CNN with tensorflow

Convolutional Neural Network (CNN)

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 our 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.

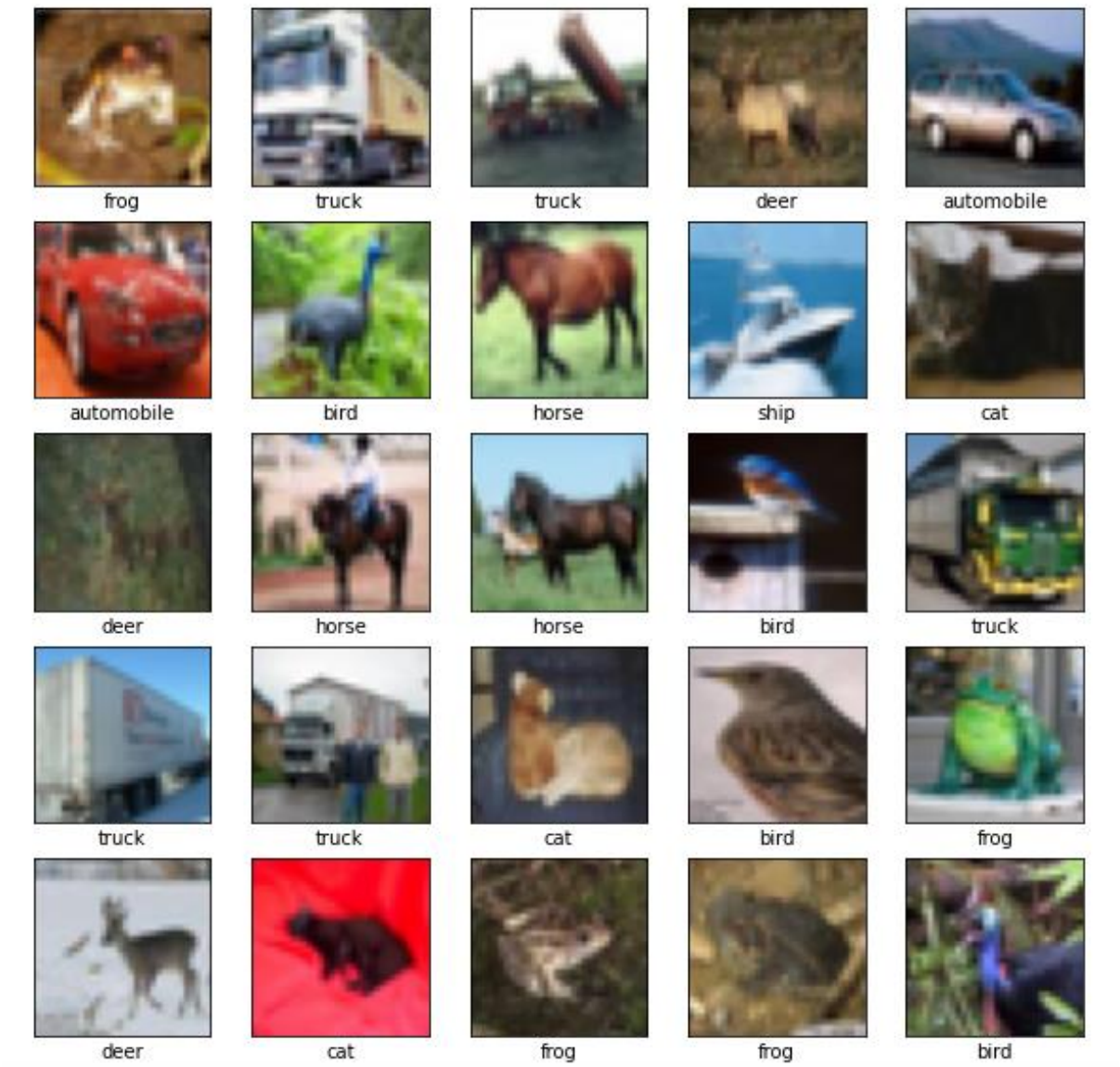
```
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

# 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
170500096/170498071 [==============================] - 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], 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 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 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.

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

```

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

```
model.summary()
```

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

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 (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 and a softmax activation.

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

Here's the complete architecture of our model.

```
1 model.summary()
2
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          36928
_________________________________________________________________
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 64)                65600
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
_________________________________________________________________
```

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

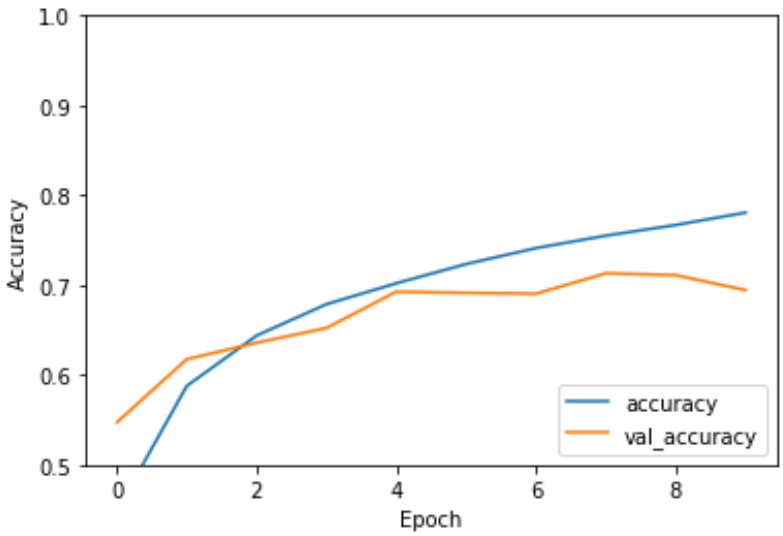
```
1 model.compile(optimizer='adam',
2               loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
3               metrics=['accuracy'])
4
5 history = model.fit(train_images, train_labels, epochs=10,
6                     validation_data=(test_images, test_labels))
7
```

```
Epoch 1/10
1563/1563 [==============================] - 65s 41ms/step - loss: 1.7444 - accuracy: 0.3514 - val_loss: 1.2495 - val_accuracy: 0.5474
Epoch 2/10
1563/1563 [==============================] - 63s 41ms/step - loss: 1.2088 - accuracy: 0.5688 - val_loss: 1.0813 - val_accuracy: 0.6176
Epoch 3/10
1563/1563 [==============================] - 62s 40ms/step - loss: 1.0274 - accuracy: 0.6373 - val_loss: 1.0251 - val_accuracy: 0.6359
Epoch 4/10
1563/1563 [==============================] - 63s 40ms/step - loss: 0.9259 - accuracy: 0.6756 - val_loss: 0.9984 - val_accuracy: 0.6523
Epoch 5/10
1563/1563 [==============================] - 62s 40ms/step - loss: 0.8645 - accuracy: 0.6961 - val_loss: 0.8952 - val_accuracy: 0.6927
Epoch 6/10
1563/1563 [==============================] - 62s 39ms/step - loss: 0.8022 - accuracy: 0.7212 - val_loss: 0.9012 - val_accuracy: 0.6914
Epoch 7/10
1563/1563 [==============================] - 62s 40ms/step - loss: 0.7402 - accuracy: 0.7418 - val_loss: 0.9034 - val_accuracy: 0.6904
Epoch 8/10
1563/1563 [==============================] - 62s 40ms/step - loss: 0.6987 - accuracy: 0.7559 - val_loss: 0.8622 - val_accuracy: 0.7132
Epoch 9/10
1563/1563 [==============================] - 63s 40ms/step - loss: 0.6546 - accuracy: 0.7686 - val_loss: 0.8790 - val_accuracy: 0.7111
Epoch 10/10
1563/1563 [==============================] - 63s 40ms/step - loss: 0.6188 - accuracy: 0.7832 - val_loss: 0.9414 - val_accuracy: 0.6945
```

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 - 3s - loss: 0.9414 - accuracy: 0.6945
```




```
print(test_acc)

```

```
0.6945000290870667
```

Image classification

This tutorial shows how to classify images of flowers. It creates an image classifier using a `keras.Sequential` model, and loads data using `preprocessing.image_dataset_from_directory`. You will gain practical experience with the following concepts:

- Efficiently loading a dataset off disk.
- Identifying overfitting and applying techniques to mitigate it, including data augmentation and Dropout.

This tutorial follows a basic machine learning workflow:

1. Examine and understand data
2. Build an input pipeline

3. Build the model
4. Train the model
5. Test the model
6. Improve the model and repeat the process

Import TensorFlow and other libraries

```
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential

```

Download and explore the dataset

This tutorial uses a dataset of about 3,700 photos of flowers. The dataset contains 5 sub-directories, one per class:

```
flower_photo/
  daisy/
  dandelion/
  roses/
  sunflowers/
  tulips/

```

```
import pathlib
dataset_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz"
data_dir = tf.keras.utils.get_file('flower_photos', origin=dataset_url, untar=True)
data_dir = pathlib.Path(data_dir)

```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz
228818944/228813984 [==============================] - 4s 0us/step
```

After downloading, you should now have a copy of the dataset available. There are 3,670 total images

```
image_count = len(list(data_dir.glob('*/*.jpg')))
print(image_count)

```

```
3670
```

Here are some roses:

```
1 roses = list(data_dir.glob('roses/*'))
2 PIL.Image.open(str(roses[0]))
3
```

```
1 PIL.Image.open(str(roses[1]))
2
```

And some tulips:

```
1 tulips = list(data_dir.glob('tulips/*'))
2 PIL.Image.open(str(tulips[0]))
3
```

Load using keras.preprocessing

Let's load these images off disk using the helpful image_dataset_from_directory utility. This will take you from a directory of images on disk to a `tf.data.Dataset` in just a couple lines of code. If you like, you can also write your own data loading code from scratch

Create a dataset

Define some parameters for the loader:

```
batch_size = 32
img_height = 180
img_width = 180

```

It's good practice to use a validation split when developing your model. Let's use 80% of the images for training, and 20% for validation.

Training set

```
train_ds = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)

```

```
Found 3670 files belonging to 5 classes.
Using 2936 files for training.
```

Validation set

```
val_ds = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)

```

```
Found 3670 files belonging to 5 classes.
Using 734 files for validation.
```

You can find the class names in the `class_names` attribute on these datasets. These correspond to the directory names in alphabetical order.

```
class_names = train_ds.class_names
print(class_names)

```

```
['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
```

Visualize the data

Here are the first 9 images from the training dataset.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(images[i].numpy().astype("uint8"))
    plt.title(class_names[labels[i]])
    plt.axis("off")

```

You will train a model using these datasets by passing them to `model.fit` in a moment. If you like, you can also manually iterate over the dataset and retrieve batches of images:

The `image_batch` is a tensor of the shape `(32, 180, 180, 3)`. This is a batch of 32 images of shape `180x180x3` (the last dimension refers to color channels RGB). The `label_batch` is a tensor of the shape `(32,)`, these are corresponding labels to the 32 images.

You can call `.numpy()` on the `image_batch` and `labels_batch` tensors to convert them to a `numpy.ndarray`.

Configure the dataset for performance

Let's make sure to use buffered prefetching so you can yield data from disk without having I/O become blocking. These are two important methods you should use when loading data.

`Dataset.cache()` keeps the images in memory after they're loaded off disk during the first epoch. This will ensure the dataset does not become a bottleneck while training your model. If your dataset is too large to fit into memory, you can also use this method to create a performant on-disk cache.

`Dataset.prefetch()` overlaps data preprocessing and model execution while training.

Interested readers can learn more about both methods, as well as how to cache data to disk in the data performance guide.

```
AUTOTUNE = tf.data.experimental.AUTOTUNE

train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)

```

Standardize the data

The RGB channel values are in the `[0, 255]` range. This is not ideal for a neural network; in general you should seek to make your input values small. Here, you will standardize values to be in the `[0, 1]` range by using a Rescaling layer.

```
normalization_layer = layers.experimental.preprocessing.Rescaling(1./255)

```

There are two ways to use this layer. You can apply it to the dataset by calling map:

```
normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
image_batch, labels_batch = next(iter(normalized_ds))
first_image = image_batch[0]
# Notice the pixels values are now in `[0,1]`.
print(np.min(first_image), np.max(first_image))

```

```
0.0 0.9870374
```

Or, you can include the layer inside your model definition, which can simplify deployment. We'll see second approach later.

Create the model

The model consists of three convolution blocks with a max pool layer in each of them. There's a fully connected layer with 128 units on top of it that is activated by a `relu` activation function. This model has not been tuned for high accuracy, the goal of this tutorial is to show a standard approach

```
num_classes = 5

model = Sequential([
  layers.experimental.preprocessing.Rescaling(1./255, input_shape=(img_height, img_width, 3)),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])

```

Compile the model

For this tutorial, choose the `optimizers.Adam` optimizer and `losses.SparseCategoricalCrossentropy` loss function. To view training and validation accuracy for each training epoch, pass the `metrics` argument.

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

```

Model summary

View all the layers of the network using the model's `summary` method:

```
1 model.summary()
2
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
rescaling_1 (Rescaling)      (None, 180, 180, 3)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 180, 180, 16)      448
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 90, 90, 16)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 90, 90, 32)        4640
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 45, 45, 32)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 45, 45, 64)        18496
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 22, 22, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 30976)             0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               3965056
_________________________________________________________________
dense_3 (Dense)              (None, 5)                 645
=================================================================
Total params: 3,989,285
Trainable params: 3,989,285
Non-trainable params: 0
```

Train the model

```
1 epochs=10
2 history = model.fit(
3   train_ds,
4   validation_data=val_ds,
5   epochs=epochs
6 )
7
```

```
Epoch 1/10
92/92 [==============================] - 85s 918ms/step - loss: 1.6004 - accuracy: 0.3601 - val_loss: 1.0398 - val_accuracy: 0.5899
Epoch 2/10
92/92 [==============================] - 82s 896ms/step - loss: 0.9737 - accuracy: 0.6153 - val_loss: 0.9661 - val_accuracy: 0.6158
Epoch 3/10
92/92 [==============================] - 82s 896ms/step - loss: 0.8063 - accuracy: 0.6982 - val_loss: 0.9045 - val_accuracy: 0.6417
Epoch 4/10
92/92 [==============================] - 83s 899ms/step - loss: 0.5645 - accuracy: 0.8061 - val_loss: 0.8798 - val_accuracy: 0.6676
Epoch 5/10
92/92 [==============================] - 83s 902ms/step - loss: 0.3209 - accuracy: 0.8996 - val_loss: 1.0014 - val_accuracy: 0.6376
Epoch 6/10
92/92 [==============================] - 83s 902ms/step - loss: 0.1637 - accuracy: 0.9577 - val_loss: 1.2816 - val_accuracy: 0.6185
Epoch 7/10
92/92 [==============================] - 83s 903ms/step - loss: 0.1137 - accuracy: 0.9672 - val_loss: 1.2762 - val_accuracy: 0.6403
Epoch 8/10
92/92 [==============================] - 83s 904ms/step - loss: 0.0525 - accuracy: 0.9913 - val_loss: 1.7757 - val_accuracy: 0.6471
Epoch 9/10
92/92 [==============================] - 83s 904ms/step - loss: 0.0468 - accuracy: 0.9905 - val_loss: 1.6880 - val_accuracy: 0.6322
Epoch 10/10
92/92 [==============================] - 83s 900ms/step - loss: 0.0302 - accuracy: 0.9934 - val_loss: 1.8118 - val_accuracy: 0.6417
```

Visualize training results

Create plots of loss and accuracy on the training and validation sets.

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

As you can see from the plots, training accuracy and validation accuracy are off by large margin and the model has achieved only around 60% accuracy on the validation set.

Let's look at what went wrong and try to increase the overall performance of the model.

Data augmentation

Overfitting generally occurs when there are a small number of training examples. Data augmentation takes the approach of generating additional training data from your existing examples by augmenting them using random transformations that yield believable-looking images. This helps expose the model to more aspects of the data and generalize better.

You will implement data augmentation using experimental Keras Preprocessing Layers. These can be included inside your model like other layers, and run on the GPU.

```
Let's visualize what a few augmented examples look like by applying
data augmentation to the same image several times:
```

You will use data augmentation to train a model in a moment.

Dropout

Another technique to reduce overfitting is to introduce Dropout to the network, a form of *regularization*.

When you apply Dropout to a layer it randomly drops out (by setting the activation to zero) a number of output units from the layer during the training process. Dropout takes a fractional number as its input value, in the form such as 0.1, 0.2, 0.4, etc. This means dropping out 10%, 20% or 40% of the output units randomly from the applied layer.

Let's create a new neural network using `layers.Dropout`, then train it using augmented images.

```
model = Sequential([
  data_augmentation,
  layers.experimental.preprocessing.Rescaling(1./255),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Dropout(0.2),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])

```

Compile and train the model

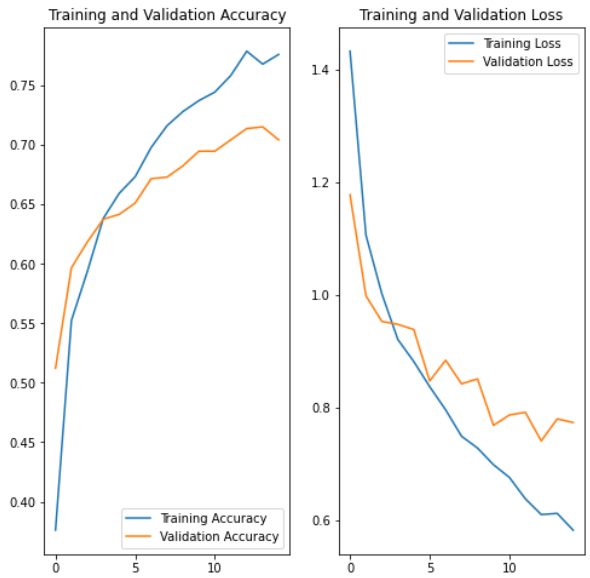
```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

```

```
1 model.summary()
2
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential_2 (Sequential)    (None, 180, 180, 3)       0
_________________________________________________________________
rescaling_2 (Rescaling)      (None, 180, 180, 3)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 180, 180, 16)      448
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 90, 90, 16)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 90, 90, 32)        4640
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 45, 45, 32)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 45, 45, 64)        18496
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 22, 22, 64)        0
_________________________________________________________________
dropout (Dropout)            (None, 22, 22, 64)        0
_________________________________________________________________
flatten_2 (Flatten)          (None, 30976)             0
_________________________________________________________________
dense_4 (Dense)              (None, 128)               3965056
_________________________________________________________________
dense_5 (Dense)              (None, 5)                 645
=================================================================
Total params: 3,989,285
Trainable params: 3,989,285
Non-trainable params: 0
```

Visualize training results

After applying data augmentation and Dropout, there is less overfitting than before, and training and validation accuracy are closer aligned.



Predict on new data

Finally, let's use our model to classify an image that wasn't included in the training or validation sets

```
sunflower_url = "https://storage.googleapis.com/download.tensorflow.org/example_images/592px-Red_sunflower.jpg"
sunflower_path = tf.keras.utils.get_file('Red_sunflower', origin=sunflower_url)

img = keras.preprocessing.image.load_img(
    sunflower_path, target_size=(img_height, img_width)
)
img_array = keras.preprocessing.image.img_to_array(img)
img_array = tf.expand_dims(img_array, 0) # Create a batch

predictions = model.predict(img_array)
score = tf.nn.softmax(predictions[0])

print(
    "This image most likely belongs to {} with a {:.2f} percent confidence."
    .format(class_names[np.argmax(score)], 100 * np.max(score))
)

```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/example_images/592px-Red_su
122880/117948 [==============================] - 0s 0us/step
This image most likely belongs to sunflowers with a 99.45 percent confidence.
```