Show code

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

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
```

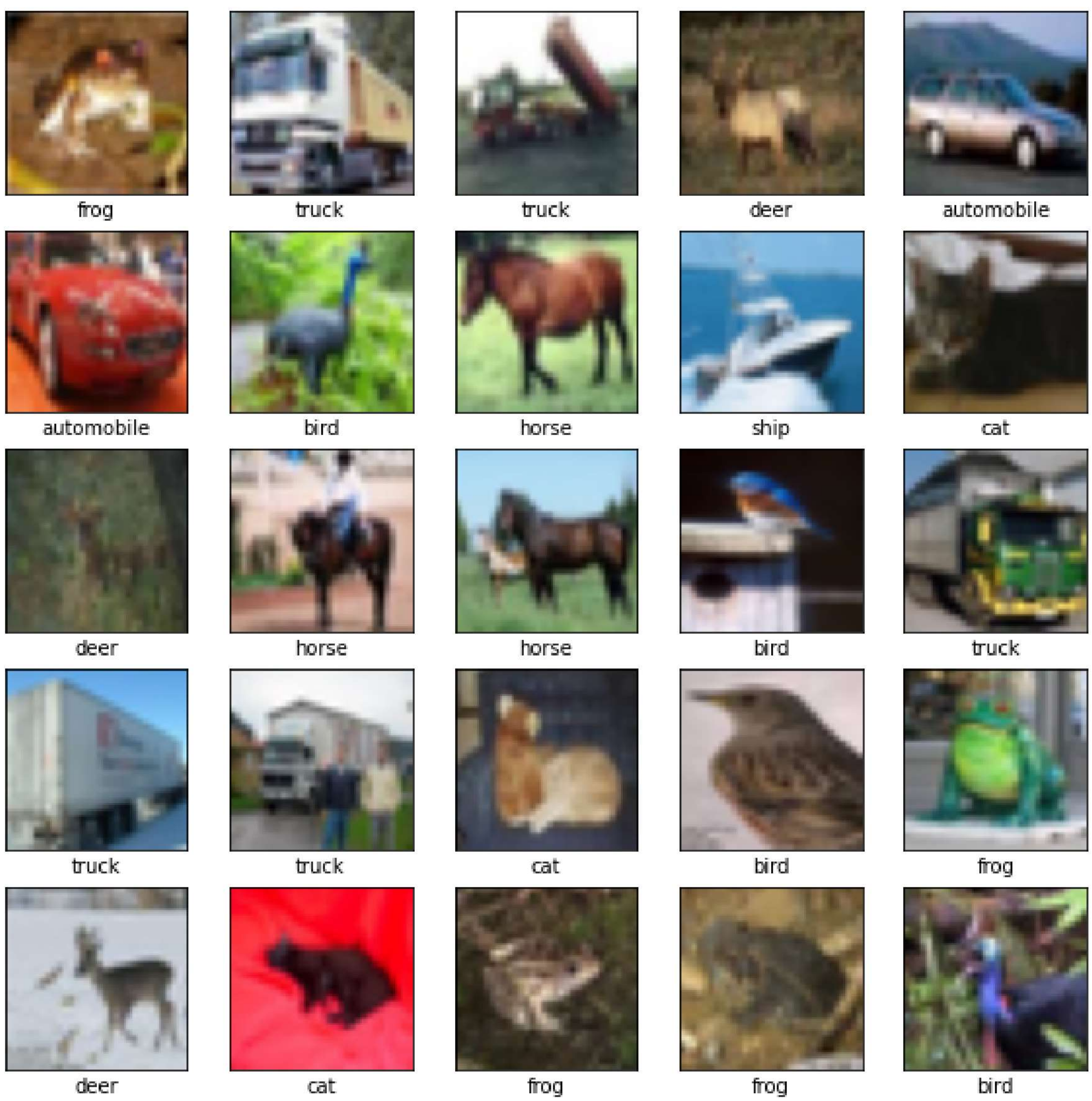
```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 6s 0us/step
```

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.

```
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.layers import Dropout
from keras.layers import BatchNormalization
model = models.Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='sam
model.add(BatchNormalization())
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='sam
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='sam
model.add(BatchNormalization())
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='sam
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='sa
model.add(BatchNormalization())
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='sa
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='relu'))
```

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

```
model.summary()
     chNormalization)

     conv2d_13 (Conv2D)          (None, 32, 32, 32)        9248

     batch_normalization_15 (Bat  (None, 32, 32, 32)       128
     chNormalization)

     max_pooling2d_6 (MaxPooling  (None, 16, 16, 32)       0
     2D)

     dropout_8 (Dropout)         (None, 16, 16, 32)        0

     conv2d_14 (Conv2D)          (None, 16, 16, 64)        18496

     batch_normalization_16 (Bat  (None, 16, 16, 64)       256
     chNormalization)

     conv2d_15 (Conv2D)          (None, 16, 16, 64)        36928

     batch_normalization_17 (Bat  (None, 16, 16, 64)       256
     chNormalization)

     max_pooling2d_7 (MaxPooling  (None, 8, 8, 64)         0
     2D)

     dropout_9 (Dropout)         (None, 8, 8, 64)          0

     conv2d_16 (Conv2D)          (None, 8, 8, 128)         73856

     batch_normalization_18 (Bat  (None, 8, 8, 128)        512
     chNormalization)

     conv2d_17 (Conv2D)          (None, 8, 8, 128)         147584

     batch_normalization_19 (Bat  (None, 8, 8, 128)        512
     chNormalization)

     max_pooling2d_8 (MaxPooling  (None, 4, 4, 128)        0
     2D)

     dropout_10 (Dropout)        (None, 4, 4, 128)         0

     flatten_2 (Flatten)         (None, 2048)              0

     dense_4 (Dense)             (None, 128)               262272


     batch_normalization_20 (Bat  (None, 128)              512
     chNormalization)

     dropout_11 (Dropout)        (None, 128)               0

     dense_5 (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))
```

Here's the complete architecture of your model:

```
model.summary()
```

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

```
from tensorflow.keras.optimizers import SGD
opt = SGD(lr=0.001, momentum=0.9)

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

history = model.fit(train_images, train_labels, epochs=300,
```

```
                    validation_data=(test_images, test_labels))
```

```
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1311 - accuracy:
Epoch 273/300
1563/1563 [==============================] - 13s 8ms/step - loss: 0.1301 - accuracy:
Epoch 274/300
1563/1563 [==============================] - 13s 8ms/step - loss: 0.1270 - accuracy:
Epoch 275/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1284 - accuracy:
Epoch 276/300
1563/1563 [==============================] - 13s 8ms/step - loss: 0.1282 - accuracy:
Epoch 277/300
1563/1563 [==============================] - 13s 8ms/step - loss: 0.1275 - accuracy:
Epoch 278/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1285 - accuracy:
Epoch 279/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1265 - accuracy:
Epoch 280/300
1563/1563 [==============================] - 13s 8ms/step - loss: 0.1252 - accuracy:
Epoch 281/300
1563/1563 [==============================] - 13s 8ms/step - loss: 0.1249 - accuracy:
Epoch 282/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1236 - accuracy:
Epoch 283/300
1563/1563 [==============================] - 13s 8ms/step - loss: 0.1264 - accuracy:
Epoch 284/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1277 - accuracy:
Epoch 285/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1267 - accuracy:
Epoch 286/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1246 - accuracy:
Epoch 287/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1297 - accuracy:
Epoch 288/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1293 - accuracy:
Epoch 289/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1202 - accuracy:
Epoch 290/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1285 - accuracy:
Epoch 291/300
1563/1563 [==============================] - 13s 8ms/step - loss: 0.1219 - accuracy:
Epoch 292/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1251 - accuracy:
Epoch 293/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1204 - accuracy:
Epoch 294/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1214 - accuracy:
Epoch 295/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1204 - accuracy:
Epoch 296/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1240 - accuracy:
Epoch 297/300
1563/1563 [==============================] - 13s 8ms/step - loss: 0.1228 - accuracy:
Epoch 298/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1223 - accuracy:
Epoch 299/300
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1218 - accuracy:
```

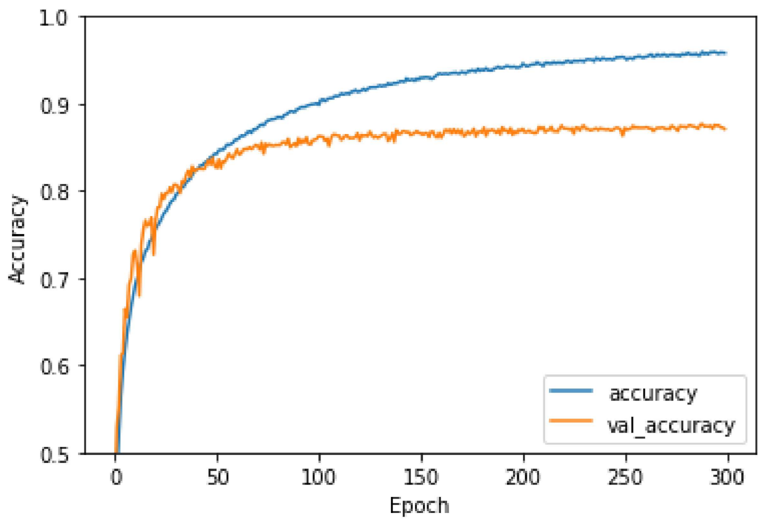
```
Epoch 300/300
1563/1563 [==============================] - 13s 8ms/step - loss: 0.1196 - accuracy:
```

Evaluate the model

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

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

```
313/313 - 1s - loss: 0.5030 - accuracy: 0.8707 - 997ms/epoch - 3ms/step
```




```
print(test_acc)
```

```
0.8707000017166138
```

```
model.save('MyGroup_CIFARmodel_baseline.h5')
```

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](https://www.tensorflow.org/tutorials/quickstart/advanced) example that uses the Keras subclassing API and `tf.GradientTape`.