Keras: An Introduction

Dylan Drover

STAT 946

December 2, 2015
Overview

What is Keras?
- Neural Network library written in Python
- Designed to be minimalistic & straightforward yet extensive
- Built on top of either Theano as newly 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
General Design

General idea is to based on layers and their input/output

- Prepare your inputs and output tensors
- Create first layer to handle input tensor
- Create output layer to handle targets
- Build virtually any model you like in between
Layers and Layers (like an Ogre)

Keras has a number of pre-built layers. Notable examples include:

- Regular dense, MLP type

```python
keras.layers.core.Dense(output_dim,
                          init='glorot_uniform',
                          activation='linear',
                          weights=None,
                          W_regularizer=None, b_regularizer=None, activity_regularizer=None,
                          W_constraint=None, b_constraint=None,
                          input_dim=None)
```

- Recurrent layers, LSTM, GRU, etc.

```python
keras.layers.recurrent.GRU(output_dim,
                           init='glorot_uniform', inner_init='orthogonal',
                           activation='sigmoid', inner_activation='hard_sigmoid',
                           return_sequences=False,
                           go_backwards=False,
                           stateful=False,
                           input_dim=None, input_length=None)
```
1D Convolutional layers

```python
keras.layers.convolutional.Convolution1D(nb_filter, filter_length,
    init='uniform',
    activation='linear',
    weights=None,
    border_mode='valid',
    subsample_length=1,
    W_regularizer=None, b_regularizer=None,
    W_constraint=None, b_constraint=None,
    input_dim=None, input_length=None)
```

2D Convolutional layers

```python
keras.layers.convolutional.Convolution2D(nb_filter, nb_row, nb_col,
    init='glorot_uniform',
    activation='linear',
    weights=None,
    border_mode='valid',
    subsample=(1, 1),
    W_regularizer=None, b_regularizer=None,
    W_constraint=None,
    dim_ordering='th')
```
Autoencoders can be built with any other type of layer

```python
from keras.layers import containers

# input shape: (nb_samples, 32)
encoder = containers.Sequential([Dense(16, input_dim=32), Dense(8)])
decoder = containers.Sequential([Dense(16, input_dim=8), Dense(32)])

autoencoder = Sequential()
autoencoder.add(AutoEncoder(encoder=encoder, decoder=decoder, output_reconstruction=False))
```
Other types of layer include:

- Dropout
- Noise
- Pooling
- Normalization
- Embedding
- And many more...
More or less all your favourite activations are available:

- Sigmoid, \( \tanh \), ReLu, softplus, hard\_sigmoid, linear
- Advanced activations implemented as a layer (after desired neural 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
Parallel Capabilities

- Training time is drastically reduced thanks to Theano’s GPU support
- Theano compiles into CUDA, NVIDIA’s GPU API
- Currently will only work with NVIDIA cards but Theano is working on OpenCL version
- TensorFlow has similar support
- THEANO_FLAGS=mode=FAST_RUN,device=gpu, floatX=float32 python your_net.py
Model architectures can be saved and loaded

```python
# 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

```python
model.save_weights('my_model_weights.h5')
model.load_weights('my_model_weights.h5')
```
Callbacks

Allow for function call during training

- Callbacks can be called at different points of training (batch or epoch)
- Existing callbacks: Early Stopping, weight saving after epoch
- Easy to build and implement, called in training function, fit()
Model Type: Sequential

- Sequential models are linear stack of layers
- The model we all know and love
- Treat each layer as object that feeds into the next
# Build and train model

```python
AE_0 = Sequential()

encoder = Sequential([GRU(50, activation='relu', inner_activation='hard_sigmoid', input_dim=6, return_sequences=True)])

decoder = Sequential([GRU(6, input_dim=50, activation='relu', inner_activation='hard_sigmoid', return_sequences=True)])

AE_0.add(AutoEncoder(encoder=encoder, decoder=decoder, output_reconstruction=True))
AE_0.compile(loss='mse', optimizer='rmsprop')
AE_0.fit(X_train, X_train, batch_size=16, nb_epoch=15, show_accuracy=True)

temp = Sequential()
temp.add(encoder)
temp.compile(loss='mse', optimizer='rmsprop')

first_output = temp.predict(X_train, batch_size=16)

AE_1 = Sequential()

encoder_0 = Sequential([GRU(60, activation='relu', inner_activation='hard_sigmoid', input_dim=50, return_sequences=True)])

decoder_0 = Sequential([GRU(50, input_dim=60, activation='relu', inner_activation='hard_sigmoid', return_sequences=True)])

AE_1.add(AutoEncoder(encoder=encoder_0, decoder=decoder_0, output_reconstruction=True))
AE_1.compile(loss='mse', optimizer='rmsprop')
AE_1.fit(first_output, first_output, batch_size=16, nb_epoch=15, show_accuracy=True)

encoder_0.save_weights('encoder_saved_pre_weights_lb_2GRU.h5', overwrite=True)
```
# Second autoencoder for second and third layers of final NN

```python
AE_2 = Sequential()

encoder_1 = Sequential([Dense(50, input_dim=60, activation='relu')])
decoder_1 = Sequential([Dense(60, input_dim=50, activation='relu')])

AE_2.add(AutoEncoder(encoder=encoder_1, decoder=decoder_1, output_reconstruction=True))
AE_2.compile(loss='mse', optimizer='rmsprop')

AE_2.fit(second_output, second_output, batch_size=16, nb_epoch=100, show_accuracy=True)
```

# Create full model with first two layers of autoencoders and an output layer with supervised learning

```python
full_model = Sequential()

full_model.add(encoder)
full_model.add(model.layers[0])
full_model.add(encoder_1)
full_model.add(Dense(1, activation='sigmoid'))
full_model.load_weights('tmp_weights_23.hdf5')
full_model.compile(loss='binary_crossentropy', optimizer='adam', class_mode='binary')

full_model.fit(X_train, y_train, batch_size=32, nb_epoch=25, show_accuracy=True, callbacks=[model_check])
```

```python
score, acc = full_model.evaluate(X_test, y_test, batch_size=8, show_accuracy=True)
```
Model Type: Graph

- Optimized over all outputs
- Graph model allows for two or more independent networks to diverge or merge
- Allows for multiple separate inputs or outputs
- Different merging layers (sum or concatenate)
vert_test = np.dstack((X_test[:,:,0], X_test[:,:,3], X_test[:,:,7]))
front_test = np.dstack((-X_test[:,:,1], X_test[:,:,5], X_test[:,:,8]))
side_test = np.dstack((X_test[:,:,2], X_test[:,:,4], X_test[:,:,6]))

Set things up such that each input takes an axis as input

model = Graph()

model.add_input(name='vert', input_shape=(16501, 3))
model.add_input(name='front', input_shape=(16501, 3))
model.add_input(name='side', input_shape=(16501, 3))

Filter for the vertical axis

model.add_node(Convolution1D(nb_filter=20, filter_length=5, activation='relu', input_dim=3, input_length=16501), name='con_vert', input='vert')
model.add_node(Dropout(0.5), name='drop_vert', input='con_vert')
model.add_node(MaxPooling1D(pool_length=10), name='pool_vert', input='drop_vert')
model.add_node(Flatten(), name='flat_vert', input='pool_vert')

Filter for the front axis

model.add_node(Convolution1D(nb_filter=20, filter_length=5, activation='relu', input_dim=3, input_length=16501), name='con_front', input='front')
model.add_node(Dropout(0.5), name='drop_front', input='con_front')
model.add_node(MaxPooling1D(pool_length=10), name='pool_front', input='drop_front')
model.add_node(Flatten(), name='flat_front', input='pool_front')
```python
model.add_node(Dense(200, activation='relu'), name='combine', inputs=['flat_vert', 'flat_front', 'flat_side'],
               merge_mode='concat')
model.add_node(Dropout(0.5), name='drop_combine', input='combine')
model.add_node(Dense(40, activation='relu'), name='mlp_2', input='drop_combine')
model.add_node(Dropout(0.5), name='drop_mlp2', input='mlp_2')
model.add_node(Dense(2, activation='softmax'), name='mlp_out', input='drop_mlp2')
model.add_output(name='output', input='mlp_out')

model.compile('adam', {'output': 'categorical_crossentropy'})
model.fit({'vert': vert, 'front': front, 'side': side, 'output': y_train}, batch_size=5, nb_epoch=150,
           validation_split=0.2)

outs = model.predict({'vert': vert_test, 'front': front_test, 'side': side_test}, batch_size=8, verbose=1)

classes = np.round(outs['output'].astype(float), decimals=0)
```
Sarcasm detection in Amazon.com reviews:

- Based on theory that sarcasm can be detected using sentiment transitions
- Training set was separated into sarcastic and regular reviews
- Stanford recursive sentiment was run on each sentence to create sentiment vector
In Summary

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

Cons:
- Lack of generative models
- High level
- Theano overhead