< Deep Learning - PART2 TF2 CNNs > ¶

Ch 5. CNNs Workshop 2 - CIFAR10 : Image Classifier with Batch Norm

2021/10/01

[Reference] :

- TensorFlow Core - Tutorials: **Convolutional Neural Network (CNN)** https://www.tensorflow.org/tutorials/images/cnn?hl=zh_tw (https://www.tensorflow.org/tutorials/images/cnn?hl=zh_tw)
- **CIFAR-10 and CIFAR-100 datasets** https://www.cs.toronto.edu/~kriz/cifar.html (https://www.cs.toronto.edu/~kriz/cifar.html)
- 李飛飛教授：Convolutional Neural Networks (教學投影片) (http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture5.pdf (http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture5.pdf))
- 李飛飛教授：Convolutional Neural Networks (CNNs / ConvNets) (https://cs231n.github.io/convolutional-networks/ (https://cs231n.github.io/convolutional-networks/))
- `tf.keras.layers.Conv2D` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D))

This tutorial demonstrates training a simple Convolutional Neural Network (https://developers.google.com/machine-learning/glossary/#convolutional_neural_network) (CNN) to classify CIFAR images (https://www.cs.toronto.edu/~kriz/cifar.html). Because this tutorial uses the Keras Sequential API (https://www.tensorflow.org/guide/keras/overview), creating and training our model will take just a few lines of code.

Import TensorFlow

In [1]:

```
import tensorflow as tf

from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

Download and prepare the CIFAR10 dataset

The CIFAR10 dataset contains 60,000 color images in 10 classes, with 6,000 images in each class. The dataset is divided into 50,000 training images and 10,000 testing images. The classes are mutually exclusive and there is no overlap between them.

In [2]:

```
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

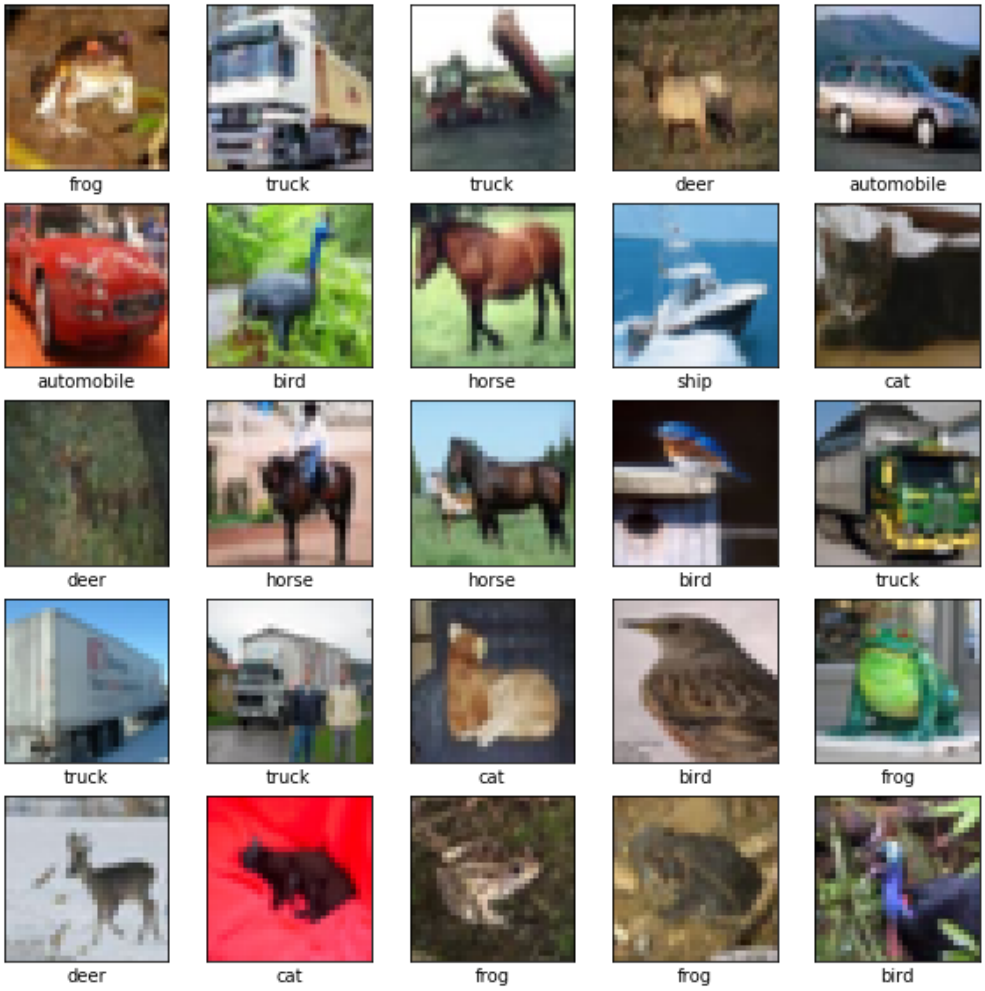
Verify the data

To verify that the dataset looks correct, let's plot the first 25 images from the training set and display the class name below each image.

In [3]:

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    # The CIFAR labels happen to be arrays,
    # which is why you need the extra index
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```



Create the convolutional base

- The 6 lines of code below define the convolutional base using a common pattern: a stack of Conv2D (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D) and MaxPooling2D (https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D) layers.

> - As input, a CNN takes tensors of shape (image_height, image_width, color_channels), ignoring the batch size. If you are new to these dimensions, color_channels refers to (R,G,B).

- In this example, you will configure our CNN to process inputs of shape (32, 32, 3), which is the format of CIFAR images. You can do this by passing the argument `input_shape` to our first layer.
- Running **`Batch Normalization`** after each convolutional layer:

In [4]:

```
model = models.Sequential()

# 1st Conv Layer + Batch Norm + MaxPooling Layer
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32,
model.add(layers.BatchNormalization())     # Batch Norm
model.add(layers.MaxPooling2D((2, 2)))

# 2nd Conv Layer + Batch Norm + MaxPooling Layer
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.BatchNormalization())     # Batch Norm
model.add(layers.MaxPooling2D((2, 2)))

# 3rd Conv Layer + Batch Norm + MaxPooling Layer
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.BatchNormalization())     # Batch Norm
```

Let's display the architecture of our model so far.

In [5]:

```
1 model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        896
_________________________________________________________________
batch_normalization (BatchNo (None, 30, 30, 32)        128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
batch_normalization_1 (Batch (None, 13, 13, 64)        256
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          36928
_________________________________________________________________
batch_normalization_2 (Batch (None, 4, 4, 64)          256
=================================================================
Total params: 56,960
Trainable params: 56,640
Non-trainable params: 320
_________________________________________________________________
```

Above, you can see that the output of every Conv2D and MaxPooling2D layer is a 3D tensor of shape (height, width, channels). The width and height dimensions tend to shrink as you go deeper in the network. The number of output channels for each Conv2D layer is controlled by the first argument (e.g., 32 or 64). Typically, as the width and height shrink, you can afford (computationally) to add more output channels in each Conv2D layer.

Add Dense layers on top

- To complete our model, you will feed the last output tensor from the convolutional base (of shape (3, 3, 64)) into one or more Dense layers to perform classification.
- Dense layers take vectors as input (which are 1D), while the current output is a 3D tensor.

1. First, you will flatten (or unroll) the 3D output to 1D.
2. Then add one or more Dense layers on top with `Batch Normalization` .
3. CIFAR has 10 output classes, so you use a final Dense layer with 10 outputs and a softmax activation.

In [6]:

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.BatchNormalization())     # Batch Norm
model.add(layers.Dense(10, activation='softmax'))
```

Here's the complete architecture of our model.

In [7]:

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        896
_________________________________________________________________
batch_normalization (BatchNo (None, 30, 30, 32)        128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
batch_normalization_1 (Batch (None, 13, 13, 64)        256
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          36928
_________________________________________________________________
batch_normalization_2 (Batch (None, 4, 4, 64)          256
_________________________________________________________________
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 64)                65600
_________________________________________________________________
batch_normalization_3 (Batch (None, 64)                256
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 123,466
Trainable params: 123,018
Non-trainable params: 448
_________________________________________________________________
```

As you can see, our (4, 4, 64) outputs were flattened into vectors of shape (1024) before going through two Dense layers.

Compile and train the model

In [8]:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
50000/50000 [==============================] - 43s 865us/sample -
loss: 1.3278 - accuracy: 0.5318 - val_loss: 1.2116 - val_accurac
y: 0.5679
Epoch 2/10
50000/50000 [==============================] - 46s 913us/sample -
loss: 0.9606 - accuracy: 0.6627 - val_loss: 1.0608 - val_accurac
y: 0.6277
Epoch 3/10
50000/50000 [==============================] - 46s 913us/sample -
loss: 0.8164 - accuracy: 0.7151 - val_loss: 1.3095 - val_accurac
y: 0.5912
Epoch 4/10
50000/50000 [==============================] - 46s 916us/sample -
loss: 0.7265 - accuracy: 0.7460 - val_loss: 0.9623 - val_accurac
y: 0.6777
Epoch 5/10
50000/50000 [==============================] - 46s 918us/sample -
loss: 0.6541 - accuracy: 0.7707 - val_loss: 0.8180 - val_accurac
y: 0.7207
Epoch 6/10
50000/50000 [==============================] - 46s 915us/sample -
loss: 0.5970 - accuracy: 0.7907 - val_loss: 0.9264 - val_accurac
y: 0.6921
Epoch 7/10
50000/50000 [==============================] - 46s 917us/sample -
loss: 0.5424 - accuracy: 0.8100 - val_loss: 0.8050 - val_accurac
y: 0.7357
Epoch 8/10
50000/50000 [==============================] - 47s 937us/sample -
loss: 0.4932 - accuracy: 0.8245 - val_loss: 0.8200 - val_accurac
y: 0.7331
Epoch 9/10
50000/50000 [==============================] - 47s 948us/sample -
loss: 0.4483 - accuracy: 0.8408 - val_loss: 0.8826 - val_accurac
y: 0.7289
Epoch 10/10
50000/50000 [==============================] - 46s 914us/sample -
loss: 0.4131 - accuracy: 0.8548 - val_loss: 0.8618 - val_accurac
y: 0.7336
```

Evaluate the model

In [15]:

```
1 history.history.keys()
```

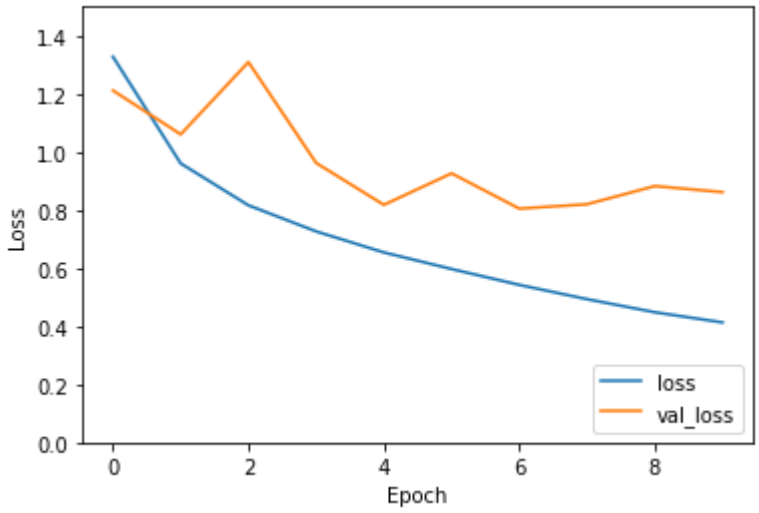
Out[15]:

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [18]:

```
1 plt.plot(history.history['loss'], label='loss')
2 plt.plot(history.history['val_loss'], label = 'val_loss')
3 plt.xlabel('Epoch')
4 plt.ylabel('Loss')
5 plt.ylim([0, 1.5])
6 plt.legend(loc='lower right')
7
8 test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
```

```
10000/1 - 3s - loss: 1.0835 - accuracy: 0.7336
```

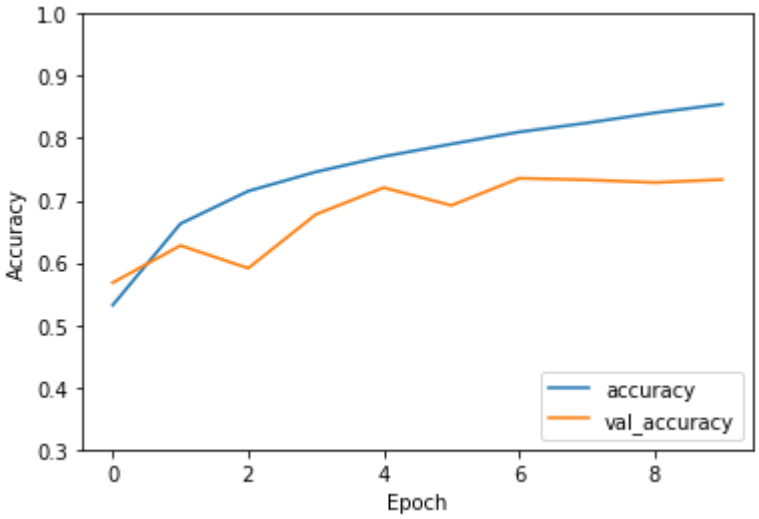


In [9]:

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.3, 1])
plt.legend(loc='lower right')

test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
```

```
10000/1 - 4s - loss: 1.0835 - accuracy: 0.7336
```



In [10]:

```
print(test_acc)
```

```
0.7336
```

> - **Our simple CNN with `Batch Normalization` has achieved a test accuracy of over 73.36%, which increases 2% accuracy more than 71.55% from the model in `Ref. 1`.**

- **Not bad for a few lines of code!**
- For another CNN style, see an example using the Keras subclassing API and a `tf.GradientTape` here (https://www.tensorflow.org/tutorials/quickstart/advanced).

In [11]:

```
test_predict = model.predict(test_images)
test_predict
```

Out[11]:

```
array([[6.60690012e-06, 1.03462753e-07, 1.41158272e-02, ...,
        8.94175173e-06, 1.76182823e-07, 4.40135857e-08],
       [9.25668776e-02, 1.22845285e-02, 8.01649148e-05, ...,
        5.81548702e-05, 8.86997938e-01, 7.88960326e-03],
       [8.10245126e-02, 2.24152133e-01, 2.54089042e-04, ...,
        1.37131382e-02, 4.59213704e-01, 2.19054461e-01],
       ...,
       [9.00711620e-06, 2.19206095e-06, 1.71041116e-02, ...,
        8.63194764e-02, 6.51853156e-07, 1.24549217e-06],
       [1.39452638e-02, 3.17217503e-03, 1.84417411e-03, ...,
        3.27146954e-05, 6.26851033e-06, 6.99571092e-06],
       [7.18135151e-09, 1.13741985e-08, 2.78889229e-06, ...,
        9.99951720e-01, 3.86602679e-08, 9.24611587e-09]], dtype=f
loat32)
```

In [12]:

```
import numpy as np
test_predict_result = np.array([np.argmax(test_predict[i]) for i in range(l
test_predict_result
```

Out[12]:

```
array([3, 8, 8, ..., 5, 4, 7], dtype=int64)
```

In [13]:

```
test_labels
```

Out[13]:

```
array([[3],
       [8],
       [8],
       ...,
       [5],
       [1],
       [7]], dtype=uint8)
```

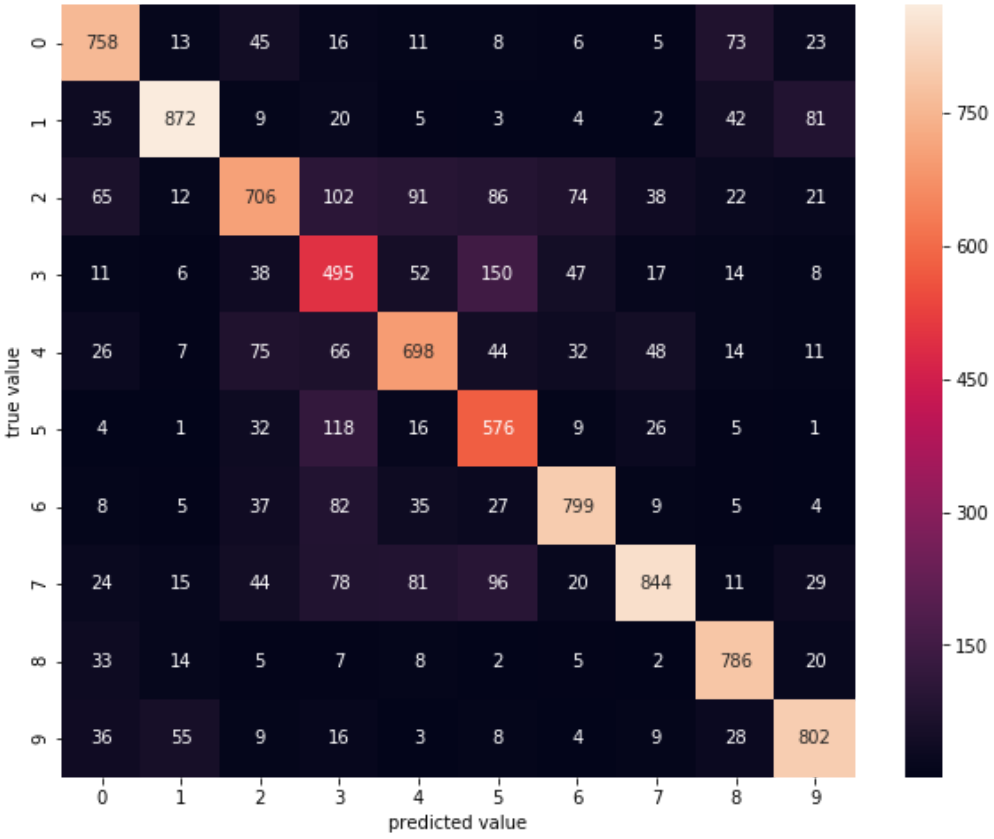
In [14]:

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

from sklearn.metrics import confusion_matrix

mat = confusion_matrix(test_predict_result, test_labels)
mat

plt.figure(figsize=(10,8))
sns.heatmap(mat, square=False, annot=True, fmt ='d', cbar=True)
plt.xlim((0, 10))
plt.ylim((10, 0))
plt.xlabel('predicted value')
plt.ylabel('true value')
plt.show()
```



Q: How to improve the performance of the CNN model above?

- *More data? Data augmentation?*
- *More Convolutional layers?*
- *Dropout?*
- *Transfer Learning (VGG-16/19, MobileNet, ResNet, etc.)?*

In [ ]:

```
1
```