Usage of callbacks

A callback is a set of functions to be applied at given stages of the training procedure. You can use callbacks to get a view on internal states and statistics of the model during training. You can pass a list of callbacks (as the keyword argument `callbacks`) to the `.fit()` method of the `Sequential` model. The relevant methods of the callbacks will then be called at each stage of the training.

BaseLogger

[source]

```
keras.callbacks.BaseLogger()
```

Callback that accumulates epoch averages of the metrics being monitored.

This callback is automatically applied to every Keras model.

Callback

[source]

```
keras.callbacks.Callback()
```

Abstract base class used to build new callbacks.

Properties

- **params**: dict. Training parameters (eg. verbosity, batch size, number of epochs...).
- **model**: instance of `keras.models.Model`. Reference of the model being trained.

The `logs` dictionary that callback methods take as argument will contain keys for quantities relevant to the current batch or epoch.

Currently, the `.fit()` method of the `Sequential` model class will include the following quantities in the `logs` that it passes to its callbacks:

- **on_epoch_end**: logs include `acc` and `loss`, and optionally include `val_loss` (if validation is enabled in `fit`), and `val_acc` (if validation and accuracy monitoring are enabled).
- **on_batch_begin**: logs include `size`, the number of samples in the current batch.
- **on_batch_end**: logs include `loss`, and optionally `acc` (if accuracy monitoring is enabled).

ProgbarLogger

[source]

```
keras.callbacks.ProgbarLogger()
```

Callback that prints metrics to stdout.

History

[source]

```
keras.callbacks.History()
```

Callback that records events into a `History` object.

This callback is automatically applied to every Keras model. The `History` object gets returned by the `fit` method of models.

ModelCheckpoint

[source]

```
keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss', verbose=0, save_best_only=False,
```

Save the model after every epoch.

`filepath` can contain named formatting options, which will be filled the value of `epoch` and keys in `logs` (passed in `on_epoch_end`).

For example: if `filepath` is `weights.{epoch:02d}-{val_loss:.2f}.hdf5`, then multiple files will be save with the epoch number and the validation loss.

Arguments

- **filepath**: string, path to save the model file.
- **monitor**: quantity to monitor.
- **verbose**: verbosity mode, 0 or 1.
- **save_best_only**: if `save_best_only=True`, the latest best model according to the validation loss will not be overwritten.
- **mode**: one of {auto, min, max}. If `save_best_only=True`, the decision to overwrite the current save file is made based on either the maximization or the minization of the monitored. For `val_acc`, this should be `max`, for `val_loss` this should be `min`, etc. In `auto` mode, the direction is automatically inferred from the name of the monitored quantity.
- **save_weights_only**: if True, then only the model's weights will be saved

(`model.save_weights(filepath)`), else the full model is saved (`model.save(filepath)`).

EarlyStopping

[source]

```
keras.callbacks.EarlyStopping(monitor='val_loss', patience=0, verbose=0, mode='auto')
```

Stop training when a monitored quantity has stopped improving.

Arguments

- **monitor**: quantity to be monitored.
- **patience**: number of epochs with no improvement after which training will be stopped.
- **verbose**: verbosity mode.
- **mode**: one of {auto, min, max}. In 'min' mode, training will stop when the quantity monitored has stopped decreasing; in 'max' mode it will stop when the quantity monitored has stopped increasing.

RemoteMonitor

[source]

```
keras.callbacks.RemoteMonitor(root='http://localhost:9000', path='/publish/epoch/end/', field='
```

Callback used to stream events to a server.

Requires the `requests` library.

Arguments

- **root**: root url to which the events will be sent (at the end of every epoch). Events are sent to `root + '/publish/epoch/end/'` by default. Calls are HTTP POST, with a `data` argument which is a JSON-encoded dictionary of event data.

LearningRateScheduler

[source]

```
keras.callbacks.LearningRateScheduler(schedule)
```

Learning rate scheduler.

Arguments

- **schedule**: a function that takes an epoch index as input (integer, indexed from 0) and returns a new learning rate as output (float).

TensorBoard

[source]

```
keras.callbacks.TensorBoard(log_dir='./logs', histogram_freq=0, write_graph=True)
```

Tensorboard basic visualizations.

This callback writes a log for TensorBoard, which allows you to visualize dynamic graphs of your training and test metrics, as well as activation histograms for the different layers in your model.

TensorBoard is a visualization tool provided with TensorFlow.

If you have installed TensorFlow with pip, you should be able to launch TensorBoard from the command line:

```
tensorboard --logdir=/full_path_to_your_logs
```

You can find more information about TensorBoard - __here.

Arguments

- **log_dir**: the path of the directory where to save the log files to be parsed by Tensorboard
- **histogram_freq**: frequency (in epochs) at which to compute activation histograms for the layers of the model. If set to 0, histograms won't be computed.
- **write_graph**: whether to visualize the graph in Tensorboard. The log file can become quite large when write_graph is set to True.

Create a callback

You can create a custom callback by extending the base class `keras.callbacks.Callback`. A callback has access to its associated model through the class property `self.model`.

Here's a simple example saving a list of losses over each batch during training:

```
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'))
```

Example: recording loss history

```
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]
'''
```

Example: model checkpoints

```
from keras.callbacks import ModelCheckpoint

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

'''
saves the model weights after each epoch if the validation loss decreased
'''
checkpointer = ModelCheckpoint(filepath="/tmp/weights.hdf5", verbose=1, save_best_only=True)
model.fit(X_train, Y_train, batch_size=128, nb_epoch=20, verbose=0, validation_data=(X_test, Y_
```