Xin Zhong

Keras and Tensorflow

Keras is a high level API built on tensorflow (the backend can also be Theano and CNTK). Compare to tf, Keras is much easier to use however with less control of the modelings. Start from the installation, we can simply pip to get Keras in Pycharm at Anaconda environment.

Advantages of Keras

(1) Keras is excellent in rapid prototyping. It is fast to test a design of neural network framework. Even complex neural networks can be built within a few minutes. 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. Graphical model can also be implemented.

```
from keras.models import Sequential
from keras.layers import Dense, Activation
model = Sequential()
model.add(Dense(units=32, activation='relu', input_dim=100))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',loss='binary_crossentropy',
              metrics=['accuracy'])
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(2, size=(1000, 1))
model.fit(data, labels, epochs=10, batch_size=32)
```

Output:

Using TensorFlow backend.

…

Epoch 10/10

32/1000 [.....] - ETA: 0s - loss: 0.6522 - acc: 0.5625

160/1000 [===>.....] - ETA: 0s - loss: 0.6611 - acc: 0.5687

288/1000 [=======>.....] - ETA: 0s - loss: 0.6636 - acc: 0.6007

448/1000 [==========>.....] - ETA: 0s - loss: 0.6644 - acc: 0.6049

640/1000 [==========>.....] - ETA: 0s - loss: 0.6614 - acc: 0.6109

832/1000 [==========>.....] - ETA: 0s - loss: 0.6655 - acc: 0.6130

1000/1000 [==========] - 0s 384us/step - loss: 0.6669 - acc: 0.6090

Looks pretty easy, right?

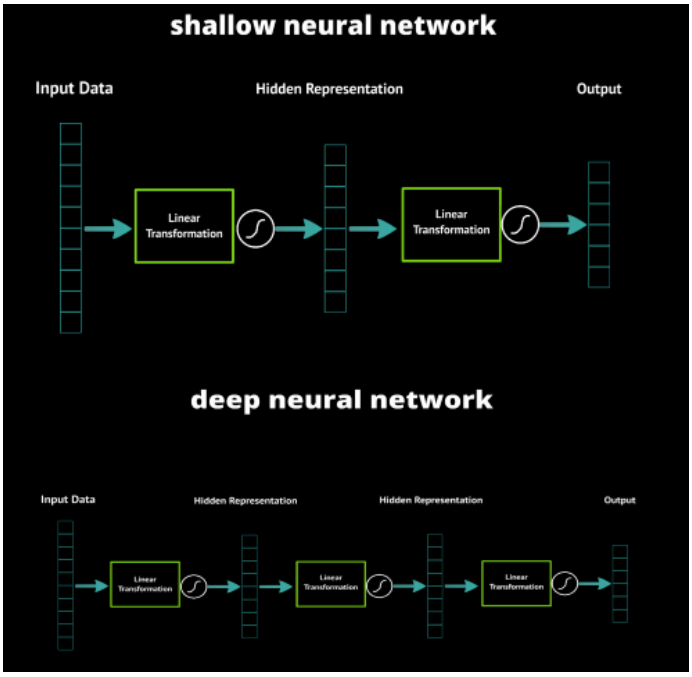
(2) Keras is more pythonic. Modular programming is convenient in Keras. Everything in Keras can be represented as modules which can further be combined as per the user's requirements.
(3) Codes will automatically run on GPU if any available GPU is detected.

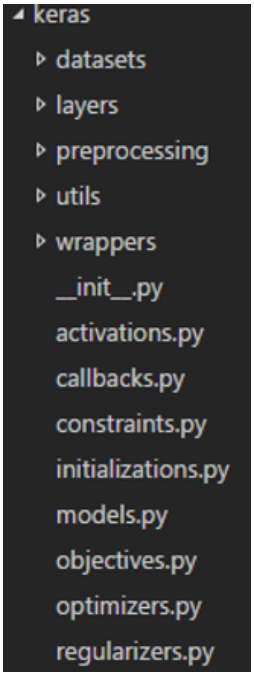
Disadvantages of Keras
(1) In terms of the customization of neural network components, such as layers and cost functions, Keras is less flexible than tf. When it comes to developing some self-defined kind of deep learning models, tf offers more advanced operations.
(2) tf provides more control over neural networks. Operations on weights or gradients can be done in tf. For example, a variable can be defined as trainable.

Keras uses epoch and tf uses iterations.
one epoch is one forward pass and one backward pass of **all** the training samples.
one iteration is a number of passes, each pass using a batch size number of samples. one pass = one forward pass + one backward pass. Where batch size is the number of training samples in one forward/backward pass.
e.g.: it takes 2 iterations to complete 1 epoch for a 1000 training sample case with batch size 500.
Since I start back with basic neural networks, I like epochs more in Keras.

Keras Components

After designing a neural network framework, it would be convenient to quickly translate it into Keras codes using its core components. https://keras.io/





1. Models, the way Keras organizes layers.
(1) Sequential: a linear stack of layers

```
from keras.models import Sequential
```

A sequential can be created via passing a list of layer instances to the constructor

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

or by initializing an empty one and adding layers later

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

The very first layer must assign a shape factor for the automatic shape inference of the following layers. e.g.

```
model.add(Dense(32, input_shape=(100,)))
```

or

```
model.add(Dense(32, input_dim=784))
```

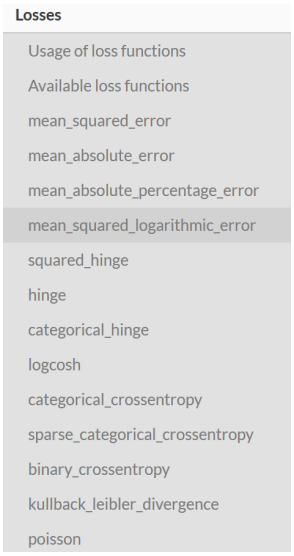
model.compile() specifies a learning process, this function requires three parameters:
a. “optimizer =” defines how we optimize the problem, e.g. sgd (stochastic gradient descent)
can be a string identifier of a built-in optimizer, or AN INSTANCE of the optimizer class. e.g.

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

Custom optimizer: pass a self-defined instance to function compile

```
from keras.optimizers import Optimizer
class my_opt(Optimizer):
    ''' a self-defined optimizer '''
my_opt_ins = my_opt()
model.compile(optimizer=my_opt_ins,
loss='categorical_crossentropy', metrics=['accuracy'])
```

b. “loss=” defines an objective that the model will try to minimize
can be a string identifier of a built-in loss, or a function.



Custom loss: pass a self-defined function to function compile

```
def my_loss(label_true, label_predict):
    ''' some functions '''
model.compile(optimizer='rmsprop',loss=my_loss,metrics=['accurac
y'])
```

c. "metrics=" defines a function that is used to judge the performance of a model
For classification it's `['accuracy']`
Custom metric: https://keras.io/metrics/

The training and prediction is just

```
model.fit(data, labels, epochs=10, batch_size=32)
model.evaluate(sample_test, label_test, batch_size=32)
```

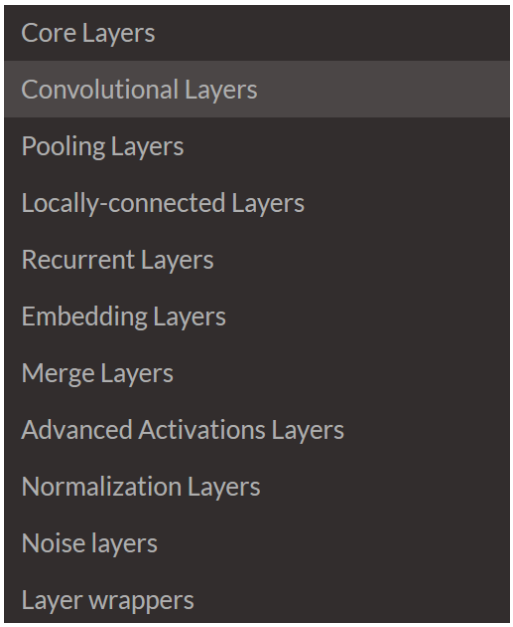
(2) Model (with functional API): defining complex models, such as multi-output models, directed acyclic graphs, or models with shared layers.
https://keras.io/getting-started/functional-api-guide/

```
from keras.layers import Input, Dense
from keras.models import Model
# This returns a tensor
inputs = Input(shape=(784,))
# a layer instance is callable on a tensor, and returns a tensor
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
```

```
predictions = Dense(10, activation='softmax')(x)
# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
```

2. Layers

Dense layers: output = activation(dot(input, kernel) + bias)



Custom Layers: https://keras.io/layers/writing-your-own-keras-layers/

Keras Notes

There are common problems in deep learning said to be “cannot proved”. But this does not imply that we can skip these problems during the study. Here summarizes / collects some common problems.

Parameter adjustment

Add more layers (self-defined layers) for complex data.

If the model is underfitting, add more nodes to layers, etc.

Because it indicates the model is too simple.

If the model is overfitting, add more training samples OR simplify current model.

Because it may be caused by the learning of the noise or random fluctuations in the training data by the model.

k-fold cross validation (resampling), validation dataset holdback, dropout, batch normalization aim to solve the common overfitting problem.

As the algorithm learns, the error for the model on the training data goes down and so does the error on the test dataset. If we train for too long, the performance on the training dataset may continue to decrease because the model is overfitting and learning the irrelevant detail and noise in the training dataset. At the same time the error for the test set starts to rise again as the model's ability to generalize decreases.
A good stopping spot is the point just before the error on the test dataset starts to increase where the model has good skill on both the training dataset and the unseen test dataset.

Non-smooth / up and down loss functions, try

Decrease the learning rate, increase the batch size, due to the unaveraged distribution of the sample, add regularizer.

For errors using custom loss function

Save/load model before saving/loading weights

model = load_model('model.h5', {'cosloss':my_loss})

For NAN loss value

Decrease the learning rate

Check the (custom) computation of loss function

Plot accuracy and loss function during each epoch, use the returning of fit()

```
history=model.fit()
plt.plot(); plt.plot(history.history['val_acc'])
plt.title('model accuracy'); plt.ylabel('accuracy'); plt.xlabel('epoch'); plt.legend(['train', 'test'], loc='upper left')
plt.show()

plt.plot(history.history['loss']); plt.plot(history.history['val_loss'])
plt.title('model loss'); plt.ylabel('loss'); plt.xlabel('epoch'); plt.legend(['train', 'test'], loc='upper left')
plt.show()

accy=history.history['acc']
np_accy=np.array(accy)
np.savetxt('save.txt',np_accy)
```

For the number of epoch

determine it by plotting the accuracy

define convergence using the loss < ?

batch normalization and dropout (overlapping purpose but can be used together)

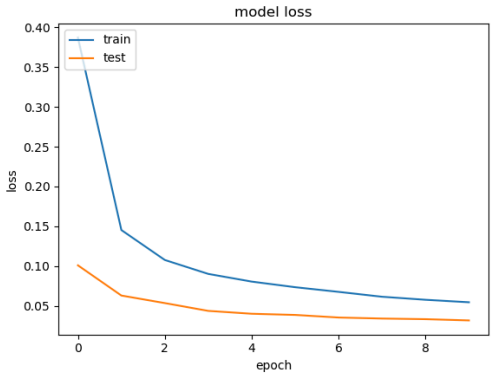
dropout is fast because just randomly abandoning data.

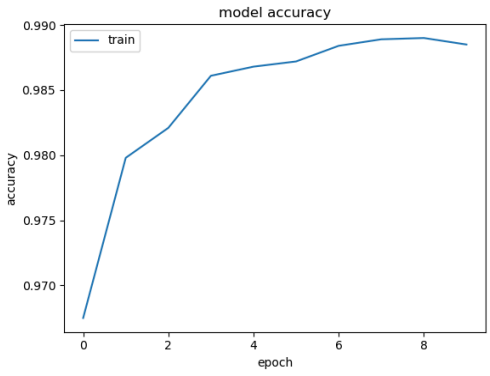
BN is slower in training, but focus on sample distribution, it not only prevents overfitting but also prevents vanishing gradient. The convergence speed is faster for BN

A CNN

This is an example presenting a CNN modeled in Keras for MNIST dataset. Mnist (http://yann.lecun.com/exdb/mnist/) is a handwriting dataset contains labeled digits from 0 to 9.

After only 10 epochs, the accuracy of the model can achieve around 99% on both the training and testing sets. It takes around 3 minutes to get the results:





1. Data preprocessing

Here we use the MNIST, shuffled and splitted between train and test, provided in keras:

```
from keras.datasets import mnist
from keras.utils import np_utils
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

There are 10 classes in this case (number 0 - 9), MNIST dataset consists of 28 * 28 images with one channel (grayscale). So:

```
nb_classes = 10
img_rows, img_cols = 28, 28
```

First we transform the labels (y) to categorical

```
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)
```

This changes the labels to the categorical format only turned on the labeled number, e.g. if the 1st label is 7):

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	1	0	0
...									

For the samples (X), we reshape them to (size(samples), img_rows, img_cols, n.o. channels) We also change the data type to 32 bit floats and roughly normalize each gray intensity by dividing 255 (the largest 8 bit intensity)

```
X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
print('X_train shape: ', X_train.shape)
print(X_train.shape[0], ' train samples')
print(X_test.shape[0], ' test samples')
```

2. The model

These are the sequential-organized layers used, the core is two convolutional layers and two fully connected layers.

Layer	Input	Filter size	Stride	Num Filters	Activation	Output
conv 1	(28,28)	(3,3)	(1,1)	32	relu	32*(28,28)
conv 2	32*(28,28)	(3,3)	(1,1)	32	relu	32*(28,28)
maxpool 1	32*(28,28)	(2,2)	(2,2)			32*(14,14)
Dropout 25%						
fc 1	32*(14,14)			128	relu	128
Dropout 50%						
fc 2	128			10	softmax	10

In codes

```
batch_size = 128
epochs = 10
nb_filters = 32
pool_size = (2, 2)
kernel_size = (3, 3)

from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D

model = Sequential()
model.add(Convolution2D(nb_filters,
                        (kernel_size[0], kernel_size[1]),
                        padding='same',
                        input_shape=input_shape)) # conv 1
model.add(Activation('relu'))
model.add(Convolution2D(nb_filters,
                        (kernel_size[0], kernel_size[1]))) #conv
2
model.add(Activation('relu'))
```

```
model.add(MaxPooling2D(pool_size=pool_size))
model.add(Dropout(0.25))
model.add(Flatten())  # 2D to 1D
model.add(Dense(128)) # fully connect 1
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes))       # fully connect 2
model.add(Activation('softmax'))
```

3. Compile and use the model
Using ‘categorical cross entropy’ as the loss function,
and ‘Adadelta optimizer’ (It is monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size)

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

training

```
history = model.fit(X_train, Y_train, batch_size=batch_size,
epochs=epochs,
          verbose=1, validation_data=(X_test, Y_test))
```

Save the model

```
model.save('m_cnn_minist.h5')
```

Test the model

```
from keras.models import load_model
model = load_model('m_cnn_minist.h5')

score = model.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

With 60000 train samples and 10000 test samples, the loss value is 0.032 and the test accuracy is 98.85% on the test samples.