Keras：High-level neural networks API

20210420

Introduction

- We're going to show you how to build and train a multilayer convolutional neural network in even fewer lines of code.
- We'll be introducing an interface that sites on top of TensorFlow, and allows you to draw on the power of TensorFlow with far more concise code.
- You'll be building a deep neural network using a new set of tools.
- You'll still have TensorFlow under the hood, but with an interface that makes testing and prototyping much faster.

Keras Overview

https://keras.io/

The Python Deep Learning library

- Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.

Use Keras if you need a deep learning library that:

- Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).
- Supports both convolutional networks and recurrent networks, as well as combinations of the two.
- Runs seamlessly on CPU and GPU.

Keras/TensorFlow are compatible with:

- Python 3.5–3.8
- Ubuntu 16.04 or later
- Windows 7 or later
- macOS 10.12.6 (Sierra) or later.

Multi-backend Keras and tf.keras

- At this time, we recommend that Keras users who use multi-backend Keras with the TensorFlow backend switch to tf.keras in TensorFlow 2.0. tf.keras is better maintained and has better integration with TensorFlow features (eager execution, distribution support and other).
- Keras 2.2.5 was the last release of Keras implementing the 2.2.* API. It was the last release to only support TensorFlow 1 (as well as Theano and CNTK).
- The current release is Keras 2.3.0, which makes significant API changes and add support for TensorFlow 2.0. The 2.3.0 release will be the last major release of multi-backend Keras. Multi-backend Keras is superseded by tf.keras
- Bugs present in multi-backend Keras will only be fixed until April 2020 (as part of minor releases).

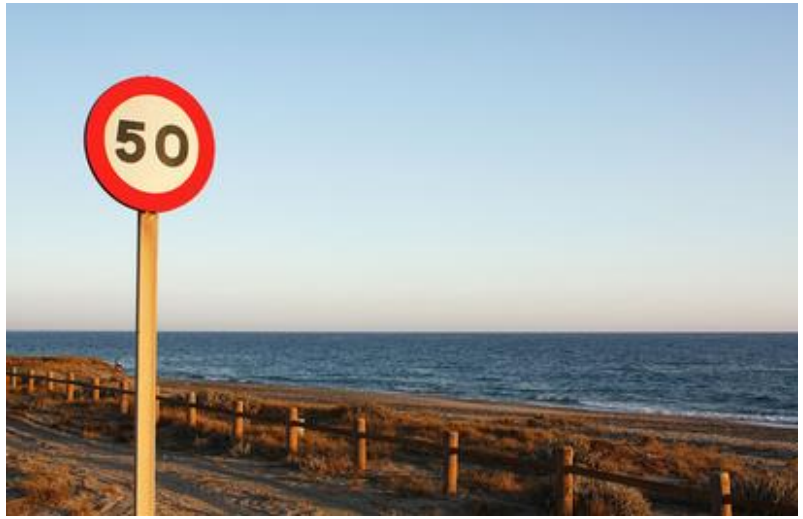
Keras backends

What is a "backend"?

- Keras is a model-level library, providing high-level building blocks for developing deep learning models. It does not handle low-level operations such as tensor products, convolutions and so on itself. Instead, it relies on a specialized, well optimized tensor manipulation library to do so, serving as the "backend engine" of Keras. Rather than picking one single tensor library and making the implementation of Keras tied to that library, Keras handles the problem in a modular way, and several different backend engines can be plugged seamlessly into Keras.
- At this time, Keras has three backend implementations available: the **TensorFlow** backend, the **Theano** backend, and the **CNTK** backend.

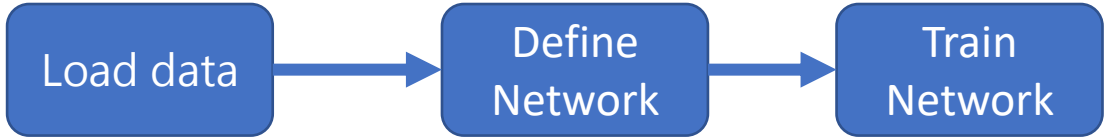
TensorFlow is an open-source symbolic tensor manipulation framework developed by Google.
Theano is an open-source symbolic tensor manipulation framework developed by LISA Lab at Université de Montréal.
CNTK is an open-source toolkit for deep learning developed by Microsoft.
In the future, we are likely to add more backend options.

Keras Overview



Keras(版本2.1.5) makes coding deep neural networks simpler.

- To demonstrate just how easy it is, you‘re going to build a simple fully-connected network in a few dozen lines of code.
- We’ll be connecting the concepts that you’ve learned in the previous lessons to the methods that Keras provides.
- The network you will build is similar to Keras’s sample network that builds out a convolutional neural network for MNIST. However for the network you will build you're going to use a small subset of the German Traffic Sign Recognition Benchmark.
- The general idea for this example is that you'll first load the data, then define the network, and then finally train the network.




```
anaconda search -t conda keras
>>conda-forge/keras        |    2.0.6 | conda          | linux-64, win-32, win-64, osx-64
                                : Deep Learning for Python
>>jaikumarm/keras          |    2.0.8 | conda          | linux-64, win-32, osx-64, linux-32, win-64
                                : Deep Learning for Python
```

```
conda install -c conda-forge keras=2.1.5
conda install -c jaikumarm keras
```

Install KERAS library in Anaconda

1- conda create --name dpKeras
2- activate dpKeras
3- conda install ipython
4- conda install jupyter
5- pip install tensorflow

```
print (sys.version)
print (sys.version_info)
```

```
3.9.2 | packaged by conda-forge | (default, Feb 21 2021, 04:59:43) [MSC v.1916 64 bit (AMD64)]
sys.version_info(major=3, minor=9, micro=2, releaselevel='final', serial=0)
```

```
(dpkeras) C:\Users\fu>pip install tensorflow
Collecting tensorflow
  Downloading tensorflow-2.5.0rc1-cp39-cp39-win_amd64.whl (422.5 MB)
     |                                                  422.5 MB 58 kB/s
```

CNN MINIST Keras samples: https://keras.io/examples/vision/mnist_convnet/
https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py

Neural Networks in Keras

Neural Networks in Keras

Here are some core concepts you need to know for working with Keras.

Sequential Model

```
from keras.models import Sequential
#Create the Sequential model
model = Sequential()
```

The keras.models.Sequential class is a wrapper for the neural network model. It provides common functions like fit(), evaluate(), and compile(). We'll cover these functions as we get to them. Let's start looking at the layers of the model.

Layers

A Keras layer is just like a neural network layer. There are fully connected layers, max pool layers, and activation layers. You can add a layer to the model using the model's add() function. For example, a simple model would look like this:

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Flatten
#Create the Sequential model
model = Sequential()
#1st Layer - Add a flatten layer
model.add(Flatten(input_shape=(32, 32, 3)))
#2nd Layer - Add a fully connected layer
model.add(Dense(100))
#3rd Layer - Add a ReLU activation layer
model.add(Activation('relu'))
#4th Layer - Add a fully connected layer
model.add(Dense(60))
#5th Layer - Add a ReLU activation layer
model.add(Activation('relu'))
```

Keras will automatically infer the shape of all layers after the first layer. This means you only have to set the input dimensions for the first layer.

The first layer from above, model.add(Flatten(input_shape=(32, 32, 3))), sets the input dimension to (32, 32, 3) and output dimension to (3072=32 x 32 x 3). The second layer takes in the output of the first layer and sets the output dimensions to (100). This chain of passing output to the next layer continues until the last layer, which is the output of the model.

Getting started: 30 seconds to Keras

The core data structure of Keras is a model, a way to organize layers. The simplest type of model is the Sequential model, a linear stack of layers. For more complex architectures, you should use the Keras functional API, which allows to build arbitrary graphs of layers.

Here is the Sequential model:

```
from keras.models import Sequential
model = Sequential()
```

Stacking layers is as easy as .add():

```
from keras.layers import Dense
model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))
```

Once your model looks good, configure its learning process with .compile():

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

If you need to, you can further configure your optimizer. A core principle of Keras is to make things reasonably simple, while allowing the user to be fully in control when they need to (the ultimate control being the easy extensibility of the source code).

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.SGD(lr=0.01, momentum=0.9, nesterov=True))
```

You can now iterate on your training data in batches:

```
# x_train and y_train are Numpy arrays --just like in the Scikit-Learn API.
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Alternatively, you can feed batches to your model manually:

```
model.train_on_batch(x_batch, y_batch)
```

Evaluate your performance in one line:

```
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
```

Or generate predictions on new data:

```
classes = model.predict(x_test, batch_size=128)
```

Neural Networks in Keras

Quiz

In this quiz you will build a multi-layer feedforward neural network to classify traffic sign images using Keras.

1.Set the first layer to a Flatten() layer with the input_shape set to (32, 32, 3).

2.Set the second layer to a Dense() layer with an output width of 128.

3.Use a ReLU activation function after the second layer.

4.Set the output layer width to 5, because for this data set there are only 5 classes.

5.Use a softmax activation function after the output layer.

6.Train the model for 3 epochs. You should be able to get over 50% training accuracy.

To get started, review the Keras documentation about models and layers. The Keras example of a Multi-Layer Perceptron network is similar to what you need to do here. Use that as a guide, but keep in mind that there are a number of differences.

Data Download

The data set used in these quizzes can be downloaded here.

01network_py.txt 01network_solution_py.txt

```
import pickle
import numpy as np
import tensorflow as tf

with open('small_train_traffic.p', mode='rb') as f:
    data = pickle.load(f)
```

```
X_train, y_train = data['features'], data['labels']

# Initial Setup for Keras
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Flatten
# Build the Fully Connected Neural Network in Keras Here
model = Sequential()
model.add(Flatten(input_shape=(32, 32, 3)))
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dense(5))
model.add(Activation('softmax'))

# preprocess data
X_normalized = np.array(X_train / 255.0 - 0.5 )

from sklearn.preprocessing import LabelBinarizer
label_binarizer = LabelBinarizer()
y_one_hot = label_binarizer.fit_transform(y_train)

model.compile('adam', 'categorical_crossentropy', ['accuracy'])
history = model.fit(X_normalized, y_one_hot, epochs=5,
validation_split=0.2)
```

Convolutions in Keras

Convolutions

Build from the previous network.

Add a convolutional layer with 32 filters, a 3x3 kernel, and valid padding before the flatten layer.

Add a ReLU activation after the convolutional layer.

Train for 5 epochs again, should be able to get over 50% accuracy.

Hint: The Keras example of a convolutional neural network for MNIST would be a good example to review.

02network_py.txt 02network_solution_py.txt

keras2.0 model.add(Conv2D(32, (3, 3), input_shape=(32, 32, 3)))

```
model = Sequential()
model.add(Conv2D(32, 3, 3, input_shape=(32, 32, 3)))
model.add(Activation('relu'))
model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dense(5))
model.add(Activation('softmax'))
```

```
Epoch 1/5
3/3 [==============================] - 0s 121ms/step - loss: 1.2890 - accuracy: 0.4375 - val_loss: 0.8306 - val_accuracy: 0.6000
Epoch 2/5
3/3 [==============================] - 0s 19ms/step - loss: 0.8937 - accuracy: 0.5375 - val_loss: 0.5491 - val_accuracy: 0.8500
Epoch 3/5
3/3 [==============================] - 0s 20ms/step - loss: 0.6506 - accuracy: 0.7625 - val_loss: 0.4800 - val_accuracy: 0.8500
Epoch 4/5
3/3 [==============================] - 0s 19ms/step - loss: 0.5259 - accuracy: 0.8500 - val_loss: 0.3521 - val_accuracy: 0.8500
Epoch 5/5
3/3 [==============================] - 0s 19ms/step - loss: 0.4254 - accuracy: 0.8000 - val_loss: 0.2997 - val_accuracy: 0.8500
```

Pooling in Keras

Pooling

Build from the previous network

Add a 2x2 max pooling layer immediately following your convolutional layer.

Train for 5 epochs again. You should be able to get over 50% training accuracy.

03network_py.txt 03network_solution_py.txt

model.add(MaxPooling2D((2,2), strides=(2,2)))

#MaxPooling2D((2, 2))

```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(32, 32, 3)))
model.add(MaxPooling2D((2, 2), strides=(2,2)))
model.add(Activation('relu'))
model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dense(5))
model.add(Activation('softmax'))
```

```
Epoch 1/5
3/3 [==============================] - 0s 136ms/step - loss: 1.5087 - accuracy: 0.3625 - val_loss: 0.9808 - val_accuracy: 0.7000
Epoch 2/5
3/3 [==============================] - 0s 24ms/step - loss: 1.1155 - accuracy: 0.6875 - val_loss: 0.6851 - val_accuracy: 0.7000
Epoch 3/5
3/3 [==============================] - 0s 24ms/step - loss: 0.5871 - accuracy: 0.7625 - val_loss: 0.3442 - val_accuracy: 0.8500
Epoch 4/5
3/3 [==============================] - 0s 24ms/step - loss: 0.4899 - accuracy: 0.8000 - val_loss: 0.2590 - val_accuracy: 0.8500
Epoch 5/5
3/3 [==============================] - 0s 25ms/step - loss: 0.3275 - accuracy: 0.8625 - val_loss: 0.2206 - val_accuracy: 0.8500
```

Dropout in Keras

Dropout

Build from the previous network.

Add a dropout layer after the pooling layer. Set the dropout rate to 50%.

04network_py.txt 04network_solution_py.txt

```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(32, 32, 3)))
model.add(MaxPooling2D((2, 2), strides=(2,2)))
model.add(Dropout(0.5))
model.add(Activation('relu'))
model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dense(5))
model.add(Activation('softmax'))
```

```
Epoch 1/5
3/3 [==============================] - 0s 65ms/step - loss: 1.3015 - accuracy: 0.3250 - val_loss: 0.7751 - val_accuracy: 0.5500
Epoch 2/5
3/3 [==============================] - 0s 20ms/step - loss: 0.7955 - accuracy: 0.6000 - val_loss: 0.5177 - val_accuracy: 0.8500
Epoch 3/5
3/3 [==============================] - 0s 22ms/step - loss: 0.5847 - accuracy: 0.8375 - val_loss: 0.4322 - val_accuracy: 0.8500
Epoch 4/5
3/3 [==============================] - 0s 19ms/step - loss: 0.4872 - accuracy: 0.8000 - val_loss: 0.2856 - val_accuracy: 0.8500
Epoch 5/5
3/3 [==============================] - 0s 28ms/step - loss: 0.3537 - accuracy: 0.8125 - val_loss: 0.1991 - val_accuracy: 1.0000
```

Testing in Keras

Test

Once you've picked out your best model, it's time to test it!

1. Try to get the highest validation accuracy possible. Feel free to use all the previous concepts and train for as many epochs as needed.
2. Select your best model and train it one more time.
3. Use the test data and the Keras evaluate() method to see how well the model does.

05network_py.txt 05network_solution_py.txt

```
model.compile('adam', 'categorical_crossentropy', ['accuracy'])
history = model.fit(X_normalized, y_one_hot, epochs=10, validation_split=0.2)
```

```
with open('small_test_traffic.p', 'rb') as f:
    data_test = pickle.load(f)

X_test = data_test['features']
y_test = data_test['labels']

# preprocess data
X_normalized_test = np.array(X_test / 255.0 - 0.5 )
y_one_hot_test = label_binarizer.fit_transform(y_test)

print("Testing")

metrics = model.evaluate(X_normalized_test, y_one_hot_test)
for metric_i in range(len(model.metrics_names)):
    metric_name = model.metrics_names[metric_i]
    metric_value = metrics[metric_i]
    print('{}: {}'.format(metric_name, metric_value))
```

```
Testing
1/1 [==============================] - 0s 2ms/step - loss: 0.2189 - accuracy: 1.0000
loss: 0.21886424720287323
accuracy: 1.0
```