Keras Introduction

Overview

What is Keras?

- Neural Network library written in Python
- Designed to be minimalistic & straight forward yet extensive
- Built on top of either **Theano** or **TensorFlow**

Why use Keras?

- Simple to get started, simple to keep going
- Written in python and highly modular; easy to expand Deep enough to build serious models

Documentation: http://keras.io/

Models

- **Sequential Model**
  a linear stack of layers

- **Model class used with functional API**
  a way to go for defining complex models, such as multi-output models, directed acyclic graphs, or models with shared layers

General idea

- Create first layer to handle input tensor
- Create output layer to handle targets
- Build virtually any model you like in between

Sequential Model

```
model = Sequential()
model.add(Dense(32, input_shape=(500,)))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
      loss='categorical_crossentropy',
      metrics=['acccuracy'])
```

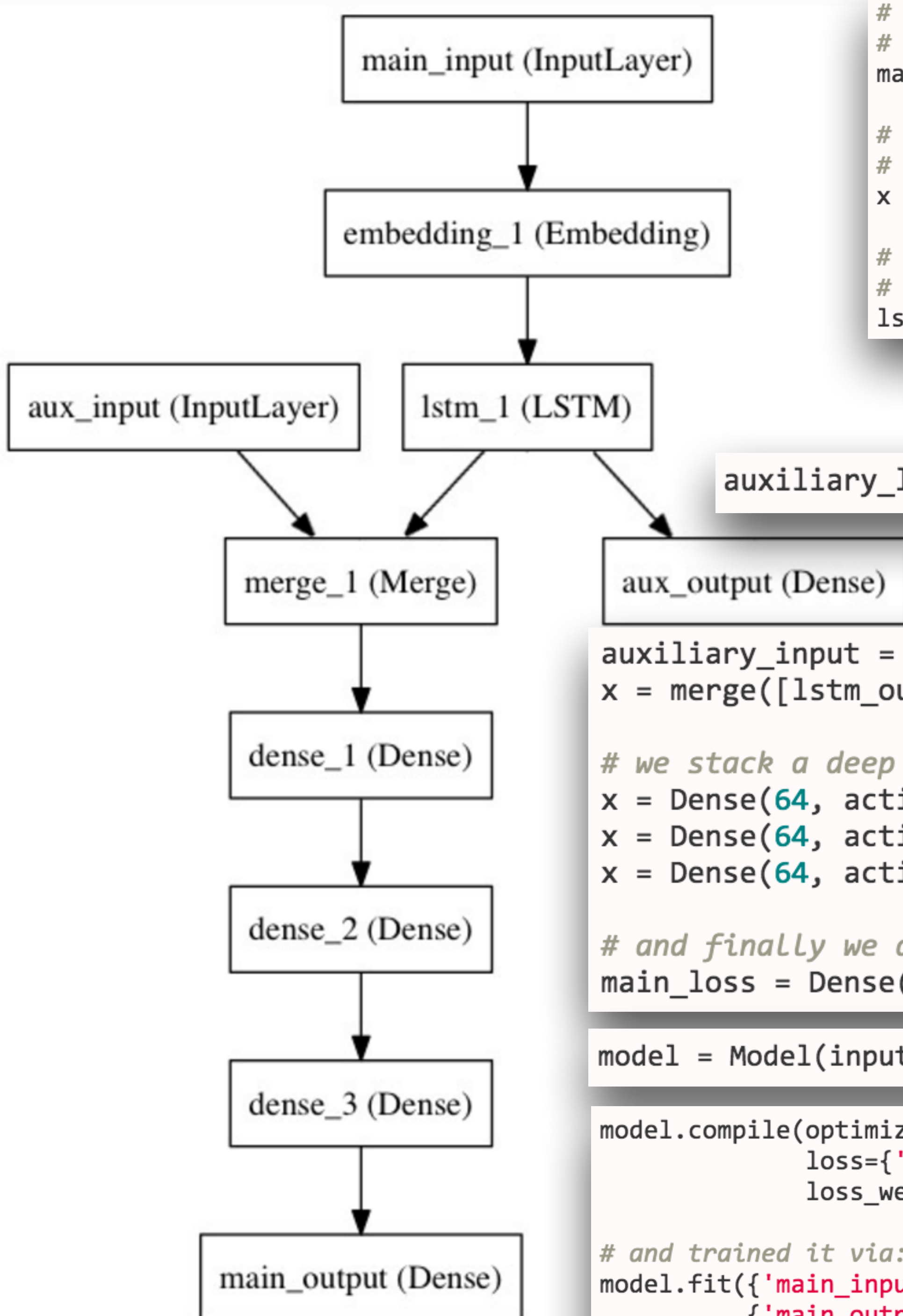
optimizer: str (name of optimizer) or optimizer object.
loss: str (name of objective function) or objective function.
metrics: list of metrics to be evaluated by the model during training/testing.

Methods

fit, evaluate, predict, predict_proba
train_on_batch, test_on_batch, predict_on_batch

http://keras.io/models/sequential/

Model Class API




```python
# headline input: meant to receive sequences of 100 integers, between 1 and 10000.
# note that we can name any layer by passing it a "name" argument.
main_input = Input(shape=(100,), dtype='int32', name='main_input')

# this embedding layer will encode the input sequence
# into a sequence of dense 512-dimensional vectors.
x = Embedding(output_dim=512, input_dim=10000, input_length=100)(main_input)

# a LSTM will transform the vector sequence into a single vector,
# containing information about the entire sequence
lstm_out = LSTM(32)(x)
```

```python
auxiliary_loss = Dense(1, activation='sigmoid', name='aux_output')(lstm_out)
```

```python
auxiliary_input = Input(shape=(5,), name='aux_input')
x = merge([lstm_out, auxiliary_input], mode='concat')

# we stack a deep fully-connected network on top
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)

# and finally we add the main logistic regression layer
main_loss = Dense(1, activation='sigmoid', name='main_output')(x)
```

```python
model = Model(input=[main_input, auxiliary_input], output=[main_loss, auxiliary_loss])
```

```python
model.compile(optimizer='rmsprop',
              loss={'main_output': 'binary_crossentropy', 'aux_output': 'binary_crossentropy'},
              loss_weight={'main_output': 1., 'aux_output': 0.2})

# and trained it via:
model.fit({'main_input': headline_data, 'aux_input': additional_data},
          {'main_output': labels, 'aux_output': labels},
          nb_epoch=50, batch_size=32)
```

http://keras.io/models/model/

Layers

- **Core Layers**

  Dense, Activation, Dropout, Merge…

- **Convolutional Layers**

- **Recurrent Layers**

  RNN, GRU, LSTM

```
model = Sequential()
model.add(RNN(HIDDEN_SIZE, input_shape=(N_gram, vocabulary.capacity), return_sequences=True))
model.add(RNN(HIDDEN_SIZE))
model.add(Dense(2, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Writing Your Own Keras Layers

Only three methods needed to implement

build(input_shape): this is where you will define your weights.
call(x): this is where the layer's logic lives.
get_output_shape_for(input_shape): allow to do automatic shape inference.

```
from keras import backend as K
from keras.engine.topology import Layer

class my_layer(Layer):
    def __init__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(Layer, self).__init__(**kwargs)

    def build(self, input_shape):
        input_dim = input_shape[1]
        initial_weight_value = np.random.random((input_dim, output_dim))
        self.W = K.variable(initial_weight_value)
        self.trainable_weights = [self.W]

    def call(self, x, mask=None):
        return K.dot(x, self.W)

    def get_output_shape_for(self, input_shape):
        return (input_shape[0] + self.output_dim)
```

Take RNN as an example
https://github.com/fchollet/keras/blob/master/keras/layers/recurrent.py

Activations

More or less all your favorite activations are available:

- Sigmoid, tanh, ReLu, softplus, hard sigmoid, linear
- Advanced activations implemented as a layer
- Advanced activations: LeakyReLu, PReLu, ELU, Parametric Softplus, Thresholded linear and Thresholded Relu

Objectives and Optimizers

Objective Functions:

- Error loss: rmse, mse, mae, mape, msle
- Hinge loss: squared hinge, hinge
- Class loss: binary crossentropy, categorical crossentropy

Optimization:

- Provides SGD, Adagrad, Adadelta, Rmsprop and Adam
- All optimizers can be customized via parameters

Running on GPU

- Automatically run on GPU when using **TensorFlow** backend
- If running on the **Theano** backend, you can use one of the following methods

Method 1: use Theano flags.

```
THEANO_FLAGS=device=gpu,floatX=float32 python my_keras_script.py
```

The name 'gpu' might have to be changed depending on your device's identifier (e.g. `gpu0`, `gpu1`, etc).

Method 2: set up your `.theanorc`: Instructions

Method 3: manually set `theano.config.device`, `theano.config.floatX` at the beginning of your code:

```
import theano
theano.config.device = 'gpu'
theano.config.floatX = 'float32'
```

Model Saving

- Model architecture can be saved and loaded

```
# save as JSON
json_string = model.to_json()

# save as YAML
yaml_string = model.to_yaml()
```

```
# model reconstruction from JSON:
from keras.models import model_from_json
model = model_from_json(json_string)

# model reconstruction from YAML
model = model_from_yaml(yaml_string)
```

- Model parameters (weights) can be saved and loaded

```
model.save_weights('my_model_weights.h5')
```

```
model.load_weights('my_model_weights.h5')
```

Callbacks

- You can use callbacks to get a view on internal states and statistics of the model during training.
- Methods: *History, ModelCheckpoint, EarlyStopping…*

```
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))

model = Sequential()
model.add(Dense(10, input_dim=784, init='uniform'))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

history = LossHistory()
model.fit(X_train, Y_train, batch_size=128, nb_epoch=20, verbose=0, callbacks=[history])

print history.losses
# outputs
'''
[0.66047596406559383, 0.3547245744908703, ..., 0.25953155204159617, 0.25901699725311789]
'''
```

In Summary

Pros:

- Easy to implement
- Lots of choice
- Extendible and customizable GPU
- High level
- Active community
- keras.io

Cons:

- Lack of generative models
- Theano overhead