Copyright 2019 The TensorFlow Authors.

Licensed under the Apache License, Version 2.0 (the "License");

```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Convolutional Neural Network (CNN)

View on TensorFlow.org Run in Google Colab View source on GitHub Download notebook

This tutorial demonstrates training a simple Convolutional Neural Network (CNN) to classify CIFAR images. Because this tutorial uses the Keras Sequential API, creating and training your model will take just a few lines of code.

Import TensorFlow

```
import tensorflow as tf
import sys
from matplotlib import pyplot
from keras.datasets import cifar10
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD
from keras.preprocessing.image import ImageDataGenerator
from keras.layers import Dropout
from keras.layers import BatchNormalization
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.

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

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

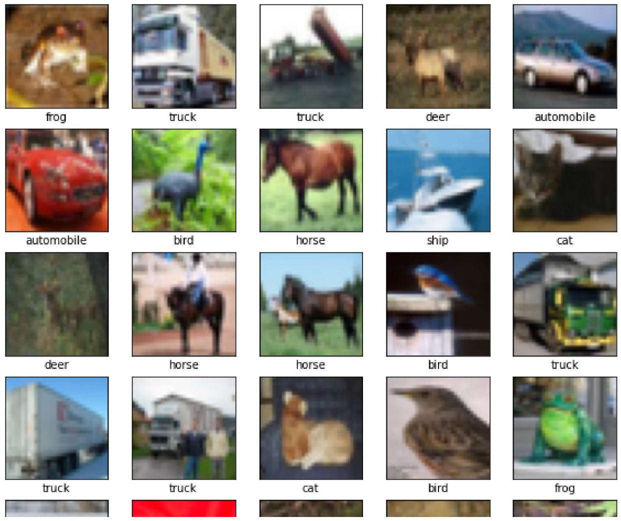
```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 14s 0us/step
```

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:

```
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])
    # 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] and [MaxPooling2D] 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 your 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 your first layer.

```
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=(32, 32, 3)))
model.add(BatchNormalization())
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.2))
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.3))
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.4))
model.add(Flatten())
model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
# compile model
opt = SGD(lr=0.001, momentum=0.9)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizers/optimizer_v2/gradient_descent.py:108: UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  super(SGD, self).__init__(name, **kwargs)
```

Let's display the architecture of your model so far:

```
model.summary()
```

```
ormalization)

conv2d_1 (Conv2D)           (None, 32, 32, 32)        9248

batch_normalization_1 (Batc  (None, 32, 32, 32)       128
hNormalization)

max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
)

dropout (Dropout)           (None, 16, 16, 32)        0

conv2d_2 (Conv2D)           (None, 16, 16, 64)        18496

batch_normalization_2 (Batc  (None, 16, 16, 64)       256
hNormalization)

conv2d_3 (Conv2D)           (None, 16, 16, 64)        36928

batch_normalization_3 (Batc  (None, 16, 16, 64)       256
hNormalization)

max_pooling2d_1 (MaxPooling  (None, 8, 8, 64)         0
```

```
2D)

dropout_1 (Dropout)         (None, 8, 8, 64)          0

conv2d_4 (Conv2D)           (None, 8, 8, 128)         73856

batch_normalization_4 (Batc  (None, 8, 8, 128)        512
hNormalization)

conv2d_5 (Conv2D)           (None, 8, 8, 128)         147584

batch_normalization_5 (Batc  (None, 8, 8, 128)        512
hNormalization)

max_pooling2d_2 (MaxPooling  (None, 4, 4, 128)        0
2D)

dropout_2 (Dropout)         (None, 4, 4, 128)         0

flatten (Flatten)           (None, 2048)              0

dense (Dense)               (None, 128)               262272

batch_normalization_6 (Batc  (None, 128)              512
hNormalization)

dropout_3 (Dropout)         (None, 128)               0

dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 552,874
Trainable params: 551,722
Non-trainable params: 1,152
_________________________________________________________________
```

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 the model, you will feed the last output tensor from the convolutional base (of shape (4, 4, 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. First, you will flatten (or unroll) the 3D output to 1D, then add one or more Dense layers on top. CIFAR has 10 output classes, so you use a final Dense layer with 10 outputs.

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
model.add(layers.Dense(128, activation='relu'))
```

Here's the complete architecture of your model:

```
model.summary()
```

```
conv2d_1 (Conv2D)           (None, 32, 32, 32)        9248

batch_normalization_1 (Batc  (None, 32, 32, 32)       128
hNormalization)

max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
)

dropout (Dropout)           (None, 16, 16, 32)        0

conv2d_2 (Conv2D)           (None, 16, 16, 64)        18496

batch_normalization_2 (Batc  (None, 16, 16, 64)       256
hNormalization)

conv2d_3 (Conv2D)           (None, 16, 16, 64)        36928

batch_normalization_3 (Batc  (None, 16, 16, 64)       256
hNormalization)

max_pooling2d_1 (MaxPooling  (None, 8, 8, 64)         0
2D)

dropout_1 (Dropout)         (None, 8, 8, 64)          0

conv2d_4 (Conv2D)           (None, 8, 8, 128)         73856

batch_normalization_4 (Batc  (None, 8, 8, 128)        512
hNormalization)

conv2d_5 (Conv2D)           (None, 8, 8, 128)         147584

batch_normalization_5 (Batc  (None, 8, 8, 128)        512
hNormalization)
```

```
 max_pooling2d_2 (MaxPooling  (None, 4, 4, 128)        0
 2D)

 dropout_2 (Dropout)         (None, 4, 4, 128)         0

 flatten (Flatten)           (None, 2048)              0

 dense (Dense)               (None, 128)               262272

 batch_normalization_6 (Batc  (None, 128)              512
 hNormalization)

 dropout_3 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 552,874
Trainable params: 551,722
Non-trainable params: 1,152
_________________________________________________________________
```

The network summary shows that (4, 4, 64) outputs were flattened into vectors of shape (1024) before going through two Dense layers.

Compile and train the model

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

history = model.fit(train_images, train_labels, epochs=150,
                    validation_data=(test_images, test_labels))
```

```
Epoch 122/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0914 - accuracy: 0.9698 - val_loss: 0.4996 - val_accuracy: 0.8796
Epoch 123/150
1563/1563 [==============================] - 13s 8ms/step - loss: 0.0891 - accuracy: 0.9703 - val_loss: 0.5229 - val_accuracy: 0.8762
Epoch 124/150
1563/1563 [==============================] - 13s 8ms/step - loss: 0.0857 - accuracy: 0.9702 - val_loss: 0.5159 - val_accuracy: 0.8783
Epoch 125/150
1563/1563 [==============================] - 13s 8ms/step - loss: 0.0856 - accuracy: 0.9708 - val_loss: 0.5059 - val_accuracy: 0.8808
Epoch 126/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0914 - accuracy: 0.9696 - val_loss: 0.4923 - val_accuracy: 0.8824
Epoch 127/150
1563/1563 [==============================] - 13s 8ms/step - loss: 0.0895 - accuracy: 0.9694 - val_loss: 0.5271 - val_accuracy: 0.8743
Epoch 128/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0874 - accuracy: 0.9702 - val_loss: 0.5038 - val_accuracy: 0.8811
Epoch 129/150
1563/1563 [==============================] - 13s 8ms/step - loss: 0.0899 - accuracy: 0.9703 - val_loss: 0.4983 - val_accuracy: 0.8815
Epoch 130/150
1563/1563 [==============================] - 13s 8ms/step - loss: 0.0875 - accuracy: 0.9706 - val_loss: 0.4873 - val_accuracy: 0.8830
Epoch 131/150
1563/1563 [==============================] - 13s 8ms/step - loss: 0.0860 - accuracy: 0.9700 - val_loss: 0.5075 - val_accuracy: 0.8802
Epoch 132/150
1563/1563 [==============================] - 13s 8ms/step - loss: 0.0889 - accuracy: 0.9696 - val_loss: 0.5069 - val_accuracy: 0.8794
Epoch 133/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0853 - accuracy: 0.9714 - val_loss: 0.4980 - val_accuracy: 0.8823
Epoch 134/150
1563/1563 [==============================] - 13s 8ms/step - loss: 0.0876 - accuracy: 0.9698 - val_loss: 0.4951 - val_accuracy: 0.8829
Epoch 135/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0876 - accuracy: 0.9703 - val_loss: 0.5006 - val_accuracy: 0.8834
Epoch 136/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0903 - accuracy: 0.9693 - val_loss: 0.4925 - val_accuracy: 0.8791
Epoch 137/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0854 - accuracy: 0.9704 - val_loss: 0.5105 - val_accuracy: 0.8828
Epoch 138/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0884 - accuracy: 0.9700 - val_loss: 0.5149 - val_accuracy: 0.8811
Epoch 139/150
1563/1563 [==============================] - 13s 8ms/step - loss: 0.0866 - accuracy: 0.9704 - val_loss: 0.4896 - val_accuracy: 0.8864
Epoch 140/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0845 - accuracy: 0.9718 - val_loss: 0.4962 - val_accuracy: 0.8852
Epoch 141/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0839 - accuracy: 0.9717 - val_loss: 0.5048 - val_accuracy: 0.8837
Epoch 142/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0891 - accuracy: 0.9690 - val_loss: 0.4946 - val_accuracy: 0.8831
Epoch 143/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0803 - accuracy: 0.9731 - val_loss: 0.5239 - val_accuracy: 0.8805
Epoch 144/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0888 - accuracy: 0.9700 - val_loss: 0.5077 - val_accuracy: 0.8802
Epoch 145/150
1563/1563 [==============================] - 13s 8ms/step - loss: 0.0835 - accuracy: 0.9726 - val_loss: 0.5234 - val_accuracy: 0.8792
Epoch 146/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0832 - accuracy: 0.9713 - val_loss: 0.5087 - val_accuracy: 0.8796
Epoch 147/150
1563/1563 [==============================] - 13s 8ms/step - loss: 0.0889 - accuracy: 0.9692 - val_loss: 0.4934 - val_accuracy: 0.8810
Epoch 148/150
1563/1563 [==============================] - 13s 8ms/step - loss: 0.0857 - accuracy: 0.9707 - val_loss: 0.4999 - val_accuracy: 0.8828
Epoch 149/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0866 - accuracy: 0.9710 - val_loss: 0.5021 - val_accuracy: 0.8829
Epoch 150/150
1563/1563 [==============================] - 12s 8ms/step - loss: 0.0826 - accuracy: 0.9714 - val_loss: 0.5209 - val_accuracy: 0.8835
```

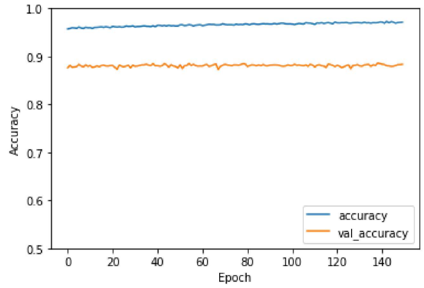
Evaluate the model

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
```

```
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')

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

```
313/313 - 1s - loss: 0.5209 - accuracy: 0.8835 - 967ms/epoch - 3ms/step
```




```
print(test_acc)
```

```
0.8834999799728394
```

Your simple CNN has achieved a test accuracy of over 70%. Not bad for a few lines of code! For another CNN style, check out the [TensorFlow 2 quickstart for experts](#) example that uses the Keras subclassing API and `tf.GradientTape`.

Colab 付费产品 - 在此处取消合同

✓ 0 秒 完成时间：14:07