Advanced Keras — Accurately Resuming a Training Process

Handling nontrivial cases where custom callbacks are used

Eyal Zakkay Following
Jan 2 · 3 min read

Photo Credit: Eyal Zakkay

. . .

In this post I will present a use case of the Keras API in which resuming a training process from a loaded checkpoint needs to be handled differently than usual.

TL;DR — If you are using custom callbacks which have internal variables that change during a training process, you need to address this when resuming by initializing these callbacks differently.

When training a Deep Learning model using Keras, we usually save checkpoints of that model's state so we could recover an interrupted training process and restart it from where we left off. Usually this is done with the ModelCheckpoint Callback. According to the documentation of Keras, a saved model (saved with `model.save(filepath)`) contains the following:

- The architecture of the model, allowing to re-create the model
- The weights of the model
- The training configuration (loss, optimizer)
- The state of the optimizer, **allowing to resume training exactly where you left off**.

In certain use cases, this last part isn't exactly true.

Example:

Let's say you are training a model with a custom learning rate scheduler callback, which updates the LR after **each batch**:

```python
class LearningRateSchedulerPerBatch(LearningRateScheduler):
    """ Callback class to modify the default learning rate scheduler to operate each batch"""
    def __init__(self, schedule, verbose=0):
        super(LearningRateSchedulerPerBatch, self).__init__(schedule, verbose)
        self.count = 0  # Global batch index (the regular batch argument refers to the batch ind

    def on_epoch_begin(self, epoch, logs=None):
        pass

    def on_epoch_end(self, epoch, logs=None):
        pass

    def on_batch_begin(self, batch, logs=None):
        super(LearningRateSchedulerPerBatch, self).on_epoch_begin(self.count, logs)

    def on_batch_end(self, batch, logs=None):
        super(LearningRateSchedulerPerBatch, self).on_epoch_end(self.count, logs)
        self.count += 1
```

LearningRateSchedulerPerBatch.py hosted with ♡ by GitHub view raw

The `counter` variable is initialized to zero when the callback is created and keeps up with the global batch index (the `batch` argument in `on_batch_end` holds the batch index within the *current* epoch).

Let's say we want to resume a training process from a checkpoint. The usual way would be:

```
# Initialize Callbacks:
ckpt_callback = ModelCheckpoint(filepath='weights.{epoch:02d}-{val_loss:.2f}.hdf5', monitor='val
lr_decay_callback = LearningRateSchedulerPerBatch(
                    lambda step: ((learning_rate - min_learning_rate) * decay_rate ** step + min
callbacks = [ckpt_callback, lr_decay_callback]

# Load checkpoint:
if checkpoint_path is not None:
    # Load model:
    model = load_model(checkpoint_path)
    # Finding the epoch index from which we are resuming
    initial_epoch = get_init_epoch(checkpoint_path)
else:
    model = build_model_func()
    initial_epoch = 0
# Start/resume training
model.fit(x, y, callbacks=callbacks, initial_epoch=initial_epoch)
```

resume_example.py hosted with ♡ by GitHub view raw

The wrong way to do it

Notice that the `LearningRateSchedulerPerBatch` callback is initialized with `counter=0` even when resuming. When training resumes **this will not recreate the same conditions that took place when the checkpoint was saved.** The learning rate would restart from its initial value and that is probably not what we would want.

The correct way to do it

We saw an example of how a wrong initialization of callbacks can lead to unwanted results when resuming. There are several ways to approach this, here I will describe two:

Solution 1: Updating the variables with correct values

When dealing with simple cases, in which the callback has only a handful of updating variables, it is rather simple to overwrite the values of these variables before resuming. In our example, if we would want to resume with the correct value of `count` we would do the following:

```python
# Initialize Callbacks:
ckpt_callback = ModelCheckpoint(filepath='weights.{epoch:02d}-{val_loss:.2f}.hdf5', monitor='val
lr_decay_callback = LearningRateSchedulerPerBatch(
                    lambda step: ((learning_rate - min_learning_rate) * decay_rate ** step + min
callbacks = [ckpt_callback, lr_decay_callback]

# Load checkpoint:
if checkpoint_path is not None:
    # Load model:
    model = load_model(checkpoint_path)
    # Finding the epoch index from which we are resuming
    initial_epoch = get_init_epoch(checkpoint_path)
    # Calculating the correct value of count
    count = initial_epoch*batches_per_epoch
    # Update the value of count in callback instance
    callbacks[1].count = count
else:
    model = build_model_func()
    initial_epoch = 0
# Start/resume training
model.fit(x, y, callbacks=callbacks, initial_epoch=initial_epoch)
```

resume_example2.py hosted with ♡ by GitHub view raw

Solution 2: Saving and loading callbacks with Pickle

When our custom callbacks have many updating variables or include complex behaviors, safely overwriting each variable might be difficult. An alternative solution is to pickle the callback instance every time we save a checkpoint, then we can load this pickle when resuming and reconstruct the original callback with all its correct values.

Note: in order to pickle your callbacks they must not hold any non-serializable elements. Also, in Keras versions <2.2.3 the model itself was not serializable. This prevented the pickling of any callback, since each callback also holds a reference to the model.

In this case, resuming will look something like this:

```python
# Initialize Callbacks:
lr_decay_callback = LearningRateSchedulerPerBatch(
                    lambda step: ((learning_rate - min_learning_rate) * decay_rate ** step + min
# Modified Checkpoint Callback
ckpt_callback = ModelCheckpointEnhanced(filepath='weights.{epoch:02d}-{val_loss:.2f}.hdf5', moni
                                        callbacks_to_save=lr_decay_callback ,callbacks_filepath='LRcallb

callbacks = [ckpt_callback, lr_decay_callback]

# Load checkpoint:
if checkpoint_path is not None:
    # Load model:
    model = load_model(checkpoint_path)
    # Finding the epoch index from which we are resuming
    initial_epoch = get_init_epoch(checkpoint_path)
    # loading the callback from pickle
    loaded_callback = pickle.load(open( callback_checkpoint_path, "rb" ))
    # Update the callback instance
    callbacks[1] = loaded_callback
else:
    model = build_model_func()
    initial_epoch = 0
# Start/resume training
model.fit(x, y, callbacks=callbacks, initial_epoch=initial_epoch)
```

resume_example3.py hosted with ♡ by GitHub view raw

You might have noticed that I have used a modified checkpoint callback called `ModelCheckpointEnhanced`. This is because using the pickling method means that we also need the `ModelCheckpoint` callback to save pickles of the relevant callbacks. An example implementation of such modified callback might look something like this:

```python
from keras.callbacks import ModelCheckpoint
import pickle

class ModelCheckpointEnhanced(ModelCheckpoint):
    def __init__(self, *args, **kwargs):
        # Added arguments
        self.callbacks_to_save = kwargs.pop('callbacks_to_save')
        self.callbacks_filepath = kwargs.pop('callbacks_filepath')
        super().__init__(*args, **kwargs)

    def on_epoch_end(self, epoch, logs=None):
        # Run normal flow:
        super().on_epoch_end(epoch,logs)

        # If a checkpoint was saved, save also the callback
        filepath = self.callbacks_filepath.format(epoch=epoch + 1, **logs)
        if self.epochs_since_last_save == 0 and epoch!=0:
            if self.save_best_only:
                current = logs.get(self.monitor)
                if current == self.best:
                    # Note, there might be some cases where the last statement will save on unwa
                    # However, in the usual case where your monitoring value space is continuous
                    with open(filepath, "wb") as f:
                        pickle.dump(self.callbacks_to_save, f)
            else:
                with open(filepath, "wb") as f:
                    pickle.dump(self.callbacks_to_save, f)
```

resume_example4.py hosted with ♡ by GitHub view raw

The example above handles the case where you need to pickle only one callback, if you have multiple callbacks to save you will need to perform some minor modifications.

Summary

We saw how in some cases taking the naive approach for resuming a training process can lead to unwanted results. We saw examples of two ways to handle such cases in order to get consistent results when restarting an interrupted training process.

The examples I used are taken from my Keras implementation of the Sketch-RNN algorithm, a sequence to sequence Variational Autoencoder model for generation of sketches.

Keras Deep Learning API Tutorial