tf-demo

May 8, 2019

1 Image classification on the MNIST data set using convolutional neural networks

[1]:
```
import matplotlib.pyplot as plt
import numpy as np

from tensorflow.data import Dataset, experimental
from tensorflow.keras.backend import clear_session
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

clear_session() # Reset notebook state
```

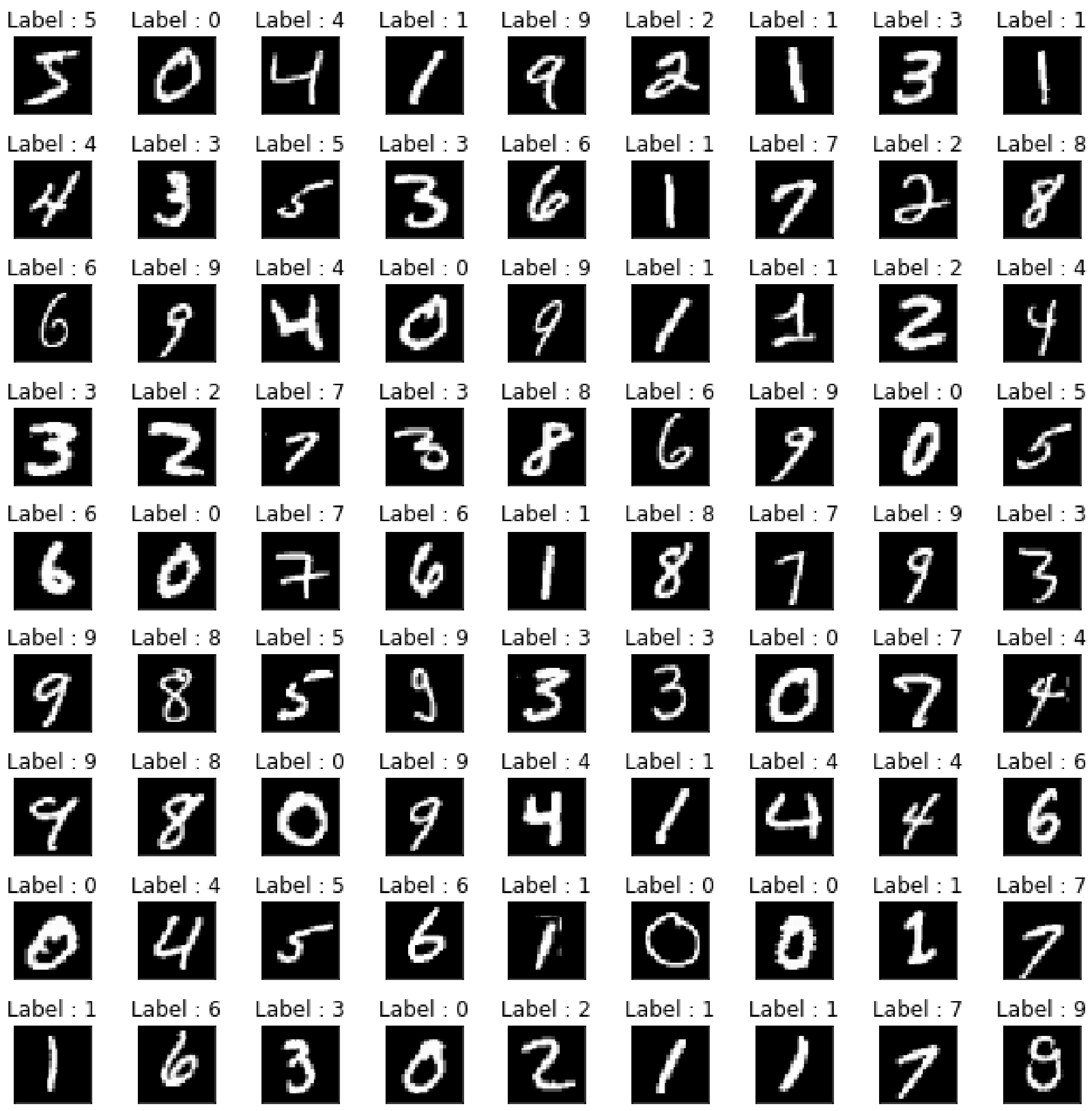
We're using MNIST, a database of hand-written digits commonly used for training and testing image processing and machine learning techniques. The database is split into 60,000 training examples and 10,000 test examples. The digits are stored as 28-pixel 8-bit grayscale images. For each digit, a numerical label of 0-9 is provided.

[2]:
```
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train.shape, y_train.shape, x_test.shape, y_test.shape
```

[2]:
```
((60000, 28, 28), (60000,), (10000, 28, 28), (10000,))
```

First, let's visualise some of the training data.

[3]:
```
fig, axes = plt.subplots(9,9,figsize=(9,9))
for i, ax in enumerate(axes.flatten()):
    ax.imshow(x_train[i], cmap='gray')
    ax.set_title(f'Label : {y_train[i]}')
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
fig.tight_layout()
```



In Keras, 2D convolutional layers take 3D tensors with dimensions (height, width, channels) as input. Since our images are grayscale, the input data has only a single channel,. We reshape the training data into 3D tensors, and convert the 8-bit integer values to floats.

[4]:
```
x_train = x_train.reshape((60000, 28, 28, 1))
x_test = x_test.reshape((10000, 28, 28, 1))

x_train, x_test = x_train / 255.0, x_test / 255.0
```

2 Building a CNN using the Keras API

The Sequential model allows us to build up our neural network's architecture by chaining together layers. This is a good place to get started with Keras, and covers the majority of real-world use cases.

[5]:
```
model = Sequential()
```

We first use a series of convolution and pooling layers to extract features from the images. Let's add one of each to the model.

[6]:
```
conv1 = Conv2D(filters=32, kernel_size=[5, 5], padding='same',␣
 ↪activation='relu', input_shape=(28, 28, 1))
pool1 = MaxPooling2D(pool_size=[2, 2])
model.add(conv1)
model.add(pool1)
```

We can inspect the architecture of our model.

[7]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        832
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
=================================================================
Total params: 832
Trainable params: 832
Non-trainable params: 0
_________________________________________________________________
```

The output of every conv2d and max_pooling2d layer is a 3D tensor of shape (height, width, channels). By applying pooling layers between convolutional layers, we can reduce the width and height as we go deeper in the network. Since smaller images are computationally cheaper to process, we can afford to use more output channels in subsequent convolutional layers. Let's add another convolution with more output channels, along with another pooling layer.

[8]:
```
conv2 = Conv2D(filters=64, kernel_size=[5, 5], padding='same',␣
 ↪activation='relu')
pool2 = MaxPooling2D(pool_size=[2, 2])
model.add(conv2)
model.add(pool2)
```

[9]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        832
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
```

```
conv2d_1 (Conv2D)            (None, 14, 14, 64)        51264
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
=================================================================
Total params: 52,096
Trainable params: 52,096
Non-trainable params: 0
_________________________________________________________________
```

To use the output from the convolutional layers to perform the classification, we feed the output into one or more dense layers. We first flatten the output to create a 1D input layer, then add dense layers on top. MNIST has ten classes, the digits 0-9, so we use a final dense layer with ten outputs and a softmax activation to perform the classification.

[10]:
```python
flat = Flatten()
dense1 = Dense(units=64, activation='relu')
dense2 = Dense(units=10, activation='softmax')
model.add(flat)
model.add(dense1)
model.add(dense2)
```

[11]:
```python
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        832
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 64)        51264
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0
_________________________________________________________________
dense (Dense)                (None, 64)                200768
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 253,514
Trainable params: 253,514
Non-trainable params: 0
_________________________________________________________________
```

Finally, we need to specify the training strategy.

[12]:
```python
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
                metrics=['accuracy'])
```

Here's a helper function to create a new instance of the model.

[13]:
```
def get_new_model():
    clear_session() # Free up memory
    model = Sequential()
    model.add(Conv2D(filters=32, kernel_size=[5, 5], padding='same',␣
 ↪activation='relu', input_shape=(28, 28, 1)))
    model.add(MaxPooling2D(pool_size=[2, 2]))
    model.add(Conv2D(filters=64, kernel_size=[5, 5], padding='same',␣
 ↪activation='relu'))
    model.add(MaxPooling2D(pool_size=[2, 2]))
    model.add(Flatten())
    model.add(Dense(units=64, activation='relu'))
    model.add(Dense(units=10, activation='softmax'))
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

Let's look at the final architecture of out model.

[14]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        832
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 64)        51264
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0
_________________________________________________________________
dense (Dense)                (None, 64)                200768
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 253,514
Trainable params: 253,514
Non-trainable params: 0
_________________________________________________________________
```

3 Training and evaluating the model

Finally, we can train our model. For each training epoch, TensorFlow reports training progress and accuracy during the training process.

[15]:
```
model.fit(x_train, y_train, epochs=1)
```

```
60000/60000 [==============================] - 30s 504us/sample - loss: 0.1201 -
accuracy: 0.9625
```

[15]:
```
<tensorflow.python.keras.callbacks.History at 0x7f0434064f60>
```

[16]:
```
model.evaluate(x_test, y_test)
```

```
10000/10000 [==============================] - 1s 135us/sample - loss: 0.0410 -
accuracy: 0.9867
```

[16]:
```
[0.04095309815197252, 0.9867]
```

We can continue training this model for additional epochs if needed.

[17]:
```
model.fit(x_train, y_train, epochs=2)
```

```
Epoch 1/2
60000/60000 [==============================] - 31s 509us/sample - loss: 0.0399 -
accuracy: 0.9875
Epoch 2/2
60000/60000 [==============================] - 31s 523us/sample - loss: 0.0280 -
accuracy: 0.9908
```

[17]:
```
<tensorflow.python.keras.callbacks.History at 0x7f04358e1710>
```

[18]:
```
model.evaluate(x_test, y_test)
```

```
10000/10000 [==============================] - 1s 139us/sample - loss: 0.0438 -
accuracy: 0.9856
```

[18]:
```
[0.04382402242935495, 0.9856]
```

4 Batch training

We can use TensorFlow datasets to shuffle our data and control how it is split into batches for training.

[19]:
```
BATCH_SIZE = 64
AUTOTUNE = experimental.AUTOTUNE
BUFFER_SIZE = len(y_train)

ds = Dataset.from_tensor_slices((x_train, y_train))
ds = ds.shuffle(buffer_size=len(y_train))
ds = ds.repeat()
ds = ds.batch(BATCH_SIZE)
ds = ds.prefetch(buffer_size=AUTOTUNE)
```

We set the buffer size equal to the size of the data set so that the entire data set is fully shuffled. Using a smaller batch size reduces memory requirements but results in worse randomisation. The buffer is filled before any items are taken from it, so a large buffer can cause a delay when beginning the dataset.

[21]:
```
model = get_new_model()
```

Let's see how the model improves as more training data are used. Here we train for 20 epochs. At each step of the epoch, the next batch of training data is used to further train the model. Training set accuracy and training loss is reported at the end of each epoch.

[22]:
```
model.fit(ds, epochs=20, steps_per_epoch=10)
```

```
Epoch 1/20
10/10 [==============================] - 1s 94ms/step - loss: 2.0608 - accuracy:
0.3453
Epoch 2/20
10/10 [==============================] - 0s 34ms/step - loss: 1.0828 - accuracy:
0.7266
Epoch 3/20
10/10 [==============================] - 0s 32ms/step - loss: 0.6933 - accuracy:
0.7734
Epoch 4/20
10/10 [==============================] - 0s 33ms/step - loss: 0.4053 - accuracy:
0.8703
Epoch 5/20
10/10 [==============================] - 0s 32ms/step - loss: 0.3661 - accuracy:
0.8828
Epoch 6/20
10/10 [==============================] - 0s 35ms/step - loss: 0.3572 - accuracy:
0.8938
Epoch 7/20
10/10 [==============================] - 0s 37ms/step - loss: 0.3276 - accuracy:
0.8953
Epoch 8/20
10/10 [==============================] - 0s 33ms/step - loss: 0.2763 - accuracy:
0.9203
Epoch 9/20
10/10 [==============================] - 0s 33ms/step - loss: 0.2268 - accuracy:
0.9328
Epoch 10/20
10/10 [==============================] - 0s 34ms/step - loss: 0.2412 - accuracy:
0.9187
Epoch 11/20
10/10 [==============================] - 0s 33ms/step - loss: 0.1481 - accuracy:
```

```
0.9531
Epoch 12/20
10/10 [==============================] - 0s 32ms/step - loss: 0.1906 - accuracy:
0.9500
Epoch 13/20
10/10 [==============================] - 0s 33ms/step - loss: 0.1465 - accuracy:
0.9578
Epoch 14/20
10/10 [==============================] - 0s 32ms/step - loss: 0.1249 - accuracy:
0.9563
Epoch 15/20
10/10 [==============================] - 0s 32ms/step - loss: 0.1628 - accuracy:
0.9563
Epoch 16/20
10/10 [==============================] - 0s 36ms/step - loss: 0.1373 - accuracy:
0.9500
Epoch 17/20
10/10 [==============================] - 0s 33ms/step - loss: 0.1606 - accuracy:
0.9547
Epoch 18/20
10/10 [==============================] - 0s 32ms/step - loss: 0.1348 - accuracy:
0.9563
Epoch 19/20
10/10 [==============================] - 0s 32ms/step - loss: 0.1243 - accuracy:
0.9609
Epoch 20/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0855 - accuracy:
0.9703
```

[22]:
```
<tensorflow.python.keras.callbacks.History at 0x7f042c150668>
```

Finally, let's see how the model performs on the test set.

[23]:
```python
model.evaluate(x_test, y_test)
```

```
10000/10000 [==============================] - 1s 136us/sample - loss: 0.1178 -
accuracy: 0.9647
```

[23]:
```
[0.11776709721330553, 0.9647]
```

What if we continue training for another 20 epochs?

[24]:
```python
print('Training:')
model.fit(ds, epochs=20, steps_per_epoch=10)
print('Test:')
model.evaluate(x_test, y_test)
```

```
Training:
Epoch 1/20
```

```
10/10 [==============================] - 1s 52ms/step - loss: 0.0926 - accuracy:
0.9641
Epoch 2/20
10/10 [==============================] - 0s 32ms/step - loss: 0.1814 - accuracy:
0.9406
Epoch 3/20
10/10 [==============================] - 0s 32ms/step - loss: 0.1244 - accuracy:
0.9609
Epoch 4/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0945 - accuracy:
0.9672
Epoch 5/20
10/10 [==============================] - 0s 31ms/step - loss: 0.1041 - accuracy:
0.9656
Epoch 6/20
10/10 [==============================] - 0s 31ms/step - loss: 0.1184 - accuracy:
0.9609
Epoch 7/20
10/10 [==============================] - 0s 33ms/step - loss: 0.0956 - accuracy:
0.9641
Epoch 8/20
10/10 [==============================] - 0s 37ms/step - loss: 0.1187 - accuracy:
0.9547
Epoch 9/20
10/10 [==============================] - 0s 32ms/step - loss: 0.1077 - accuracy:
0.9594
Epoch 10/20
10/10 [==============================] - 0s 35ms/step - loss: 0.1174 - accuracy:
0.9641
Epoch 11/20
10/10 [==============================] - 0s 34ms/step - loss: 0.0724 - accuracy:
0.9766
Epoch 12/20
10/10 [==============================] - 0s 32ms/step - loss: 0.1107 - accuracy:
0.9688
Epoch 13/20
10/10 [==============================] - 0s 33ms/step - loss: 0.0829 - accuracy:
0.9781
Epoch 14/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0623 - accuracy:
0.9812
Epoch 15/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0784 - accuracy:
0.9750
Epoch 16/20
10/10 [==============================] - 0s 33ms/step - loss: 0.0642 - accuracy:
0.9797
Epoch 17/20
```

```
10/10 [==============================] - 0s 32ms/step - loss: 0.0955 - accuracy:
0.9641
Epoch 18/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0985 - accuracy:
0.9750
Epoch 19/20
10/10 [==============================] - 0s 36ms/step - loss: 0.0841 - accuracy:
0.9766
Epoch 20/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0617 - accuracy:
0.9812
Test:
10000/10000 [==============================] - 1s 136us/sample - loss: 0.0795 -
accuracy: 0.9750
```

[24]:
```
[0.07947156231738627, 0.975]
```

We can keep training as long as we like. When all training examples have been used, the buffer will be refilled with a random shuffling of the training data and training batches will continue to be taken from the buffer.

[25]:
```python
print('Training:')
model.fit(ds, epochs=20, steps_per_epoch=10)
print('Test:')
model.evaluate(x_test, y_test)
```

```
Training:
Epoch 1/20
10/10 [==============================] - 1s 51ms/step - loss: 0.0707 - accuracy:
0.9781
Epoch 2/20
10/10 [==============================] - 0s 33ms/step - loss: 0.1151 - accuracy:
0.9656
Epoch 3/20
10/10 [==============================] - 0s 34ms/step - loss: 0.0737 - accuracy:
0.9766
Epoch 4/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0700 - accuracy:
0.9766
Epoch 5/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0588 - accuracy:
0.9812
Epoch 6/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0954 - accuracy:
0.9719
Epoch 7/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0616 - accuracy:
0.9828
Epoch 8/20
```

```
10/10 [==============================] - 0s 32ms/step - loss: 0.0661 - accuracy:
0.9812
Epoch 9/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0620 - accuracy:
0.9766
Epoch 10/20
10/10 [==============================] - 0s 34ms/step - loss: 0.0645 - accuracy:
0.9781
Epoch 11/20
10/10 [==============================] - 0s 33ms/step - loss: 0.0441 - accuracy:
0.9891
Epoch 12/20
10/10 [==============================] - 0s 31ms/step - loss: 0.0890 - accuracy:
0.9734
Epoch 13/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0611 - accuracy:
0.9844
Epoch 14/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0380 - accuracy:
0.9859
Epoch 15/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0466 - accuracy:
0.9844
Epoch 16/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0365 - accuracy:
0.9891
Epoch 17/20
10/10 [==============================] - 0s 33ms/step - loss: 0.0763 - accuracy:
0.9750
Epoch 18/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0772 - accuracy:
0.9812
Epoch 19/20
10/10 [==============================] - 0s 34ms/step - loss: 0.0830 - accuracy:
0.9766
Epoch 20/20
10/10 [==============================] - 0s 32ms/step - loss: 0.0546 - accuracy:
0.9750
Test:
10000/10000 [==============================] - 1s 141us/sample - loss: 0.0864 -
accuracy: 0.9727
```

[25]:
```
[0.08643419327465818, 0.9727]
```

[ ]:
```
```