Brief Introduction to Keras

CS-F441

Dr. Kamlesh Tiwari

Introduction: Keras

- High-level neural networks API, written in Python.
- Capable of running on top of TensorFlow, CNTK, or Theano.
- Everyone has been interacting with Keras as it is in use at Netflix, Uber, Yelp, Instacart, Zocdoc, Square, and many others.

Advantages:

  - It offers consistent & simple APIs i.e. it minimizes the number of user actions required for common use cases.
  - It also provides clear and actionable feedback upon user error.
  - This ease of use does not come at the cost of reduced flexibility, because Keras integrates with lower-level deep learning languages (in particular TensorFlow).
  - Runs seamlessly on CPUs and GPUs.

Installation

1. Install Engine
    a. Keras backend engines: TensorFlow, Theano, or CNTK.
    b. It is recommended to install t**ensorflow** because it can be deployed in production via Tensorflow Serving.
    (See the instructions from https://www.tensorflow.org/install/)
2. To install keras on Linux/Mac use,
    a. `$ sudo pip install keras`

Source: https://keras.io/

Keras Models

1. The core data structure of Keras is a **model**, a way to organize layers.
2. The simplest type of model is the `Sequential` model, a linear stack of layers.
3. For more complex architectures, `Keras functional API` can be used that allows to build arbitrary graphs of layers.

The **sequential** API allows you to create models layer-by-layer for most problems. It is limited in that it **does not allow** you to create models that share layers or have multiple inputs or outputs.

Alternatively, the **functional** API allows you to create models that have a lot more flexibility as you can easily define models where layers connect to more than just the previous and next layers. In fact, you can connect layers to any other layer. As a result, creating complex networks such as siamese networks and residual networks become possible.

Keras Sequential Model

- The `Sequential` model is a linear stack of layers.
- A `Sequential` model can be created by passing a list of layer instances to the constructor:

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

- The layers can be stacked using the .add():

```
model = Sequential()
model.add(Dense(32, input_shape=(784,)))
model.add(Activation('relu'))
```

1. Specifying the input shape

The first layer in a `Sequential` model expects information regarding the shape of the input.

This can be done by passing a value to the `input_shape` argument to the first layer. Batch size is not included to this argument.

```
model = Sequential()
model.add(Dense(32, input_shape=(784,)))
```

Note: The model will take as input arrays of shape (, 784) and output arrays of shape (*, 32).*

2. Compilation

The learning process is configured via the `compile` method.

```
compile(optimizer, loss=None, metrics=None, loss_weights=None, sample_weight_mode=None, weighted_metrics=None,
target_tensors=None)
```

Three arguments are compulsory:

- Optimizer: This can be a string identifier of an existing optimizer (such as `SGD`, `adam`, `rmsprop` etc.), or an instance of the `Optimizer` class. See https://keras.io/optimizers
- Loss function: This is the objective function that the model aims to minimize. It is a string identifier of an existing loss function such as `categorical_crossentropy`, `binary_crossentropy`, `mean_squared_error` etc. (see https://keras.io/losses)
- Metrics. A metric is a function that is used to judge the performance of your model. A metric function is similar to a loss function, except that the results from evaluating a metric are not used when training the model. See https://keras.io/metrics

```
# For a binary classification problem

model.compile(optimizer='rmsprop',loss='binary_crossentropy', metrics=['accuracy'])
```

3. Training

Keras models are trained on Numpy arrays of input data and labels. For training a model, the `fit` function is used.

```
fit(x=None, y=None, batch size=None, epochs=1, verbose=1, callbacks=None, validation split=0.0,
validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None)
```

- **x** is the input data. A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
- **y** is the target data. A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
- **batch_size**: Integer or `None`. Number of samples per gradient update. If unspecified, `batch_size` will default to 32.
- **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire `x` and `y` data provided.
- **verbose**: Integer. 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.
- **Callbacks:** A callback is a set of functions that can be used to get a view on internal states and statistics of the model during training such as `History()` or `ModelCheckpoint()` etc.
- **validation_split**: Float between 0 and 1. Fraction of the training data to be used as validation data.
- **shuffle**: Boolean (**whether** to shuffle the training data before each epoch).

```
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(10, size=(1000, 1))

# Convert labels to categorical one-hot encoding
one_hot_labels = keras.utils.to_categorical(labels, num_classes=10)

# Train the model, iterating on the data in batches of 32 samples
model.fit(data, one_hot_labels, epochs=10, batch_size=32)
```

4. Predict

Generates output predictions for the input samples. Computation is done in batches.

```
predict(x, batch_size=None, verbose=0, steps=None, callbacks=None)
```

5. Evaluate

Returns the loss value & metrics values for the model in test mode.

```
evaluate(x=None, y=None, batch_size=None, verbose=1, sample_weight=None, steps=None, callbacks=None)
```

- **batch_size**: Integer or None. Number of samples per gradient update. If unspecified, batch_size will default to 32.
- **verbose**: Verbosity mode, 0 - silent or 1- progress bar.
- **steps**: Total number of steps (batches of samples) before declaring the prediction/evaluation round finished. Ignored with the default value of None.

Keras functional API

- The Keras functional API is the way to go for defining complex models, such as multi-output models, directed acyclic graphs, or models with shared layers.
- The following example includes all layers required in the computation of `b` given `a`.

```
from keras.models import Model
from keras.layers import Input, Dense

a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(inputs=a, outputs=b)
```

- In the case of multi-input or multi-output models, you can use lists as well:

```
model = Model(inputs=[a1, a2], outputs=[b1, b2, b3])
```

- It also has compile(), fit(), predict() and evaluate() as the Sequential Model.

Keras Layers

1. **Conv2D**

   This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs.

```
keras.layers.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid')
```

2. **Conv2DTranspose**

   This refers to the transposed convolution layer (sometimes called Deconvolution). The need for transposed convolutions generally arises from the desire to use a transformation going in the opposite direction of a normal convolution.

```
keras.layers.Conv2DTranspose(filters, kernel_size, strides=(1, 1), padding='valid')
```

3. MaxPooling2D

This layer performs Max pooling operation for spatial data.

```
keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid')
```

4. Dense

`Dense` implements the operation: `output = activation(dot(input, kernel) + bias)` where `activation` is the element-wise activation function passed as the `activation` argument, `kernel` is a weights matrix created by the layer, and `bias` is a bias vector created by the layer (only applicable if `use_bias` is `True`).

```
model.add(Dense(32, input_shape=(16,)))
```

5. Dropout

Dropout consists in randomly setting a fraction `rate` of input units to 0 at each update during training time, which helps prevent overfitting.

```
keras.layers.Dropout(rate, noise_shape=None, seed=None)
```

6. Flatten

Flattens the input. Does not affect the batch size.

Here, **data_format** can have value either `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. The purpose of this argument is to preserve weight ordering when switching a model from one data format to another.

```
keras.layers.Flatten(data_format=None)
```

7. Batch Normalization

Normalize the activations of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.

Here, axis refers to the axis that should be normalized.

```
keras.layers.BatchNormalization(axis=-1)
```

Coding Example

The Problem: Fashion MNIST classification

Classify the Fashion-MNIST dataset using a Convolutional Neural Network (CNN) architecture.

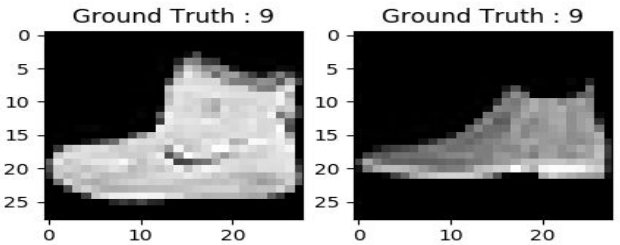
Fashion-MNIST database (https://github.com/zalandoresearch/fashion-mnist) is similar to MNIST dataset (having 10 categories of handwritten digits). It shares the same image size (28x28) and structure of training (60,000) and testing (10,000) splits. Categories are (0) T-shirt/top, (1) Trouser, (2) Pullover, (3) Dress, (4) Coat, (5) Sandal, (6) Shirt, (7) Sneaker, (8) Bag, (9) Ankle boot

1. Importing Libraries

```
import numpy as np
from keras.utils import to_categorical
import matplotlib.pyplot as plt
#matplotlib inline
from keras.datasets import fashion_mnist
from sklearn.model_selection import train_test_split
import keras

from keras.models import Sequential,Input,Model
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.layers.normalization import BatchNormalization
from keras.layers.advanced_activations import LeakyReLU
```

2. Load the Data

```
(train_X,train_Y), (test_X,test_Y) = fashion_mnist.load_data()

print('Training data shape : ', train_X.shape, train_Y.shape)

print('Testing data shape : ', test_X.shape, test_Y.shape)

# Find the unique numbers from the train labels
classes = np.unique(train_Y)
nClasses = len(classes)
plt.figure(figsize=[5,5])

# Display the first image in training data
plt.subplot(121)
plt.imshow(train_X[0,:,:], cmap='gray')
plt.title("Ground Truth : {}".format(train_Y[0]))

# Display the first image in testing data
plt.subplot(122)
plt.imshow(test_X[0,:,:], cmap='gray')
plt.title("Ground Truth : {}".format(test_Y[0]))
plt.show()
```



3. Data Preprocessing

```
train_X = train_X.reshape(-1, 28,28, 1)
test_X = test_X.reshape(-1, 28,28, 1)
train_X.shape, test_X.shape
train_X = train_X.astype('float32')
test_X = test_X.astype('float32')
train_X = train_X / 255.
test_X = test_X / 255.

# Change the labels from categorical to one-hot encoding
train_Y_one_hot = to_categorical(train_Y)
test_Y_one_hot = to_categorical(test_Y)

# Display the change for category label using one-hot encoding
print('Original label:', train_Y[0])
print('After conversion to one-hot:', train_Y_one_hot[0])


#Splitting the dataset in training (80%) and testing (20%)
train_X,valid_X,train_label,valid_label = train_test_split(train_X, train_Y_one_hot,
test_size=0.2, random_state=13)

#Checking the size again
train_X.shape,valid_X.shape,train_label.shape,valid_label.shape
```

4. Model

```
batch_size = 64
epochs = 20
num_classes = 10

#Architecture
fashion_model = Sequential()
fashion_model.add(Conv2D(32, kernel_size=(3,3),activation='linear',input_shape=(28,28,1),padding='same'))
fashion_model.add(LeakyReLU(alpha=0.1))
fashion_model.add(MaxPooling2D((2, 2),padding='same'))
fashion_model.add(Conv2D(64, (3, 3), activation='linear',padding='same'))
fashion_model.add(LeakyReLU(alpha=0.1))
fashion_model.add(MaxPooling2D(pool_size=(2, 2),padding='same'))
fashion_model.add(Conv2D(128, (3, 3), activation='linear',padding='same'))
fashion_model.add(LeakyReLU(alpha=0.1))
fashion_model.add(MaxPooling2D(pool_size=(2, 2),padding='same'))
fashion_model.add(Flatten())
fashion_model.add(Dense(128, activation='linear'))
fashion_model.add(LeakyReLU(alpha=0.1))
fashion_model.add(Dense(num_classes, activation='softmax'))

#Compilation of model
fashion_model.compile(loss=keras.losses.categorical_crossentropy,
optimizer=keras.optimizers.Adam(),metrics=['accuracy'])
fashion_model.summary()

#Train the model
fashion_train = fashion_model.fit(train_X, train_label,
batch_size=batch_size,epochs=epochs,verbose=1,validation_data=(valid_X, valid_label))
```

5. Model Evaluation

```
test_eval = fashion_model.evaluate(test_X, test_Y_one_hot, verbose=0)
print('Test loss:', test_eval[0])
print('Test accuracy:', test_eval[1])



accuracy = fashion_train.history['acc']
val_accuracy = fashion_train.history['val_acc']
loss = fashion_train.history['loss']
val_loss = fashion_train.history['val_loss']
epochs = range(len(accuracy))
plt.plot(epochs, accuracy, 'bo', label='Training accuracy')
plt.plot(epochs, val_accuracy, 'b', label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

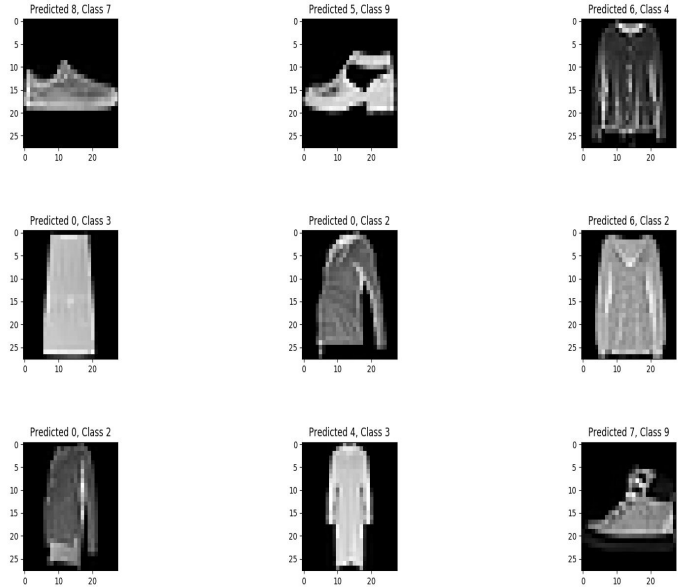
5. Predict Labels

```
predicted_classes = fashion_model.predict(test_X)
predicted_classes = np.argmax(np.round(predicted_classes),axis=1)
predicted_classes.shape, test_Y.shape
correct = np.where(predicted_classes==test_Y)[0]
print ('Found %d correct labels',len(correct))
for i, correct in enumerate(correct[:9]):
    plt.subplot(3,3,i+1)
    plt.imshow(test_X[correct].reshape(28,28), cmap='gray', interpolation='none')
    plt.title("Predicted {}, Class {}".format(predicted_classes[correct], test_Y[correct]))
    plt.tight_layout()
plt.show()

incorrect = np.where(predicted_classes!=test_Y)[0]
print ("Found %d incorrect labels",len(incorrect))
for i, incorrect in enumerate(incorrect[:9]):
    plt.subplot(3,3,i+1)
    plt.imshow(test_X[incorrect].reshape(28,28), cmap='gray', interpolation='none')
    plt.title("Predicted {}, Class {}".format(predicted_classes[incorrect], test_Y[incorrect]))
    plt.tight_layout()
plt.show()
```

Prediction

Correct

Incorrect



6. Generating Classification Report

```
from sklearn.metrics import classification_report
target_names = ["Class {}".format(i) for i in
range(num_classes)]
print(classification_report(test_Y, predicted_classes,
target_names=target_names))
```

```
Test accuracy: 0.8976
Found %d correct labels 8900
Found %d incorrect labels 1100
              precision    recall  f1-score

     Class 0       0.72      0.84      0.78
     Class 1       0.99      0.98      0.98
     Class 2       0.93      0.72      0.81
     Class 3       0.93      0.88      0.90
     Class 4       0.75      0.91      0.82
     Class 5       0.99      0.98      0.98
     Class 6       0.75      0.68      0.72
     Class 7       0.94      0.98      0.96
     Class 8       0.98      0.98      0.98
     Class 9       0.98      0.95      0.97
```

Precision: What proportion of positive identifications was actually correct?
Recall: What proportion of actual positives was identified correctly?

Homework

1. Code the model discussed the class and get the results for different architectural settings
   - Change layers
   - Add dropout
   - Change optimizers and parameters
2. Apply same on MNIST database

Report what changes you observe in accuracy. What is the best accuracy you have achieved?

Thank You !