LEARNING keras

#keras

Table of Contents

About

Chapter 1: Getting started with keras

Remarks

Guiding principles

- **Modularity**

A model is understood as a sequence or a graph of standalone, fully-configurable modules that can be plugged together with as little restrictions as possible. In particular, neural layers, cost functions, optimizers, initialization schemes, activation functions, regularization schemes are all standalone modules that you can combine to create new models.

- **Minimalism**

Each module should be kept short and simple. Every piece of code should be transparent upon first reading. No black magic: it hurts iteration speed and ability to innovate.

- **Easy extensibility**

New modules are dead simple to add (as new classes and functions), and existing modules provide ample examples. To be able to easily create new modules allows for total expressiveness, making Keras suitable for advanced research.

- **Work with Python**

No separate models configuration files in a declarative format. Models are described in Python code, which is compact, easier to debug, and allows for ease of extensibility.

Examples

Installation and Setup

Keras is a high-level neural networks library, written in Python and capable of running on top of either TensorFlow 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 total modularity, minimalism, and extensibility).
- Supports both convolutional networks and recurrent networks, as well as combinations of the two.
- Supports arbitrary connectivity schemes (including multi-input and multi-output training).
- Runs seamlessly on CPU and GPU.

Installation

Keras uses the following dependencies:

- numpy, scipy
- pyyaml
- HDF5 and h5py (optional, required if you use model saving/loading functions)
- Optional but recommended if you use CNNs: cuDNN
- scikit-image (optional, required if you use keras built-in functions for preprocessing and augmenting image data)

Keras is a high-level library that provides a convenient Machine Learning API on top of other low-level libraries for tensor processing and manipulation, called *Backends*. At this time, Keras can be used on top any of the three available backends: *TensorFlow*, *Theano*, and *CNTK*.

Theano is installed automatically if you install *Keras* using *pip*. If you want to install *Theano* manually, please refer to *Theano* installation instructions.

TensorFlow is a recommended option, and by default, *Keras* uses *TensorFlow* backend, if available. To install *TensorFlow*, the easiest way is to do

```
$ pip install tensorflow
```

If you want to install it manually, please refer to *TensorFlow* installation instructions.

To install *Keras*, cd to the *Keras* folder and run the install command:

```
$ python setup.py install
```

You can also install Keras from PyPI:

```
$ pip install keras
```

Configuration

If you have run Keras at least once, you will find the Keras configuration file at:

```
~/.keras/keras.json
```

If it isn't there, you can create it. The default configuration file looks like this:

```
{
    "image_dim_ordering": "tf",
    "epsilon": 1e-07,
    "floatx": "float32",
    "backend": "tensorflow"
}
```

Switching from TensorFlow to Theano

By default, Keras will use TensorFlow as its tensor manipulation library. If you want to use other backend, simply change the field backend to either `"theano"` or `"tensorflow"`, and Keras will use the new configuration next time you run any Keras code.

Getting Started with Keras : 30 Second

The core data structure of Keras is a **model**, a way to organize layers. The main type of model is the **Sequential** model, a linear stack of layers. For more complex architectures, you should use the Keras functional API.

Here's the Sequential model:

```
from keras.models import Sequential

model = Sequential()
```

Stacking layers is as easy as `.add()`:

```
from keras.layers import Dense, Activation

model.add(Dense(output_dim=64, input_dim=100))
model.add(Activation("relu"))
model.add(Dense(output_dim=10))
model.add(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).

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

You can now iterate on your training data in batches:

```
model.fit(X_train, Y_train, nb_epoch=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=32)
```

Or generate predictions on new data:

```
classes = model.predict_classes(X_test, batch_size=32)
proba = model.predict_proba(X_test, batch_size=32)
```

Building a question answering system, an image classification model, a Neural Turing Machine, a word2vec embedder or any other model is just as fast. The ideas behind deep learning are simple, so why should their implementation be painful?

You will find more advanced models: question-answering with memory networks, text generation with stacked LSTMs, etc in example folder.

Read Getting started with keras online: https://riptutorial.com/keras/topic/8695/getting-started-with-keras

Chapter 2: Classifying Spatiotemporal Inputs with CNNs, RNNs, and MLPs

Introduction

Spatiotemporal data, or data with spatial and temporal qualities, are a common occurrence. Examples include videos, as well as sequences of image-like data, such as spectrograms.

Convolutional Neural Networks (CNNs) are particularly suited for finding spatial patterns. Recurrent Neural Networks (RNNs), on the other hand, are particularly suited for finding temporal patterns. These two, in combination with Multilayer Perceptrons, can be effective for classifying spatiotemporal inputs.

Remarks

In this example, a VGG-16 model pre-trained on the ImageNet database was used. If a trainable VGG-16 model is desired, set the VGG-16 `weights` parameter to `None` for random initialization and set the `cnn.trainable` attribute to `True`.

The number and kind of layers, units, and other parameters should be tweaked as necessary for specific application needs.

Examples

VGG-16 CNN and LSTM for Video Classification

For this example, let's assume that the inputs have a dimensionality of *(frames, channels, rows, columns)*, and the outputs have a dimensionality of *(classes)*.

```
from keras.applications.vgg16 import VGG16
from keras.models import Model
from keras.layers import Dense, Input
from keras.layers.pooling import GlobalAveragePooling2D
from keras.layers.recurrent import LSTM
from keras.layers.wrappers import TimeDistributed
from keras.optimizers import Nadam

video = Input(shape=(frames,
                     channels,
                     rows,
                     columns))
cnn_base = VGG16(input_shape=(channels,
                              rows,
                              columns),
                 weights="imagenet",
                 include_top=False)
cnn_out = GlobalAveragePooling2D()(cnn_base.output)
cnn = Model(input=cnn_base.input, output=cnn_out)
```

```
cnn.trainable = False
encoded_frames = TimeDistributed(cnn)(video)
encoded_sequence = LSTM(256)(encoded_frames)
hidden_layer = Dense(output_dim=1024, activation="relu")(encoded_sequence)
outputs = Dense(output_dim=classes, activation="softmax")(hidden_layer)
model = Model([video], outputs)
optimizer = Nadam(lr=0.002,
                  beta_1=0.9,
                  beta_2=0.999,
                  epsilon=1e-08,
                  schedule_decay=0.004)
model.compile(loss="categorical_crossentropy",
              optimizer=optimizer,
              metrics=["categorical_accuracy"])
```

Read Classifying Spatiotemporal Inputs with CNNs, RNNs, and MLPs online: https://riptutorial.com/keras/topic/9658/classifying-spatiotemporal-inputs-with-cnns--rnns--and-mlps

Chapter 3: Create a simple Sequential Model

Introduction

The `Sequential` model is a linear stack of layers.

Examples

Simple Multi Layer Perceptron wtih Sequential Models

You can create a Sequential model 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_dim=784),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

You can also simply add layers via the `.add()` method:

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

Models must be compiled before use:

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

Read Create a simple Sequential Model online: https://riptutorial.com/keras/topic/8850/create-a-simple-sequential-model

Chapter 4: Custom loss function and metrics in Keras

Introduction

You can create a custom loss function and metrics in Keras by defining a TensorFlow/Theano symbolic function that returns a scalar for each data-point and takes the following two arguments: tensor of true values, tensor of the corresponding predicted values.

Note that the loss/metric (for display and optimization) is calculated as the mean of the losses/metric across all datapoints in the batch.

Remarks

Keras loss functions are defined in losses.py

Additional loss functions for Keras can be found in keras-contrib repository.

Examples

Euclidean distance loss

Define a custom loss function:

```
import keras.backend as K


def euclidean_distance_loss(y_true, y_pred):
    """
    Euclidean distance loss
    https://en.wikipedia.org/wiki/Euclidean_distance
    :param y_true: TensorFlow/Theano tensor
    :param y_pred: TensorFlow/Theano tensor of the same shape as y_true
    :return: float
    """
    return K.sqrt(K.sum(K.square(y_pred - y_true), axis=-1))
```

Use it:

```
model.compile(loss=euclidean_distance_loss, optimizer='rmsprop')
```

Read Custom loss function and metrics in Keras online: https://riptutorial.com/keras/topic/10674/custom-loss-function-and-metrics-in-keras

Chapter 5: Dealing with large training datasets using Keras fit_generator, Python generators, and HDF5 file format

Introduction

Machine learning problems often require dealing with large quantities of training data with limited computing resources, particularly memory. It is not always possible to load an entire training set into memory. Fortunately, this can be dealt with through the use of Keras' fit_generator method, Python generators, and HDF5 file format.

Remarks

This example assumes keras, numpy (as np), and h5py have already been installed and imported. It also assumes that video inputs and labels have already been processed and saved to the specified HDF5 file, in the format mentioned, and a video classification model has already been built to work with the given input.

Examples

Training a model to classify videos

For this example, let **model** be a Keras model for classifying video inputs, let **X** be a large data set of video inputs, with a shape of *(samples, frames, channels, rows, columns)*, and let **Y** be the corresponding data set of one-hot encoded labels, with a shape of *(samples, classes)*. Both datasets are stored within an HDF5 file called **video_data.h5**. The HDF5 file also has the attribute **sample_count** for the number of samples.

Here is the function for training the model with fit_generator

```
def train_model(model, video_data_fn="video_data.h5", validation_ratio=0.3, batch_size=32):
    """ Train the video classification model
    """
    with h5py.File(video_data_fn, "r") as video_data:
         sample_count = int(video_data.attrs["sample_count"])
         sample_idxs = range(0, sample_count)
         sample_idxs = np.random.permutation(sample_idxs)
         training_sample_idxs = sample_idxs[0:int((1-validation_ratio)*sample_count)]
         validation_sample_idxs = sample_idxs[int((1-validation_ratio)*sample_count):]
         training_sequence_generator = generate_training_sequences(batch_size=batch_size,
                                                                   video_data=video_data,

training_sample_idxs=training_sample_idxs)
         validation_sequence_generator = generate_validation_sequences(batch_size=batch_size,
                                                                       video_data=video_data,
validation_sample_idxs=validation_sample_idxs)
         model.fit_generator(generator=training_sequence_generator,
                             validation_data=validation_sequence_generator,
                             samples_per_epoch=len(training_sample_idxs),
                             nb_val_samples=len(validation_sample_idxs),
                             nb_epoch=100,
                             max_q_size=1,
                             verbose=2,
                             class_weight=None,
                             nb_worker=1)
```

Here are the training and validation sequence generators

```
def generate_training_sequences(batch_size, video_data, training_sample_idxs):
    """ Generates training sequences on demand
    """
    while True:
        # generate sequences for training
        training_sample_count = len(training_sample_idxs)
        batches = int(training_sample_count/batch_size)
        remainder_samples = training_sample_count%batch_size
        if remainder_samples:
            batches = batches + 1
        # generate batches of samples
        for idx in xrange(0, batches):
            if idx == batches - 1:
                batch_idxs = training_sample_idxs[idx*batch_size:]
            else:
                batch_idxs = training_sample_idxs[idx*batch_size:idx*batch_size+batch_size]
            batch_idxs = sorted(batch_idxs)

            X = video_data["X"][batch_idxs]
            Y = video_data["Y"][batch_idxs]

            yield (np.array(X), np.array(Y))

def generate_validation_sequences(batch_size, video_data, validation_sample_idxs):
    """ Generates validation sequences on demand
    """
    while True:
        # generate sequences for validation
        validation_sample_count = len(validation_sample_idxs)
        batches = int(validation_sample_count/batch_size)
        remainder_samples = validation_sample_count%batch_size
        if remainder_samples:
            batches = batches + 1
        # generate batches of samples
        for idx in xrange(0, batches):
            if idx == batches - 1:
                batch_idxs = validation_sample_idxs[idx*batch_size:]
            else:
                batch_idxs = validation_sample_idxs[idx*batch_size:idx*batch_size+batch_size]
            batch_idxs = sorted(batch_idxs)

            X = video_data["X"][batch_idxs]
            Y = video_data["Y"][batch_idxs]

            yield (np.array(X), np.array(Y))
```

Read Dealing with large training datasets using Keras fit_generator, Python generators, and HDF5 file format online: https://riptutorial.com/keras/topic/9656/dealing-with-large-training-datasets-using-keras-fit-generator--python-generators--and-hdf5-file-format

Chapter 6: Transfer Learning and Fine Tuning using Keras

Introduction

This topic includes short, brief but comprehensive examples of loading pre-trained weights, inserting new layers on top or in the middle of pre-tained ones, and training a new network with partly pre-trained weights. An example for each of out-of-the-box pre-trained networks, available in *Keras* library (VGG, ResNet, Inception, Xception, MobileNet), is required.

Examples

Transfer Learning using Keras and VGG

In this example, three brief and comprehensive sub-examples are presented:

- Loading weights from available pre-trained models, included with *Keras* library
- Stacking another network for training on top of any layers of VGG
- Inserting a layer in the middle of other layers
- Tips and general rule-of-thumbs for Fine-Tuning and transfer learning with VGG

Loading pre-trained weights

Pre-trained on *ImageNet* models, including *VGG-16* and *VGG-19*, are available in *Keras*. Here and after in this example, *VGG-16* will be used. For more information, please visit *Keras Applications documentation*.

```
from keras import applications

# This will load the whole VGG16 network, including the top Dense layers.
# Note: by specifying the shape of top layers, input tensor shape is forced
# to be (224, 224, 3), therefore you can use it only on 224x224 images.
vgg_model = applications.VGG16(weights='imagenet', include_top=True)

# If you are only interested in convolution filters. Note that by not
# specifying the shape of top layers, the input tensor shape is (None, None, 3),
# so you can use them for any size of images.
vgg_model = applications.VGG16(weights='imagenet', include_top=False)

# If you want to specify input tensor
from keras.layers import Input
input_tensor = Input(shape=(160, 160, 3))
vgg_model = applications.VGG16(weights='imagenet',
                               include_top=False,
                               input_tensor=input_tensor)
```

```
# To see the models' architecture and layer names, run the following
vgg_model.summary()
```

Create a new network with bottom layers taken from VGG

Assume that for some specific task for images with the size `(160, 160, 3)`, you want to use pre-trained bottom layers of VGG, up to layer with the name `block2_pool`.

```
vgg_model = applications.VGG16(weights='imagenet',
                               include_top=False,
                               input_shape=(160, 160, 3))

# Creating dictionary that maps layer names to the layers
layer_dict = dict([(layer.name, layer) for layer in vgg_model.layers])

# Getting output tensor of the last VGG layer that we want to include
x = layer_dict['block2_pool'].output

# Stacking a new simple convolutional network on top of it
x = Conv2D(filters=64, kernel_size=(3, 3), activation='relu')(x)
x = MaxPooling2D(pool_size=(2, 2))(x)
x = Flatten()(x)
x = Dense(256, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(10, activation='softmax')(x)

# Creating new model. Please note that this is NOT a Sequential() model.
from keras.models import Model
custom_model = Model(input=vgg_model.input, output=x)

# Make sure that the pre-trained bottom layers are not trainable
for layer in custom_model.layers[:7]:
    layer.trainable = False

# Do not forget to compile it
custom_model.compile(loss='categorical_crossentropy',
                     optimizer='rmsprop',
                     metrics=['accuracy'])
```

Remove multiple layers and insert a new one in the middle

Assume that you need to speed up VGG16 by replacing `block1_conv1` and `block2_conv2` with a single convolutional layer, in such a way that the pre-trained weights are saved. The idea is to disassemble the whole network to separate layers, then assemble it back. Here is the code specifically for your task:

```
vgg_model = applications.VGG16(include_top=True, weights='imagenet')

# Disassemble layers
layers = [l for l in vgg_model.layers]

# Defining new convolutional layer.
# Important: the number of filters should be the same!
# Note: the receiptive field of two 3x3 convolutions is 5x5.
new_conv = Conv2D(filters=64,
                  kernel_size=(5, 5),
                  name='new_conv',
                  padding='same')(layers[0].output)

# Now stack everything back
# Note: If you are going to fine tune the model, do not forget to
#       mark other layers as un-trainable
x = new_conv
for i in range(3, len(layers)):
    layers[i].trainable = False
    x = layers[i](x)

# Final touch
result_model = Model(input=layer[0].input, output=x)
```

Read Transfer Learning and Fine Tuning using Keras online:
https://riptutorial.com/keras/topic/10887/transfer-learning-and-fine-tuning-using-keras

Credits

S. No	Chapters	Contributors
1	Getting started with keras	Arman, Community, FalconUA
2	Classifying Spatiotemporal Inputs with CNNs, RNNs, and MLPs	Robert Valencia
3	Create a simple Sequential Model	Arman, Sam Zeng
4	Custom loss function and metrics in Keras	FalconUA, Sergii Gryshkevych
5	Dealing with large training datasets using Keras fit_generator, Python generators, and HDF5 file format	Robert Valencia
6	Transfer Learning and Fine Tuning using Keras	FalconUA