Notebook credit: Based on the original D2L notebook here.

Convolutional Neural Networks (LeNet)

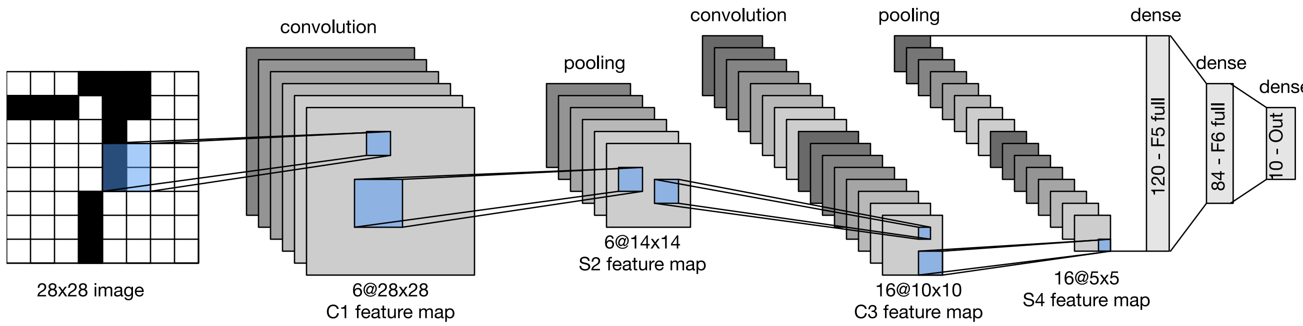
We now have all the ingredients required to assemble a fully-functional CNN. In our earlier encounter with image data, we applied a softmax regression model and an MLP model to MNIST dataset. To make such data amenable to softmax regression and MLPs, we first flattened each image from a 28×28 matrix into a fixed-length 784-dimensional vector, and thereafter processed them with fully-connected layers. Now that we have a handle on convolutional layers, we can retain the spatial structure in our images. As an additional benefit of replacing fully-connected layers with convolutional layers, we will enjoy more parsimonious models that require far fewer parameters.

In this section, we will introduce *LeNet*, among the first published CNNs to capture wide attention for its performance on computer vision tasks. The model was introduced by (and named for) Yann LeCun, then a researcher at AT&T Bell Labs, for the purpose of recognizing handwritten digits in images. This work represented the culmination of a decade of research developing the technology. In 1989, LeCun published the first study to successfully train CNNs via backpropagation.

At the time LeNet achieved outstanding results matching the performance of support vector machines, then a dominant approach in supervised learning. LeNet was eventually adapted to recognize digits for processing deposits in ATM machines. To this day, some ATMs still run the code that Yann and his colleague Leon Bottou wrote in the 1990s!

LeNet

At a high level, (**LeNet (LeNet-5) consists of two parts: (i) a convolutional encoder consisting of two convolutional layers; and (ii) a dense block consisting of three fully-connected layers**); The architecture is summarized below.



The basic units in each convolutional block are a convolutional layer, a sigmoid activation function, and a subsequent average pooling operation. Note that while ReLUs and max-pooling work better, these discoveries had not yet been made in the 1990s. Each convolutional layer uses a 5×5 kernel and a sigmoid activation function. These layers map spatially arranged inputs to a number of two-dimensional feature maps, typically increasing the number of channels. The first convolutional layer has 6 output channels, while the second has 16. Each 2×2 pooling operation (stride 2) reduces dimensionality by a factor of 4 via spatial downsampling. The convolutional block emits an output with shape given by (batch size, number of channel, height, width).

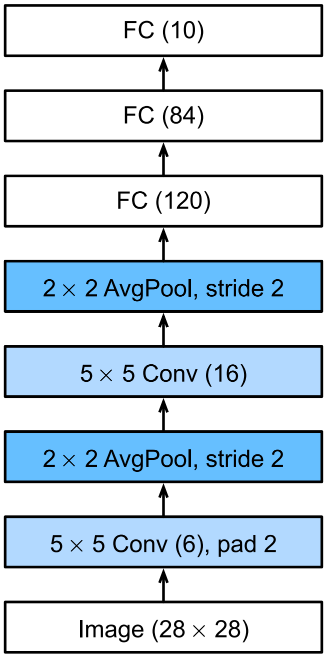
In order to pass output from the convolutional block to the dense block, we must flatten each example in the minibatch. In other words, we take this four-dimensional input and transform it into the two-dimensional input expected by fully-connected layers: as a reminder, the two-dimensional representation that we desire uses the first dimension to index examples in the minibatch and the second to give the flat vector representation of each example. LeNet's dense block has three fully-connected layers, with 120, 84, and 10 outputs, respectively. Because we are still performing classification, the 10-dimensional output layer corresponds to the number of possible output classes.

While getting to the point where you truly understand what is going on inside LeNet may have taken a bit of work, hopefully the following code snippet will convince you that implementing such models with modern deep learning frameworks is remarkably simple. We need only to instantiate a `Sequential` block and chain together the appropriate layers.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

We took a small liberty with the original model, removing the Gaussian activation in the final layer. Other than that, this network matches the original LeNet-5 architecture.

By passing a single-channel (black and white) 28×28 image through the network and printing the output shape at each layer, we can [**inspect the model**] to make sure that its operations line up with what we expect from the figure above.



Note that the height and width of the representation at each layer throughout the convolutional block is reduced (compared with the previous layer). The first convolutional layer uses 2 pixels of padding to compensate for the reduction in height and width that would otherwise result from using a 5×5 kernel. In contrast, the second convolutional layer forgoes padding, and thus the height and width are both reduced by 4 pixels. As we go up the stack of layers, the number of channels increases layer-over-layer from 1 in the input to 6 after the first convolutional layer and 16 after the second convolutional layer. However, each pooling layer halves the height and width. Finally, each fully-connected layer reduces dimensionality, finally emitting an output whose dimension matches the number of classes.

Training

Now that we have implemented the model, let us [**run an experiment to see how LeNet fares on Fashion-MNIST**].

```
batch_size = 256
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
29515/29515 [==============================] - 0s 0us/step
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
26421880/26421880 [==============================] - 1s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
4422102/4422102 [==============================] - 0s 0us/step
```

While CNNs have fewer parameters, they can still be more expensive to compute than similarly deep MLPs because each parameter participates in many more multiplications. **If you have access to a GPU, this might be a good time to put it into action to speed up training.**

[**Now let us train and evaluate the LeNet-5 model.**]

```
num_epochs = 20
model = keras.Sequential([
        layers.Conv2D(filters=6, kernel_size=5, activation='sigmoid',
                               padding='same', input_shape=(28, 28, 1)),
        layers.AvgPool2D(pool_size=2, strides=2),
        layers.Conv2D(filters=16, kernel_size=5,
                               activation='sigmoid'),
        layers.AvgPool2D(pool_size=2, strides=2),
        layers.Flatten(),
        layers.Dense(120, activation='sigmoid'),
        layers.Dense(84, activation='sigmoid'),
        layers.Dense(10, activation='softmax')])
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

```
history = model.fit(x_train, y_train, epochs=num_epochs, batch_size=batch_size, vali
```

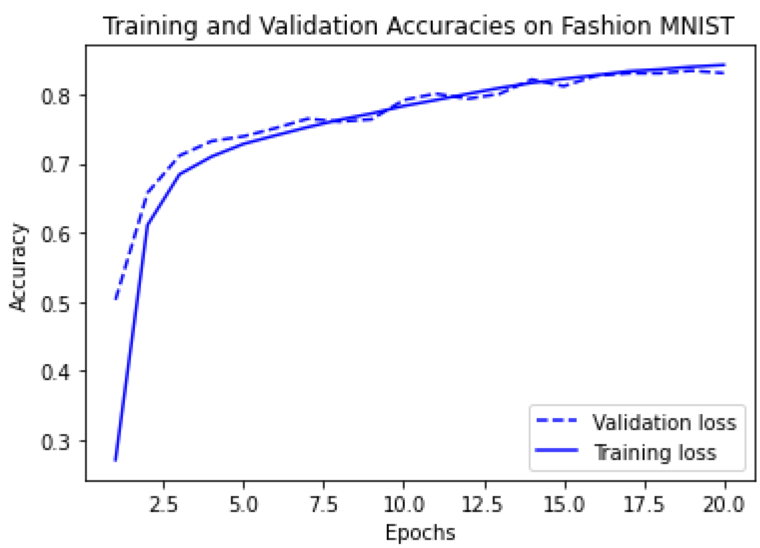
```
Epoch 1/20
188/188 [==============================] - 15s 7ms/step - loss: 2.0212 - accurac
Epoch 2/20
188/188 [==============================] - 1s 7ms/step - loss: 1.0770 - accuracy
Epoch 3/20
188/188 [==============================] - 1s 6ms/step - loss: 0.8575 - accuracy
Epoch 4/20
188/188 [==============================] - 1s 5ms/step - loss: 0.7761 - accuracy
Epoch 5/20
188/188 [==============================] - 1s 5ms/step - loss: 0.7160 - accuracy
Epoch 6/20
188/188 [==============================] - 1s 5ms/step - loss: 0.6734 - accuracy
Epoch 7/20
188/188 [==============================] - 1s 5ms/step - loss: 0.6413 - accuracy
Epoch 8/20
188/188 [==============================] - 1s 6ms/step - loss: 0.6145 - accuracy
Epoch 9/20
```

```
188/188 [==============================] - 1s 5ms/step - loss: 0.5907 - accuracy
Epoch 10/20
188/188 [==============================] - 1s 5ms/step - loss: 0.5663 - accuracy
Epoch 11/20
188/188 [==============================] - 1s 5ms/step - loss: 0.5446 - accuracy
Epoch 12/20
188/188 [==============================] - 1s 5ms/step - loss: 0.5253 - accuracy
Epoch 13/20
188/188 [==============================] - 1s 7ms/step - loss: 0.5074 - accuracy
Epoch 14/20
188/188 [==============================] - 1s 7ms/step - loss: 0.4906 - accuracy
Epoch 15/20
188/188 [==============================] - 1s 5ms/step - loss: 0.4779 - accuracy
Epoch 16/20
188/188 [==============================] - 1s 5ms/step - loss: 0.4653 - accuracy
Epoch 17/20
188/188 [==============================] - 1s 5ms/step - loss: 0.4549 - accuracy
Epoch 18/20
188/188 [==============================] - 1s 5ms/step - loss: 0.4454 - accuracy
Epoch 19/20
188/188 [==============================] - 1s 5ms/step - loss: 0.4372 - accuracy
Epoch 20/20
188/188 [==============================] - 1s 5ms/step - loss: 0.4293 - accuracy
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 6)         156

 average_pooling2d (AverageP  (None, 14, 14, 6)        0
 ooling2D)

 conv2d_1 (Conv2D)           (None, 10, 10, 16)        2416

 average_pooling2d_1 (Averag  (None, 5, 5, 16)         0
 ePooling2D)

 flatten (Flatten)           (None, 400)               0

 dense (Dense)               (None, 120)               48120

 dense_1 (Dense)             (None, 84)                10164

 dense_2 (Dense)             (None, 10)                850

=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
_________________________________________________________________
```

```
import matplotlib.pyplot as plt

val_acc = history.history["val_accuracy"]
acc = history.history["accuracy"]
epochs = range(1, 21)
plt.plot(epochs, val_acc, "b--",
         label="Validation loss")
plt.plot(epochs, acc, "b",
         label="Training loss")
plt.title("Training and Validation Accuracies on Fashion MNIST")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```




```
model.evaluate(x_test, y_test)
```

```
313/313 [==============================] - 2s 5ms/step - loss: 0.4747 - accuracy
[0.4747280180454254, 0.8259999752044678]
```

Function API

We have only used the Sequential API in Keras to build deep learning model. However, there are two more choices:

- Functional API
- Model subclassing

Let's look at each of them to redo the LeNet example above. If your architecture is not just a simple linear sequence of layers, you cannot use the Sequential API. A common use case is **residual connections**. For example, see this toy ResNet model.

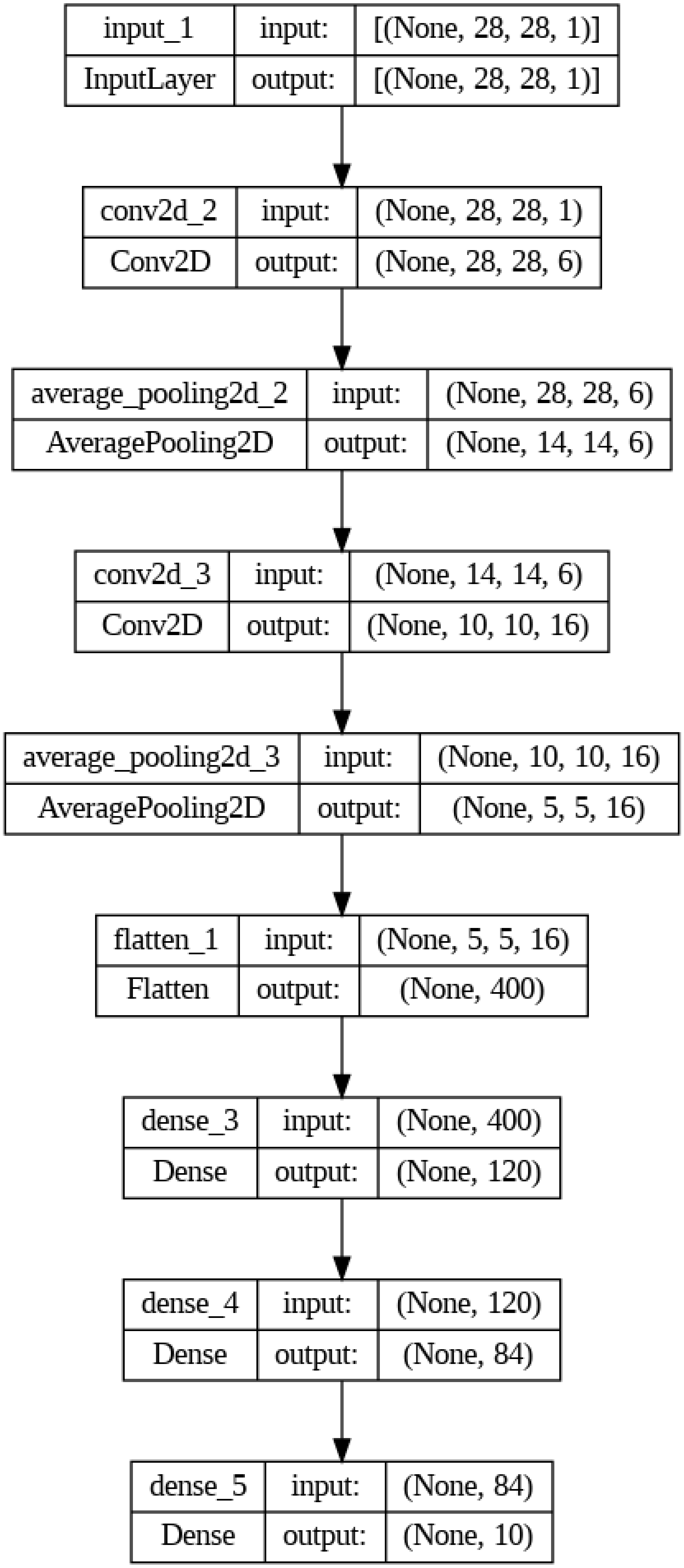
```
inputs = keras.Input(shape=(28,28,1))

conv1 = layers.Conv2D(filters=6, kernel_size=5, activation='sigmoid', padding='same'
x = conv1(inputs)

pool1 = layers.AvgPool2D(pool_size=2, strides=2)
x = pool1(x)

conv2 = layers.Conv2D(filters=16, kernel_size=5, activation='sigmoid')
x = conv2(x)

pool2 = layers.AvgPool2D(pool_size=2, strides=2)
x = pool2(x)

flatten = layers.Flatten()
x = flatten(x)

fc1 = layers.Dense(120, activation='sigmoid')
x = fc1(x)

fc2 = layers.Dense(84, activation='sigmoid')
x = fc2(x)

fc3 = layers.Dense(10, activation='softmax')
outputs = fc3(x)

model = keras.Model(inputs=inputs, outputs=outputs, name="LeNet_model")
```

```
model.summary()
```

```
Model: "LeNet_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d_2 (Conv2D)           (None, 28, 28, 6)         156

 average_pooling2d_2 (Averag  (None, 14, 14, 6)        0
 ePooling2D)

 conv2d_3 (Conv2D)           (None, 10, 10, 16)        2416

 average_pooling2d_3 (Averag  (None, 5, 5, 16)         0
 ePooling2D)

 flatten_1 (Flatten)         (None, 400)               0
```

```
 dense_3 (Dense)             (None, 120)               48120

 dense_4 (Dense)             (None, 84)                10164

 dense_5 (Dense)             (None, 10)                850

=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
_________________________________________________________________
```

```
keras.utils.plot_model(model, show_shapes=True)
```


input_1
input:
[(None, 28, 28, 1)]
InputLayer
output:
[(None, 28, 28, 1)]
conv2d_2
input:
(None, 28, 28, 1)
Conv2D
output:
(None, 28, 28, 6)
average_pooling2d_2
input:
(None, 28, 28, 6)
AveragePooling2D
output:
(None, 14, 14, 6)
conv2d_3
input:
(None, 14, 14, 6)
Conv2D
output:
(None, 10, 10, 16)
average_pooling2d_3
input:
(None, 10, 10, 16)
AveragePooling2D
output:
(None, 5, 5, 16)
flatten_1
input:
(None, 5, 5, 16)
Flatten
output:
(None, 400)
dense_3
input:
(None, 400)
Dense
output:
(None, 120)
dense_4
input:
(None, 120)
Dense
output:
(None, 84)
dense_5
input:
(None, 84)
Dense
output:
(None, 10)

Model fitting works exactly the same as with sequential API.

```
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

```
history = model.fit(x_train, y_train, epochs=num_epochs, batch_size=batch_size, vali
```

```
Epoch 1/20
188/188 [==============================] - 6s 16ms/step - loss: 2.1938 - accurac
Epoch 2/20
188/188 [==============================] - 2s 8ms/step - loss: 1.1798 - accuracy
Epoch 3/20
188/188 [==============================] - 2s 8ms/step - loss: 0.8607 - accuracy
Epoch 4/20
188/188 [==============================] - 1s 6ms/step - loss: 0.7500 - accuracy
Epoch 5/20
188/188 [==============================] - 1s 8ms/step - loss: 0.6978 - accuracy
Epoch 6/20
188/188 [==============================] - 2s 8ms/step - loss: 0.6645 - accuracy
Epoch 7/20
188/188 [==============================] - 2s 9ms/step - loss: 0.6362 - accuracy
Epoch 8/20
188/188 [==============================] - 1s 8ms/step - loss: 0.6126 - accuracy
Epoch 9/20
188/188 [==============================] - 2s 8ms/step - loss: 0.5883 - accuracy
Epoch 10/20
188/188 [==============================] - 1s 8ms/step - loss: 0.5658 - accuracy
Epoch 11/20
188/188 [==============================] - 1s 7ms/step - loss: 0.5464 - accuracy
Epoch 12/20
188/188 [==============================] - 1s 8ms/step - loss: 0.5280 - accuracy
Epoch 13/20
188/188 [==============================] - 1s 7ms/step - loss: 0.5118 - accuracy
Epoch 14/20
188/188 [==============================] - 1s 7ms/step - loss: 0.4976 - accuracy
Epoch 15/20
188/188 [==============================] - 1s 7ms/step - loss: 0.4841 - accuracy
Epoch 16/20
188/188 [==============================] - 2s 8ms/step - loss: 0.4721 - accuracy
Epoch 17/20
188/188 [==============================] - 1s 7ms/step - loss: 0.4627 - accuracy
Epoch 18/20
188/188 [==============================] - 1s 7ms/step - loss: 0.4517 - accuracy
Epoch 19/20
188/188 [==============================] - 1s 6ms/step - loss: 0.4431 - accuracy
Epoch 20/20
188/188 [==============================] - 1s 6ms/step - loss: 0.4363 - accuracy
```

Model Subclassing

```
class LeNet_Model(keras.Model):

  def __init__(self, num_classes, input_shape):
    super(LeNet_Model, self).__init__()

    self.conv1 = layers.Conv2D(filters=6, kernel_size=5, activation='sigmoid', paddi
    self.pool1 = layers.AvgPool2D(pool_size=2, strides=2)
    self.conv2 = layers.Conv2D(filters=16, kernel_size=5, activation='sigmoid')
    self.pool2 = layers.AvgPool2D(pool_size=2, strides=2)
    self.flatten = layers.Flatten()
    self.fc1 = layers.Dense(120, activation='sigmoid')
    self.fc2 = layers.Dense(84, activation='sigmoid')
    self.fc3 = layers.Dense(num_classes, activation='softmax')

  def call(self, inputs):
    x = self.conv1(inputs)
    x = self.pool1(x)
    x = self.conv2(x)
    x = self.pool2(x)
    x = self.flatten(x)
    x = self.fc1(x)
    x = self.fc2(x)
    outputs = self.fc3(x)
    return outputs
```

```
lenet_model = LeNet_Model(10, (None, 28, 28, 1))
lenet_model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

```
x_train = tf.expand_dims(x_train, -1)
x_train.shape
```

```
TensorShape([60000, 28, 28, 1])
```

```
history = lenet_model.fit(x_train, y_train, epochs=num_epochs, batch_size=batch_size
```

```
Epoch 1/20
188/188 [==============================] - 3s 7ms/step - loss: 1.9922 - accuracy
Epoch 2/20
188/188 [==============================] - 1s 6ms/step - loss: 1.0558 - accuracy
Epoch 3/20
188/188 [==============================] - 1s 6ms/step - loss: 0.8290 - accuracy
Epoch 4/20
188/188 [==============================] - 1s 5ms/step - loss: 0.7360 - accuracy
Epoch 5/20
```

```
188/188 [==============================] - 1s 5ms/step - loss: 0.6897 - accuracy
Epoch 6/20
188/188 [==============================] - 1s 5ms/step - loss: 0.6597 - accuracy
Epoch 7/20
188/188 [==============================] - 1s 5ms/step - loss: 0.6368 - accuracy
Epoch 8/20
188/188 [==============================] - 1s 8ms/step - loss: 0.6161 - accuracy
Epoch 9/20
188/188 [==============================] - 1s 7ms/step - loss: 0.5979 - accuracy
Epoch 10/20
188/188 [==============================] - 1s 6ms/step - loss: 0.5775 - accuracy
Epoch 11/20
188/188 [==============================] - 1s 6ms/step - loss: 0.5599 - accuracy
Epoch 12/20
188/188 [==============================] - 1s 6ms/step - loss: 0.5433 - accuracy
Epoch 13/20
```