Practical-2

January 24, 2021

1 Keras API for Deep Learning

1.1 Introduction

Keras is an open-source software library that provides a Python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library. Up until version 2.3 Keras supported multiple backends, including *TensorFlow*, *Microsoft Cognitive Toolkit*, *R*, *Theano*, and *PlaidML*.

As of version 2.4, only TensorFlow is supported. Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible. It was developed as part of the research effort of project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System), and its primary author and maintainer is François Chollet, a Google engineer. Chollet also is the author of the XCeption deep neural network model.

1.2 Models in Keras (`keras.models`)

Keras `Model` is an abstraction provided by the Keras API to easily build Deep Learning models in Python. It is the most basic construct that enables the user to quickly prototype DL models. The most basic model that can be made using this construct is shown below. The full explaination of the code presented here is given in the subsequent sections.

```
import tensorflow as tf

inputs = tf.keras.Input(shape=(3,))
x = tf.keras.layers.Dense(4, activation=tf.nn.relu)(inputs)
outputs = tf.keras.layers.Dense(5, activation=tf.nn.softmax)(x)
model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

The above code demonstrates the use of the `Model` class to create a vanilla neural network with two fully connected hidden layers with 4 and 5 neurons respectively.

To start training the model, we first need to call the `.compile` method that builds the model using the optimizer, learning rate, and other arguments provided to the method. An example of such a call is shown below:

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

We can then optionally call the `.summary` method to get the summary of the model.

```
model.summary()
```

To start training the model with some available data, the `.fit` method can be called.

```
model.fit(X_train, Y_train,
          batch_size=32, epochs=10)
```

where `X_train` and `Y_train` are the training data and lables respectively.

Finally, we can evaluate our model on the test data:

```
score = model.evaluate(X_test, Y_test)
```

where `X_test` and `Y_test` are the testing data and lables respectively.

The model can then be used to predict labels of some new data using the `.predict` method.

```
model.predict(X_test, Y_test)
```

1.3 Neuron Layers in Keras (`keras.layers`)

1.3.1 Introduction

Layers are the basic building blocks of neural networks in Keras. A layer consists of a tensor-in tensor-out computation function (the layer's call method) and some state, held in TensorFlow variables (the layer's weights).

A Layer instance is callable, much like a function:

```
>>> from tensorflow.keras import layers
>>>
>>> layer = layers.Dense(32, activation='relu')
>>> inputs = tf.random.uniform(shape=(10, 20))
>>> outputs = layer(inputs)
```

Unlike a function, though, layers maintain a state, updated when the layer receives data during training, and stored in `layer.weights`:

```
>>> layer.weights
[<tf.Variable 'dense/kernel:0' shape=(20, 32) dtype=float32>,
 <tf.Variable 'dense/bias:0' shape=(32,) dtype=float32>]
```

1.3.2 Core Layers

- Input layer
- Dense layer
- Activation layer
- Embedding layer
- Masking layer
- Lambda layer

Input Layer `Input()` is used to instantiate a Keras tensor.

Example:

```
>>> # this is a logistic regression in Keras
>>> x = Input(shape=(32,))
>>> y = Dense(16, activation='softmax')(x)
>>> model = Model(x, y)
```

Dense Layer `Dense()` is a regular densely-connected NN layer..

Example:

```
>>> # Create a `Sequential` model and add a Dense layer as the first layer.
>>> model = tf.keras.models.Sequential()
>>> model.add(tf.keras.Input(shape=(16,)))
>>> model.add(tf.keras.layers.Dense(32, activation='relu'))
>>> # Now the model will take as input arrays of shape (None, 16)
>>> # and output arrays of shape (None, 32).
>>> # Note that after the first layer, you don't need to specify
>>> # the size of the input anymore:
>>> model.add(tf.keras.layers.Dense(32))
>>> model.output_shape
(None, 32)
```

Activation Layer `Activation()` applies an activation function to an output..

Example:

```
>>> layer = tf.keras.layers.Activation('relu')
>>> output = layer([-3.0, -1.0, 0.0, 2.0])
>>> list(output.numpy())
[0.0, 0.0, 0.0, 2.0]
>>> layer = tf.keras.layers.Activation(tf.nn.relu)
>>> output = layer([-3.0, -1.0, 0.0, 2.0])
>>> list(output.numpy())
[0.0, 0.0, 0.0, 2.0]
```

Layer activations:

- `relu` function
  - The *ReLU* activation function.
- `sigmoid` function
  - The *Sigmoid* activation function.
- `softmax` function
  - The *Softmax* activation function.
- `softplus` function
  - The *Softplus* activation function.
- `softsign` function
  - The *Softsign* activation function.
- `tanh` function
  - The *Hyperbolic Tangent* activation function.
- `selu` function
  - The *SeLU* activation function.
- `elu` function
  - The *eLU* activation function.

- `exponential` function
  - The *Exponential* activation function.

Embedding Layer Turns positive integers (indexes) into dense vectors of fixed size.

`e.g. [[4], [20]] -> [[0.25, 0.1], [0.6, -0.2]]`

Note: This layer can only be used as the first layer in a model.

Example:

```
>>> model = tf.keras.Sequential()
>>> model.add(tf.keras.layers.Embedding(1000, 64, input_length=10))
>>> # The model will take as input an integer matrix of size (batch,
>>> # input_length), and the largest integer (i.e. word index) in the input
>>> # should be no larger than 999 (vocabulary size).
>>> # Now model.output_shape is (None, 10, 64), where `None` is the batch
>>> # dimension.
>>> input_array = np.random.randint(1000, size=(32, 10))
>>> model.compile('rmsprop', 'mse')
>>> output_array = model.predict(input_array)
>>> print(output_array.shape)
(32, 10, 64)
```

Masking Layer Masks a sequence by using a mask value to skip timesteps.

For each timestep in the input tensor (dimension #1 in the tensor), if all values in the input tensor at that timestep are equal to mask_value, then the timestep will be masked (skipped) in all downstream layers (as long as they support masking).

If any downstream layer does not support masking yet receives such an input mask, an exception will be raised.

Example:

```
>>> samples, timesteps, features = 32, 10, 8
>>> inputs = np.random.random([samples, timesteps, features]).astype(np.float32)
>>> inputs[:, 3, :] = 0.
>>> inputs[:, 5, :] = 0.
>>>
>>> model = tf.keras.models.Sequential()
>>> model.add(tf.keras.layers.Masking(mask_value=0.,
...                                   input_shape=(timesteps, features)))
>>> model.add(tf.keras.layers.LSTM(32))
>>>
>>> output = model(inputs)
>>> # The time step 3 and 5 will be skipped from LSTM calculation.
```

Lambda Layer Wraps arbitrary expressions as a Layer object.

The Lambda layer exists so that arbitrary TensorFlow functions can be used when constructing Sequential and Functional API models. Lambda layers are best suited for simple operations or quick experimentation.

Example:

```
>>> # add a x -> x^2 layer
>>> model.add(Lambda(lambda x: x ** 2))
```

1.4 Traning an ANN using Keras API!

[1]:
```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.datasets import mnist
from sklearn.preprocessing import OneHotEncoder
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt
```

1.5 Loading the Dataset

[2]:
```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

[3]:
```
print(f"{X_train.shape=}\n{y_train.shape=}\n{X_test.shape=}\n{y_test.shape=}")
```

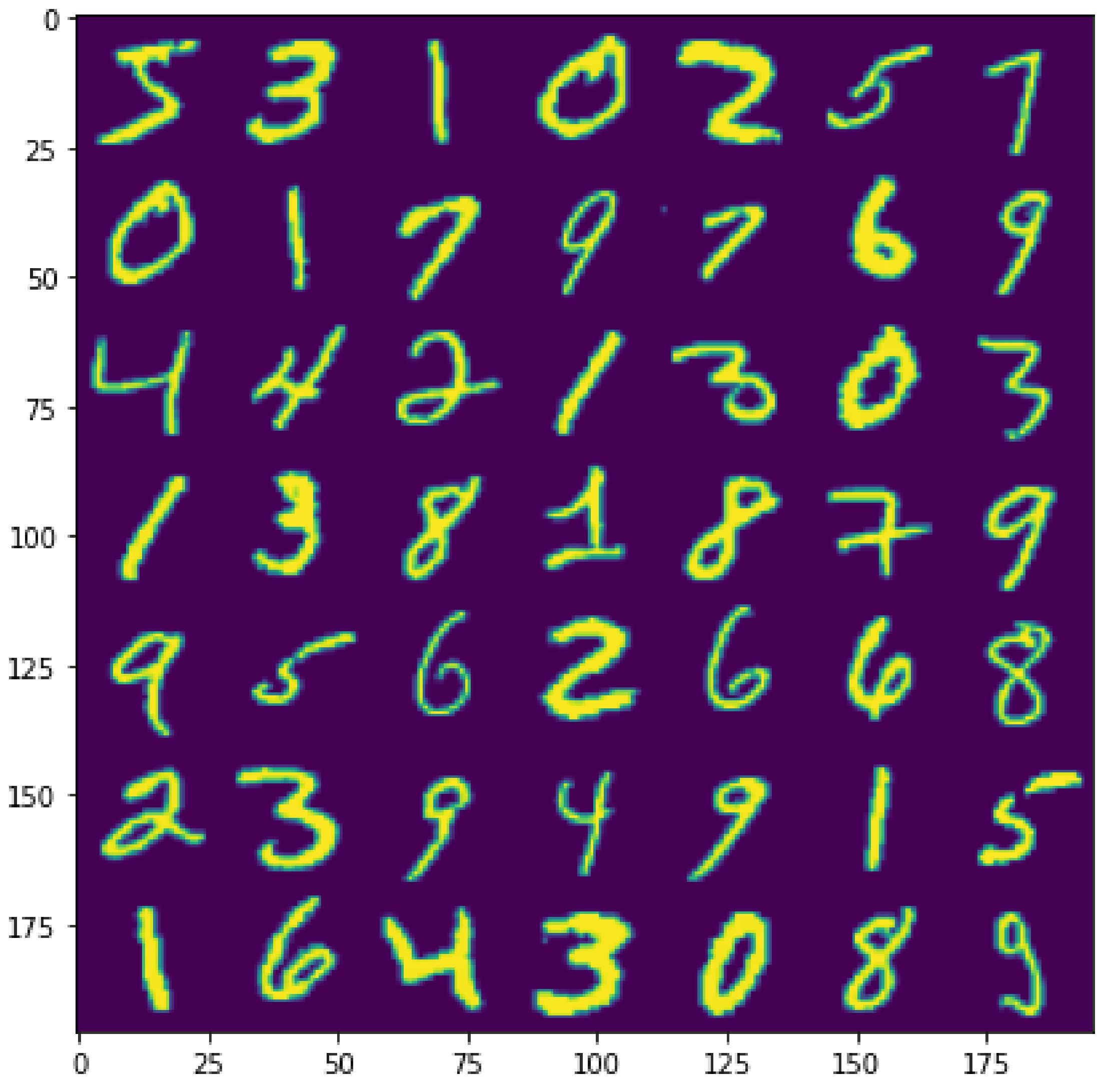
```
X_train.shape=(60000, 28, 28)
y_train.shape=(60000,)
X_test.shape=(10000, 28, 28)
y_test.shape=(10000,)
```

1.6 Processing and Visualizing the Data

[4]:
```
X_train = X_train / 255.
X_test  = X_test  / 255.
X_train = X_train.reshape(60000, 28*28)
X_test = X_test.reshape(10000, 28*28)
encoder = OneHotEncoder()
y_train = encoder.fit_transform(y_train.reshape(-1, 1)).toarray()
y_test = encoder.transform(y_test.reshape(-1, 1)).toarray()
```

[5]:
```
fig, ax = plt.subplots(figsize=(7, 7))
X_to_plot = X_train[:49, :]
X_to_plot = np.c_[X_to_plot[  : 7, ...].reshape(28*7, 28),
                  X_to_plot[ 7:14, ...].reshape(28*7, 28),
                  X_to_plot[14:21, ...].reshape(28*7, 28),
                  X_to_plot[21:28, ...].reshape(28*7, 28),
                  X_to_plot[28:35, ...].reshape(28*7, 28),
                  X_to_plot[35:42, ...].reshape(28*7, 28),
                  X_to_plot[42:49, ...].reshape(28*7, 28)]
ax.imshow(X_to_plot)
```

```
[5]: <matplotlib.image.AxesImage at 0x7f838feeb130>
```



1.7 Creating the Keras Model

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

```
[7]: model.add(Dense(units=16, activation='relu'))
     model.add(Dense(units=16, activation='relu'))
     model.add(Dense(units=10, activation='softmax'))
```

```
[8]: model.compile(loss='categorical_crossentropy',
                   optimizer='adam',
                   metrics=['accuracy'])
```

1.8 Training the Model

[9]:
```
history = model.fit(X_train, y_train,
                    validation_data=(X_test, y_test),
                    epochs=5, batch_size=32)
```

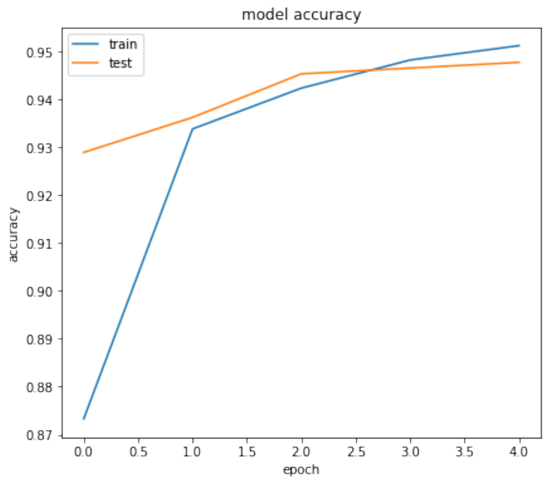
```
Epoch 1/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.7522 -
accuracy: 0.7738 - val_loss: 0.2441 - val_accuracy: 0.9289
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2389 -
accuracy: 0.9316 - val_loss: 0.2199 - val_accuracy: 0.9362
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2017 -
accuracy: 0.9404 - val_loss: 0.1954 - val_accuracy: 0.9453
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.1765 -
accuracy: 0.9481 - val_loss: 0.1884 - val_accuracy: 0.9465
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.1642 -
accuracy: 0.9519 - val_loss: 0.1767 - val_accuracy: 0.9477
```

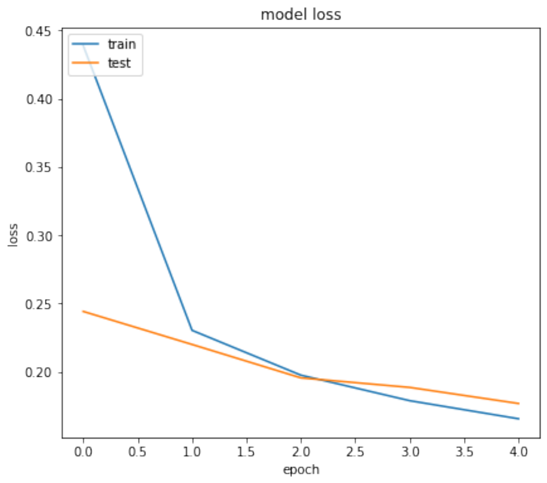
1.9 Evaluating on the Test Set

[10]:
```
loss_and_metrics = model.evaluate(X_test, y_test, batch_size=128)
```

```
79/79 [==============================] - 0s 2ms/step - loss: 0.1767 - accuracy:
0.9477
```

[11]:
```
# summarize history for accuracy
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(15, 6))
ax[0].plot(history.history['accuracy']);
ax[0].plot(history.history['val_accuracy']);
ax[0].set_title('model accuracy');
ax[0].set_ylabel('accuracy');
ax[0].set_xlabel('epoch');
ax[0].legend(['train', 'test'], loc='upper left');
# summarize history for loss
ax[1].plot(history.history['loss']);
ax[1].plot(history.history['val_loss']);
ax[1].set_title('model loss');
ax[1].set_ylabel('loss');
ax[1].set_xlabel('epoch');
ax[1].legend(['train', 'test'], loc='upper left');
```


model accuracy
train
test
accuracy
epoch


model loss
train
test
loss
epoch

```
[ ]:
```