Keras Callbacks — Monitor and Improve Your Deep Learning

How to build deep learning models more quickly and efficiently using Keras callbacks.

Matthew Cloney Follow
Sep 6, 2018 · 5 min read

If you're building deep learning models and you're unable to troubleshoot them, you're going to drive yourself insane.

Using techniques like callbacks can mean the difference between spending your weekends at the beach and spending them in a straightjacket. If you're using Keras, you already appreciate how quickly you can build deep learning models, but if you're not using callbacks, you're not taking advantage of one of its greatest features.

Keras callbacks can help you fix bugs more quickly, and can help you build better models. They can help you visualize how your model's training is going, and can even help prevent overfitting by implementing early stopping or customizing the learning rate on each iteration.

Keras callbacks return information from a training algorithm while training is taking place. In a sense, they're similar to TensorFlow's `fetches` parameter when you call `tf.Session().run` . From the documentation:

> *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.*

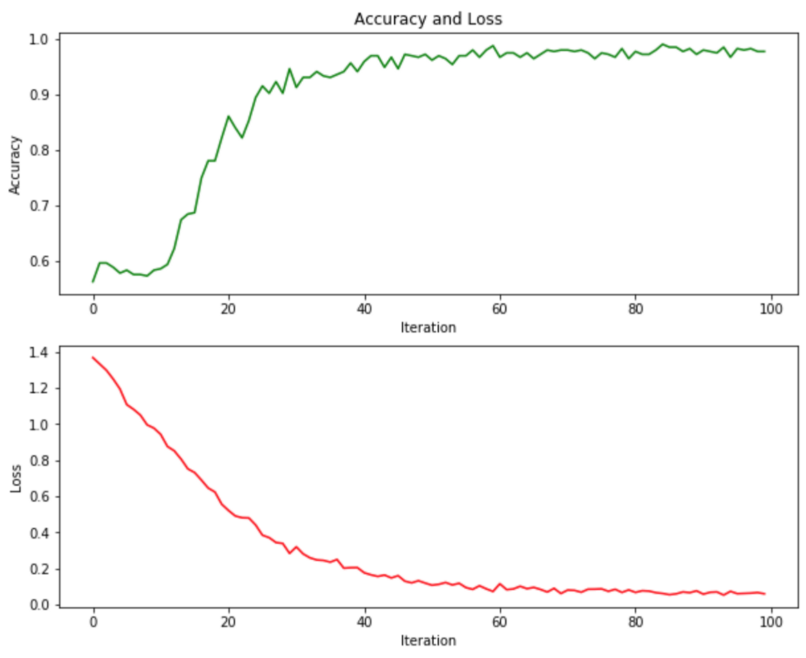
Note a couple of things here. First, callbacks are functions, which implies that you can roll your own if need be. Second, you can use more than one callback to monitor or affect the training of your model. Here's a brief review of what I believe are the most useful ones.

BaseLogger and History

You get these without doing anything special—they're applied to the model by default. By convention, most of the time you'll just call `model.fit()` on a model object without assigning the result to anything. Instead of calling `model.fit()`, you can write `my_history = model.fit()`. The `my_history` variable is assigned a `keras.callbacks.History` object. The `history` property of this object is a dict with average accuracy and average loss information for each epoch. You can also inspect the `params` property, which is a dict of the parameters used to fit the model.

ModelCheckpoint

This callback will save your model as a checkpoint file (in `hdf5` format) to disk after each successful epoch. You can actually set the output file to be dynamically named based on the epoch. You can also write either the loss value or accuracy value as part of the log's file name. This is especially handy for models where epochs take an extremely long time, or if you're running them on AWS spot instances and the spot instance price rises above your maximum bid.



Plot of accuracy (top) and loss (bottom) over 100 epochs training an RNN

CSVLogger

The CSVLogger is pretty self-explanatory. It writes a CSV log file containing information about epochs, accuracy, and loss to disk so you can inspect it later. It's great if you want to roll your own graphs or keep a record of your model training process over time. The graph above was created from one of these files.

EarlyStopping

One technique to reduce overfitting in neural networks is to use early stopping. Early stopping prevents overtraining of your model by terminating the training process if it's not really learning anything. This is pretty flexible—you can control what metric to monitor, how much it needs to change to be considered "still learning", and how many epochs in a row it can falter before the model stops training.

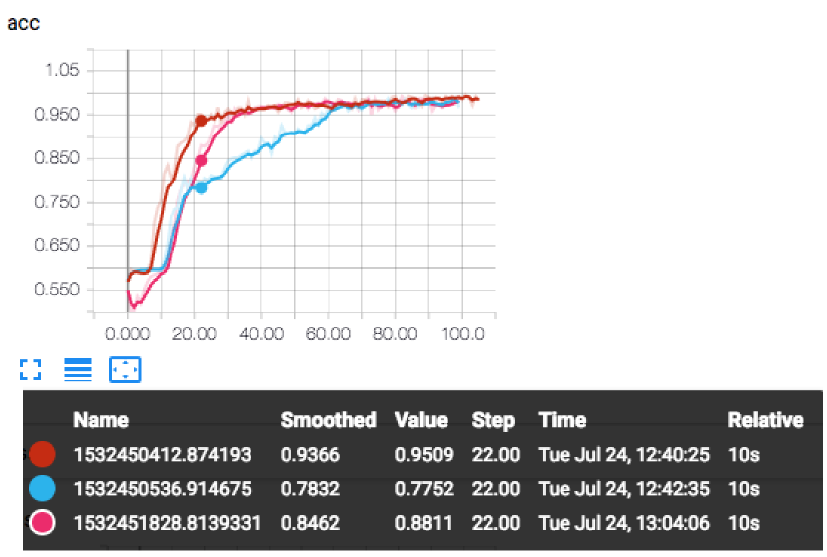
RemoteMonitor

This callback sends JSON status messages via an HTTP POST method. This could be easily integrated with a pub/sub messaging service like Kafka or a queue like Amazon SQS. For instance, you could setup a lightweight HTTP endpoint and route any POSTed JSON as a payload to an SQS queue. You could then have another process monitor the queue to kick off other processes or to handle specific events.

LearningRateScheduler

Another optimization technique with deep learning is to adjust the learning rate over time. The learning rate determines the size of the steps taken during the gradient descent process.

One method is to start with a relatively large value, and decrease it in later training epochs. All you need to do is write a simple function that returns the desired learning rate based on the current epoch, and pass it as the `schedule` parameter to this callback.



	Name	Smoothed	Value	Step	Time	Relative
●	1532450412.874193	0.9366	0.9509	22.00	Tue Jul 24, 12:40:25	10s
●	1532450536.914675	0.7832	0.7752	22.00	Tue Jul 24, 12:42:35	10s
●	1532451828.8139331	0.8462	0.8811	22.00	Tue Jul 24, 13:04:06	10s

TensorBoard visualization of accuracy by training epoch for three separate training runs of a recurrent neural network

TensorBoard

This is probably the coolest of all the out-of-the-box callbacks. By using a TensorBoard callback, logs will be written to a directory that you can then examine with TensorFlow's excellent TensorBoard visualization tool. It even works (to an extent) if you're using a backend other than TensorFlow, like Theano, or CNTK (if you're a glutton for punishment).

LambdaCallback

If none of the techniques above fit what you're trying to achieve, you can create your own callback. If you just want to create a basic one, you can use LambdaCallback. This allows you to trigger events when an epoch, batch, or training process begins or ends.

If you want to create something a little more complex, you can create your own callback by inheriting from the `keras.callbacks.Callback` class. This is only for those who truly want low-level control over how a callback is executed. You may also want to create your own callback if you're implementing some custom logic that's not available in the existing callbacks.

It's a good idea to use more than one of these callbacks. I use **TensorBoard** at virtually every phase of the modeling process, and **ModelCheckpoint** and **RemoteMonitor** for long-running processes. Callbacks like **EarlyStopping** and **LearningRateScheduler** are great for optimization. This isn't an exhaustive list, but for me, they're the most useful.

In my next article, I'll share how to create a custom Keras callback to publish to Amazon's Simple Notification Services (SNS).