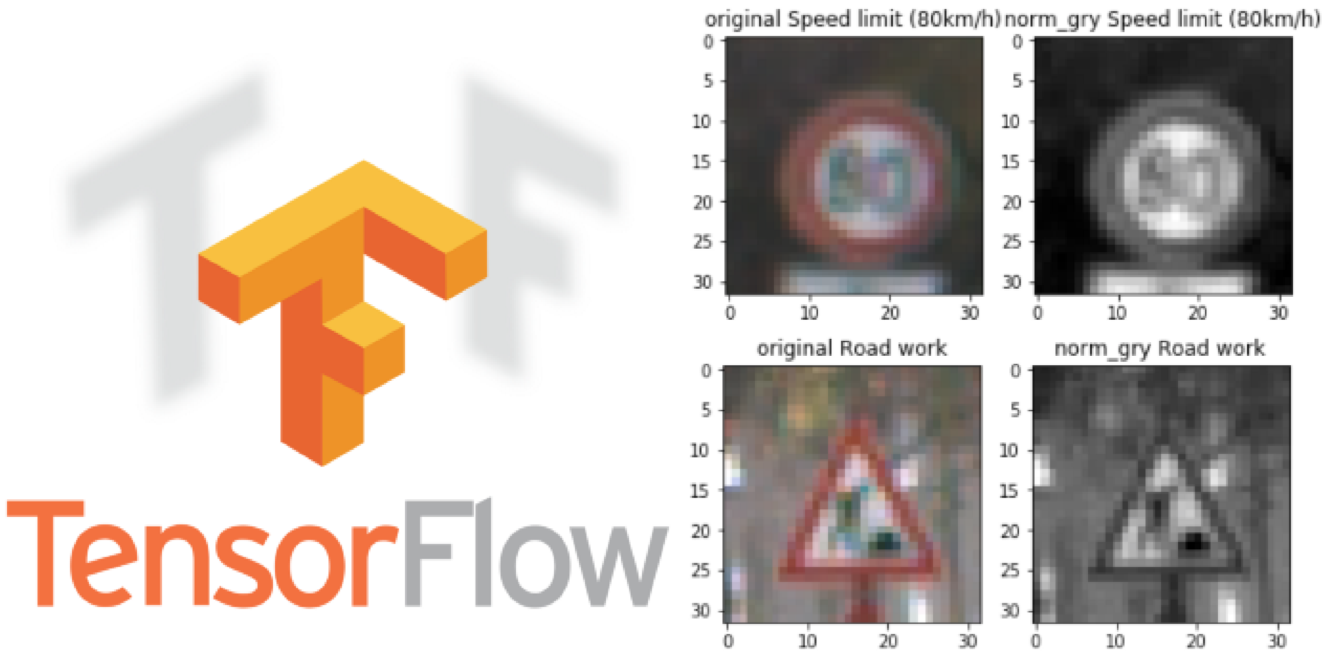
Traffic Sign Recognition with TensorFlow 2.x

End to end example from raw image processing to model evaluation

Jun M. Jan 21 · 4 min read ★



Left: TensorFlow logo. Image credit: tensorflow.com Right: original and grayscale traffic sign from data below

Introduction

TensorFlow is a software library most famous for its flexibility and ease of use in neural networks. You can find a lot of examples online from image classification to object detection, but many of them are based on TensorFlow 1.x. It is a big change from TensorFlow 1.0 to 2.0 with a tighter Keras integration, where the focus is more on higher level APIs. Many methods have been depreciated (or you may use `tf.compat.v1`). Model construction becomes a lot easier and default parameters in each model already work very well for general use. With all the benefits, it still provides flexibility should you need to change the parameters.

It this article, I will use TensorFlow 2.0 (more specifically, Keras in TensorFlow) to classify traffic signs. The dataset is available many places on the web, but I will use this one hosted on Kaggle.

Data overview

The data package includes folders of `Train`, `Test` and a `test.csv`. There are a `meta.csv` and a `Meta` folder to show the standard image for each traffic sign. There is also a `signname.csv` for mapping a label to its description. `Train` folder contains 43 sub-folders whose names are the labels of the images in them. For example, all the images in folder `0` has a class label of `0` and so on… The images are of different sizes ranging from 20x20 to 70x70, and all have 3 channels: RGB.

So the first thing I have to do is to resize all the images to 32x32x3 and read them into a numpy array as training features. At the same time, I created another numpy array with labels of each image, which is from the fold name where the image loaded from.

In [2]:
```
import cv2
import glob
import pickle
import numpy as np
import pandas as pd

# function to read and resize images, get labels and store
 them into np array
def get_image_label_resize(label, filelist, dim = (32, 32),
dataset = 'Train'):
    x = np.array([cv2.resize(cv2.imread(fname), dim, interp
olation = cv2.INTER_AREA) for fname in filelist])
    y = np.array([label] * len(filelist))

    #print('{} examples loaded for label {}'.format(x.shape
[0], label))
    return (x, y)

# data for label 0. I store them in parent level so that th
ey won't be uploaded to github
filelist = glob.glob('../Train/'+'0'+'/*.png')
trainx, trainy = get_image_label_resize(0, glob.glob('../Tr
ain/'+str(0)+'/*.png'))
```

In [3]:
```
# go through all others labels and store images into np arr
```

```
ay
```

load training and testing data.ipynb hosted with ❤ by **GitHub** **view raw**

I need to do the same for testing images. However the labels for testing images are stored as `ClassId` in `test.csv` with paths of that image. So I use pandas to read the `csv` file, load the image from path and assign the corresponding `ClassId`.

From the training set, I randomly spitted 20% as validation set for use during the process of model training. The model accuracy of training and validation will give us information about underfitting or overfitting.

```
In [11]: # shuffle training data and split them into training and va
         lidation
         indices = np.random.permutation(trainx.shape[0])
         # 20% to val
         split_idx = int(trainx.shape[0]*0.8)
         train_idx, val_idx = indices[:split_idx], indices[split_idx
         :]
         X_train, X_validation = trainx[train_idx,:], trainx[val_idx
         ,:]
         y_train, y_validation = trainy[train_idx], trainy[val_idx]

In [16]: # get overall stat of the whole dataset
         n_train = X_train.shape[0]
         n_validation = X_validation.shape[0]
         n_test = X_test.shape[0]
         image_shape = X_train[0].shape
         n_classes = len(np.unique(y_train))
         print("There are {} training examples ".format(n_train))
         print("There are {} validation examples".format(n_validatio
         n))
         print("There are {} testing examples".format(n_test))
         print("Image data shape is {}".format(image_shape))
         print("There are {} classes".format(n_classes))

         There are 31367 training examples
         There are 7842 validation examples
         There are 12630 testing examples
```

create val.ipynb hosted with ❤ by **GitHub** **view raw**

Next, I converted images to grayscale and normalized each pixels. Normalization makes model to converge more quickly.

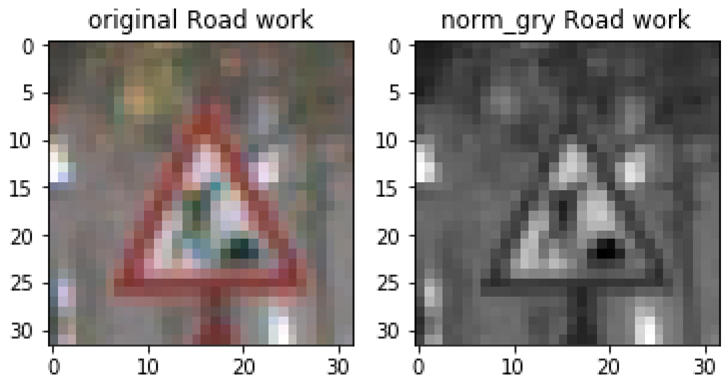
```
In [15]: # convert the images to grayscale
         X_train_gry = np.sum(X_train/3, axis=3, keepdims=True)
         X_validation_gry = np.sum(X_validation/3, axis=3, keepdims=
         True)
         X_test_gry = np.sum(X_test/3, axis=3, keepdims=True)
```

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

# Normalize data
X_train_normalized_gry = (X_train_gry-128)/128
X_validation_normalized_gry = (X_validation_gry-128)/128
X_test_normalized_gry = (X_test_gry-128)/128
```

```
In [23]: import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('seaborn-colorblind')

# descriptions for each label
sign = pd.read_csv('signnames.csv')

# pick an image, display the original and the normalized gr
ay image
index = np.random.randint(0, n_train)
fig, ax = plt.subplots(1,2)
ax[0].set_title('original ' + sign.loc[sign['ClassId'] ==y_
train[index], 'SignName'].values[0])
ax[0].imshow(cv2.cvtColor(X_train[index], cv2.COLOR_BGR2RGB
))
```

pre_processing.ipynb hosted with ❤ by **GitHub** **view raw**

Here is a comparison between an RGB and grayscale image. The grayscale image still retains its features and can be recognized but with much smaller size.



A comparison of original and grayscale image

Model construction

I will use the famous LeNet published in 1998 by Yann LeCun et al. The input shape is 32x32x1. First convolution layer will have a depth of 6, a filter size of (5, 5), and a stride of (1, 1). Valid padding is used (i.e. no padding). Therefore the width (or height) of this layer is 32–5+1 = 28, i.e. the shape is 28x28x6. The activation of this layer is relu.

```
In [78]: import tensorflow as tf
from tensorflow.keras import datasets, layers, models
```

```
model = models.Sequential()
# Conv 32x32x1 => 28x28x6.
model.add(layers.Conv2D(filters = 6, kernel_size = (5, 5), strides=(1, 1), padding='valid',
                        activation='relu', data_format = 'channels_last', input_shape = (32, 32, 1)))
# Maxpool 28x28x6 => 14x14x6
model.add(layers.MaxPooling2D((2, 2)))
# Conv 14x14x6 => 10x10x16
model.add(layers.Conv2D(16, (5, 5), activation='relu'))
# Maxpool 10x10x16 => 5x5x16
model.add(layers.MaxPooling2D((2, 2)))
# Flatten 5x5x16 => 400
model.add(layers.Flatten())
# Fully connected 400 => 120
model.add(layers.Dense(120, activation='relu'))
# Fully connected 120 => 84
model.add(layers.Dense(84, activation='relu'))
# Dropout
model.add(layers.Dropout(0.2))
# Fully connected, output layer 84 => 43
model.add(layers.Dense(43, activation='softmax'))
```

lenet.ipynb hosted with ❤ by **GitHub** **view raw**

Following the first convolution layer is a max polling layer. It effectively downsizes the data by only selecting the max value pixel for adjacent pixels. LeNet uses a (2, 2) kernel size. The default stride is the same as kernel, which means there is no overlap between the group of pixels the max is selected from. Now the shape of output becomes 14x14x6.

Next LeNet has a second convolution layer with depth of 16, filter size of (5, 5) and relu activation function, followed by a max pooling layer. The width (or height) of output is now (14–5+1)/2 = 5, i.e. the shape is 5x5x16.

The data is then flattened before the fully connected layers. The shape of output is 5x5x16 = 400. This followed by 2 fully connected layers of size 120 and 84, with relu as activation function for both. A dropout layer is added to reduce overfitting. And finally a fully connected layer with size of 43 (the no. of classes). Softmax is used to return the probabilities of each class.

```
In [79]: model.summary()

Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Para
```

```
m #
=================================================================
======
conv2d_12 (Conv2D)           (None, 28, 28, 6)         156
_________________________________________________________________
______
max_pooling2d_12 (MaxPooling (None, 14, 14, 6)         0
_________________________________________________________________
______
conv2d_13 (Conv2D)           (None, 10, 10, 16)        2416
_________________________________________________________________
______
max_pooling2d_13 (MaxPooling (None, 5, 5, 16)          0
_________________________________________________________________
______
flatten_6 (Flatten)          (None, 400)               0
_________________________________________________________________
______
dense_17 (Dense)             (None, 120)               4812
0
_________________________________________________________________
```

model summary.ipynb hosted with ❤ by **GitHub** view raw

Model training and evaluation

Training is very straightforward with Keras. We only need to specify optimizer, loss function and validation metric. Within 10 epochs, the accuracy of both training and validation is above 0.97. With a dropout layer, there is no apparent overfitting. On the other hand, increasing training will only yield minimum improvement, so I stopped only after 10 epochs.

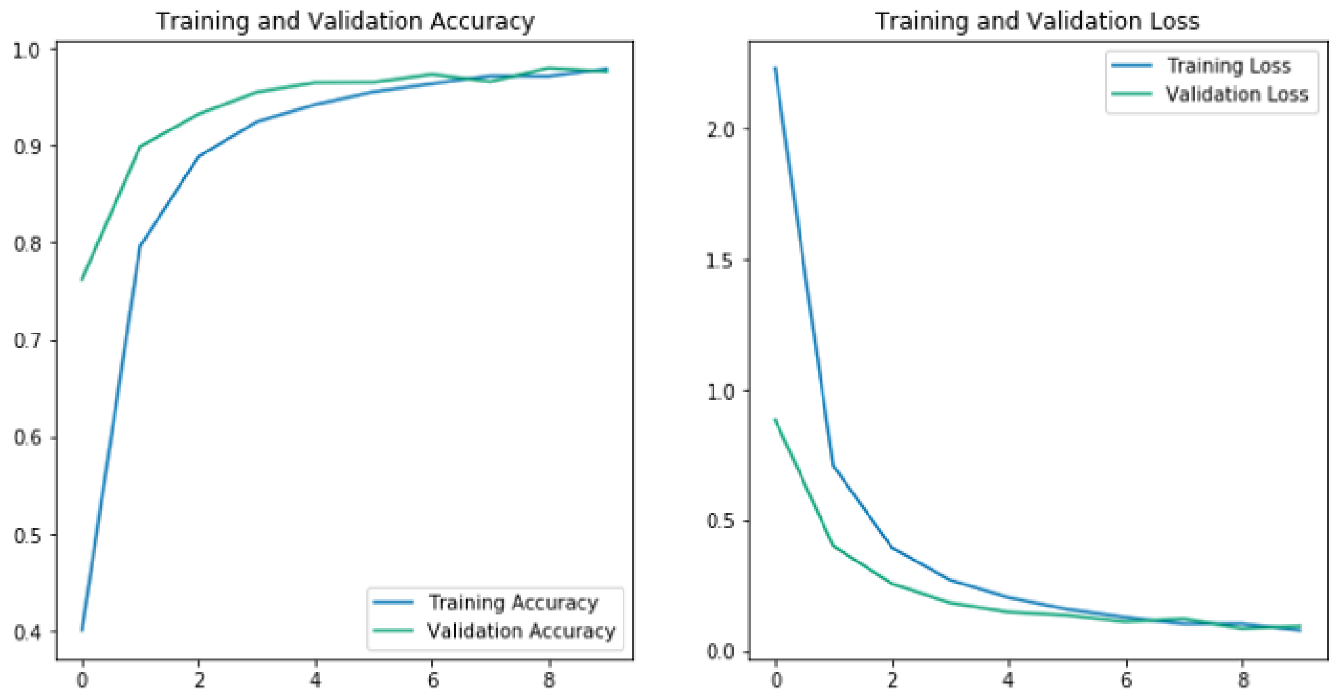
In [80]:
```
# specify optimizer, loss function and metric
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# training batch_size=128, epochs=10
conv = model.fit(X_train, y_train, batch_size=128, epochs=1
0,
                    validation_data=(X_validation, y_valida
tion))
```

```
Train on 31367 samples, validate on 7842 samples
Epoch 1/10
31367/31367 [==============================] - 8s 255us/sam
ple - loss: 2.2325 - accuracy: 0.4007 - val_loss: 0.8845 -
val_accuracy: 0.7619
Epoch 2/10
31367/31367 [==============================] - 8s 241us/sam
ple - loss: 0.7074 - accuracy: 0.7961 - val_loss: 0.4006 -
val_accuracy: 0.8986
Epoch 3/10
31367/31367 [==============================] - 8s 242us/sam
ple - loss: 0.3948 - accuracy: 0.8884 - val_loss: 0.2567 -
val_accuracy: 0.9320
```

```
Epoch 4/10
31367/31367 [==============================] - 8s 243us/sam
ple - loss: 0.2703 - accuracy: 0.9242 - val_loss: 0.1827 -
```

training.ipynb hosted with ❤ by **GitHub** **view raw**

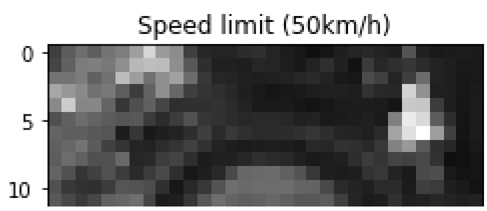
We can also plot the model performance on training and validation with each epoch. Indeed, the model appears to be quite generalized and not overfitting the training data.

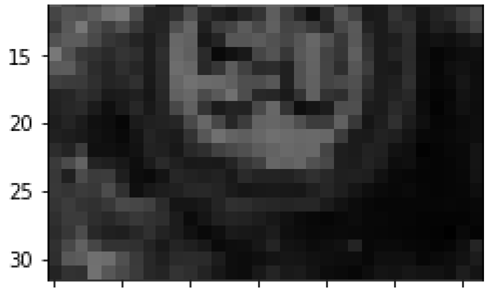


Accuracy and loss of training and validation for each epoch

Finally, the model is used to predict the labels of test set. The accuracy is about 0.925.

```
         12630/12630 [==============================] - 1s 82us/samp
         le - loss: 0.6084 - accuracy: 0.9250
Out[54]: [0.6084245350228081, 0.9250198]

In [97]: index = np.random.randint(0, n_test)
         im = X_test[index]
         fig, ax = plt.subplots()
         ax.set_title(sign.loc[sign['ClassId'] ==np.argmax(model.pre
         dict(np.array([im]))), 'SignName'].values[0])
         ax.imshow(im.squeeze(), cmap = 'gray')

Out[97]: <matplotlib.image.AxesImage at 0x149dfdbd0>
```





test_pred.ipynb hosted with ❤ by **GitHub** **view raw**

Conclusion

Previously I wrote an article on building a neural network from scratch, which requires hardcore linear algebra. By using libraries like TensorFlow, the task becomes much easier and the model is more powerful.

You can get the full code from here.

Convolution Neural Net Image Recognition Python TensorFlow Keras