Understanding callbacks in fastai

Mar 29, 2019
deep learning fastai
12 min read

Table of Content

What's a callback?
Callbacks in fastai
Callbacks in the training loop
Examples of fastai callbacks and how they work
Gradient clipping
Early stopping
Conclusion

fastai is a great library for Deep Learning with many powerful features, which make it very easy to quickly build state of the art models, but also to tweak them as you wish. One of the best features of fastai is its callbacks system that lets you customize simply pretty much everything.

However, it can take getting used to and that's the purpose of this post: presenting the callback system in fastai, explaining how it works and how to use it and finally showing you a few examples.

This post is mainly based on Sylvain Gugger's talk, and the fast.ai part 2 v3 course and forums (not yet opened to everyone, but they should be in a few months).

What's a callback?

A callback is a fancy name for a function. Nothing more. It's important to understand that under that unfamiliar term hides a very familiar concept, at least if you've been programming for a bit.

Now obviously it's a special type of function, otherwise it wouldn't have that special name. The main difference between callbacks and other "regular" functions is in how you use it. Callbacks are common in a lot of languages and libraries/framework, and even if there might be some differences it's always the same concept.

A callback is a function that you write *in case something happens*. When that something happens, you call back (hence the name) that function to do something.

One language that uses a lot of callbacks is JavaScript, the web scripting language. Let's further understand what a callback is with an example from JavaScript.

Say you have a webpage with a button on it. Obviously, you want something to be done when the user clicks on the button, but only if and when the user clicks on the button. So you write a function, let's call it `on_button_click()` that does what needs to be done when the user clicks on

the button. You then specify elsewhere in the code that if the button is ever clicked on you have to use (call back!) the `on_button_click()` function to do whatever. Because the `on_button_click()` function is only called (back) when something happens (a button click), it's called a callback.

See? In the end, the concept of callback isn't that hard, but callbacks can also get quite tricky quickly. So please keep at it even if you struggle when implementing them!

Callbacks in fastai

In fastai, callbacks are used to customize the training loop. We'll see how they work in detail in the next section, but here's the gist of it: every main step in the fastai training loop (the one that computes the forward loop, computes the loss with the optimizer, does the backward pass, ...) is interleaved with calls to callbacks.

For example, if you want to do something just after the forward pass, you write a function (a callback) that does the thing you want. You tell fastai (we'll see how) that this function is to be executed at the end of the forward pass, and when the training loop comes to that point it will remember that you wanted a function called (back) at that point in time.

As callbacks have access to and can modify pretty much everything, the fastai callback system allows for an "infinitely customizable training loop" (to quote the title of Sylvain Gugger's talk).

That way, you can easily implement technics like Gradient Clipping, One Cycle training Schedule, and many others.

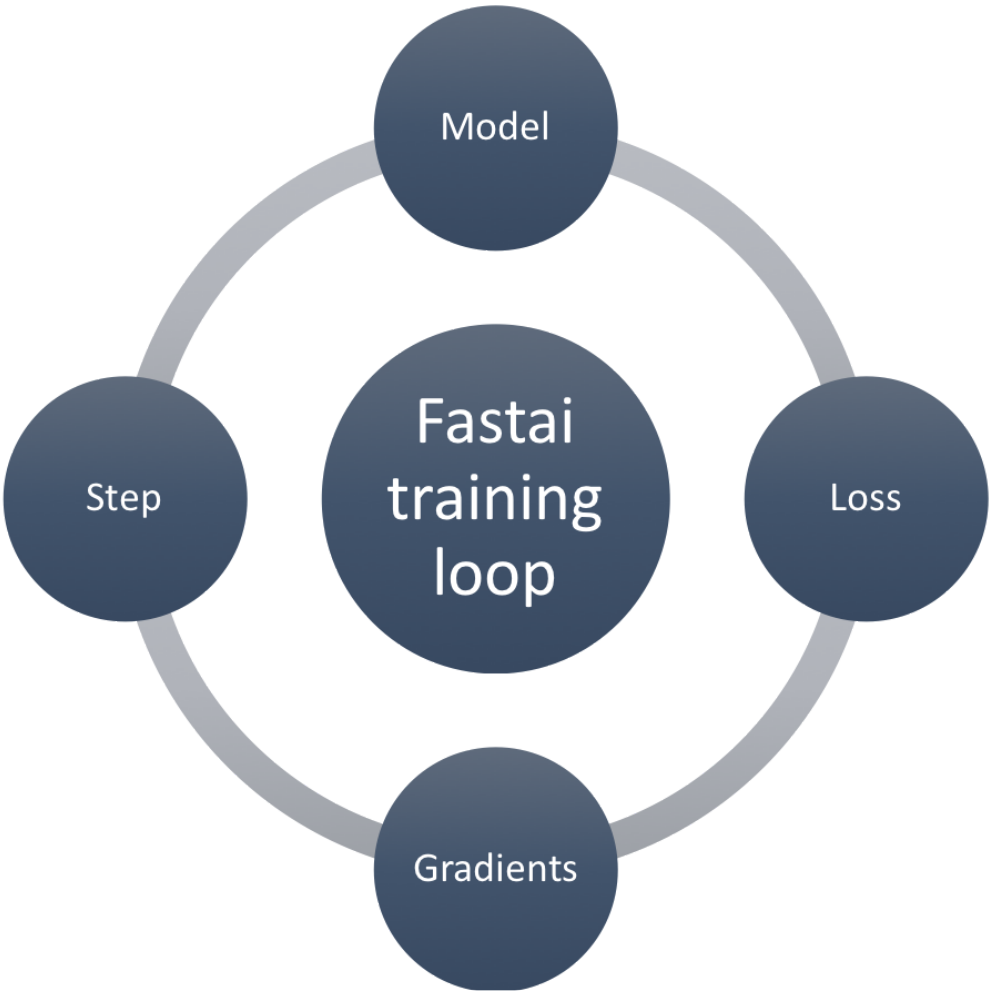
Let's see precisely how all this works in fastai.

Callbacks in the training loop

Let's have a look at the basic fastai / PyTorch training loop first:

```
def train(train_dl, model, epoch, opt, loss_func):
  for _ in range(epoch):
    model.train() # model in training mode
    for xb,yb in train_dl: # loop through the batches
      out = model(xb) # forward pass
      loss = loss_func(out, yb) # compute the loss
      loss.backward() # backward pass
      opt.step()
      opt.zero_grad()
```

There are four main steps, shown in the following image:



fastai training loop, taken from Sylvain's talk
The forward pass is computed between the "model" and the "loss", and the backward pass is computed between the "gradients" and the "step"

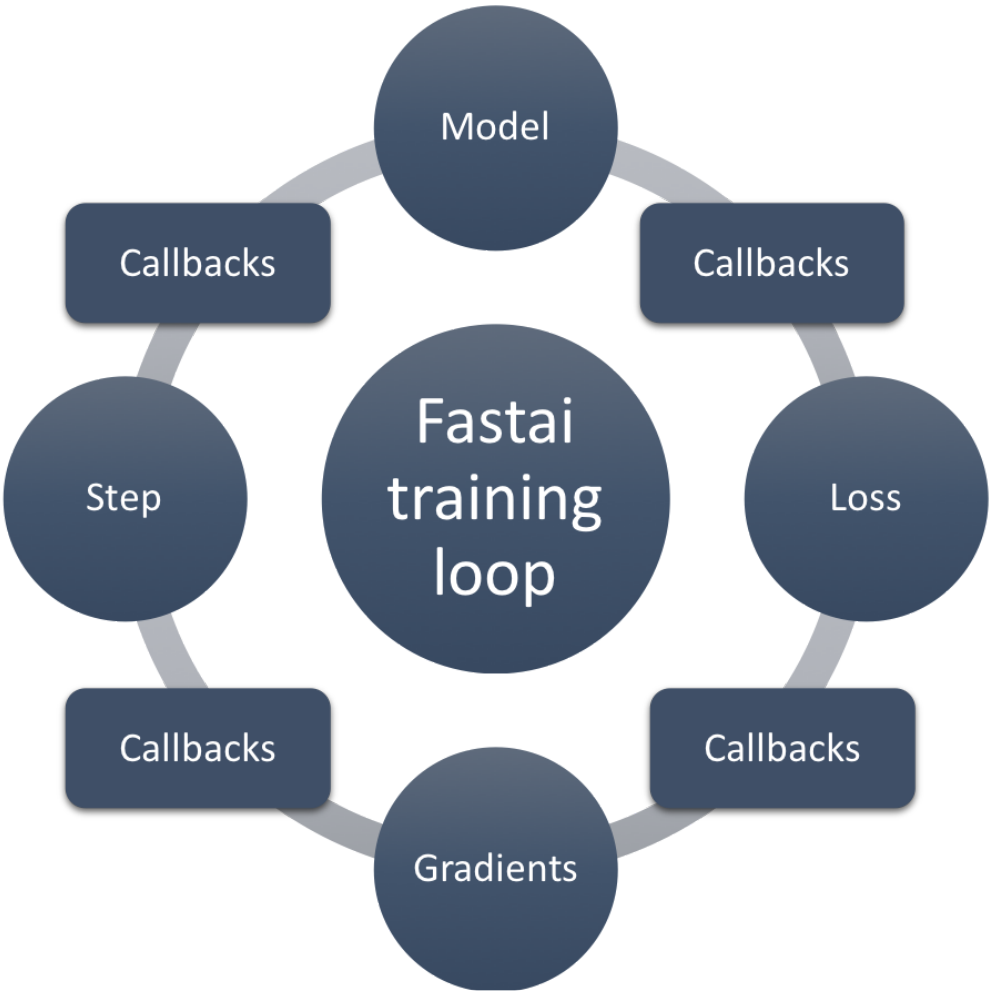
In order to be able to use functions in between all of those steps, you simply have to add a list `callbacks` of functions that we will call at every step in case there's something to do. Something like this:

```
def train(train_dl, model, epoch, opt, loss_func, callbacks):
  callbacks.on_train_begin()

  for _ in range(epoch):
    callbacks.on_epoch_begin()
    model.train() # model in training mode

    for xb,yb in train_dl: # loop through the batches
      callbacks.on_batch_begin()
      out = model(xb) # forward pass

      callbacks.on_loss_begin()
      loss = loss_func(out, yb) # compute the loss
```

```
        callbacks.on_backward_begin()
        loss.backward() # backward pass
        callbacks.on_backward_end()

        opt.step()
        callbacks.on_step_end()

        opt.zero_grad()
        callbacks.on_batch_end()

    callbacks.on_epoch_end()

callbacks.on_train_end()
```

You can see that now we can modify our training loop pretty much however we want, at every step. We just added callbacks everywhere.

And the names are pretty self-explanatory: `callbacks.on_epoch_begin()` simply means “Hey there callback functions, I’m beginning a new epoch. Does someone here want to do something?” and `callbacks.on_step_end()` means “Hey, I’ve just taken an optimizer step and I’m about to zero the gradients. Anything to do now?”.



fastai training loop with callbacks, taken from Sylvain's talk

Can we really do whatever we want? Maybe you've noticed something missing: how are the callbacks supposed to do anything without any access to the state of the training? You're right, something is missing, and that's why fastai created a `CallbackHandler` that takes care of taking the relevant data and transmitting it to the callbacks. Also, it returns values to modify the behaviour of the training loop.

For instance, on the beginning of a new batch the training data `xb` and the targets `yb` for the batch will be passed to the `CallbackHandler` through `on_batch_begin(xb,yb)`. If some callbacks want to do something with `xb` and `yb` they can, and they can even return new `xb` and `yb` that will be used for next steps of the batch. Here's what that line would look like, with a `CallbackHandler` named `cb_handler`:

```
xb,yb = cb_handler.on_batch_begin(xb,yb)
```

Another example: say you want to do something just after you've calculated the loss. For instance, say you want to skip the backward pass if the loss is too high. And maybe also if the loss is too small you want to scale it up. Well easy: just pass the `loss` value to the `CallbackHandler` right after the loss is computed. In turn, the `CallbackHandler` will spit out a new loss (maybe unchanged) and a flag `skip_backward` that tells whether or not to skip the backward pass, like so:

```
loss, skip_backward = cb_handler.on_backward_begin(loss)
```

And then you only do the backward pass if `skip_backward` is `False`:

```
if not skip_backward: loss.backward()
```

Also, in fastai, the model, the data, the loss function and the optimizer are wrapped in a `Learner` class.

All in all, the full training loop looks like this now:

```
def train(learn, epochs, callbacks, metrics):
  cb_handler = CallbackHandler(callbacks)
  cb_handler.on_train_begin(epochs, learn, metrics)

  for epoch in range(epochs):
    learn.model.train() # model in training mode
    cb_handler.on_epoch_begin(epoch)

    for xb,yb in train_dl: # loop through the batches
      xb,yb = cb_handler.on_batch_begin(xb,yb)

      out = learn.model(xb) # forward pass
      out = cb_handler.on_loss_begin(out)

      loss = learn.loss_func(out, yb) # compute the loss

      loss, skip_backward = cb_handler.on_backward_begin(loss)
      if not skip_backward: loss.backward() # backward pass
      if not cb_handler.on_backward_end(): learn.opt.step()

      if not cb_handler.on_step_end(): learn.opt.zero_grad()
      if not cb_handler.on_batch_end(): break

    val_loss, mets = validate(learn.data.valid_dl, model, metrics)
    if not cb_handler.on_epoch_end(val_loss, mets): break

  cb_handler.on_train_end()
```

The training loop is much more heavy than it was before, and maybe there's some stuff where you're not too sure what it's exactly about. That doesn't matter too much (there's always the documentation) as long as you understand that **there's every callback call you need, at every**

step of the training loop, with the right arguments and the right power over the training loop.

If you ever need to find out what a specific callback call you want is about, what arguments it has and what it needs to return, go back to the documentation and you'll find what you need.

Examples of fastai callbacks and how they work

Hopefully you're now getting what a callback is and where the callbacks are used in the fastai training loop. Let's see a few useful examples of callbacks already implemented in fastai.

Gradient clipping

One recurrent issue of deep neural networks is the vanishing or the exploding of the gradient. A good initialization can go a long way towards minimizing that phenomenon (see my post about Xavier and Kaiming initialization). Another often used technique is gradient clipping, in which you simply don't allow the gradient to get too big; if it does, you "clip" it.

Gradient Clipping is implemented as a simple callback in the fastai library. Here is the full code:

```
class GradientClipping(LearnerCallback):
    "Gradient clipping during training."
    def __init__(self, learn:Learner, clip:float = 0.):
        super().__init__(learn)
        self.clip = clip

    def on_backward_end(self, **kwargs):
        "Clip the gradient before the optimizer step."
        if self.clip: nn.utils.clip_grad_norm_(self.learn.model.parameters(), self.clip)
```

You can see how it works: in the initialization you simply tell `GradientClipping` where you want to clip the gradient, and at the end of the backward pass you clip them (with a function I won't get into but you get the idea).

Perhaps you've noticed that `Gradient Clipping` is a class, whereas I told you callbacks were functions. Did I lie? Well, not really. Callbacks are functions, but you can also implement them as objects to get the advantages of Object Oriented programming (also, in Python functions are objects so I didn't lie at all!). All fastai callbacks inherit from the `Callback` class that implements dummy functions I talked about above (e.g. `on_epoch_begin()`, `on_backward_end()`, ...). When you want to implement a callback yourself you just subclass from `Callback` and redefine one of those functions. Here's the (nearly complete) definition:

```
class Callback():
    "Base class for callbacks that want to record values, dynamically change learner params, etc."
    _order=0
    def on_train_begin(self, **kwargs:Any)->None:
        "To initialize constants in the callback."
        pass
    def on_epoch_begin(self, **kwargs:Any)->None:
        "At the beginning of each epoch."
        pass
    def on_batch_begin(self, **kwargs:Any)->None:
        "Set HP before the output and loss are computed."
        pass
    def on_loss_begin(self, **kwargs:Any)->None:
        "Called after forward pass but before loss has been computed."
        pass
    def on_backward_begin(self, **kwargs:Any)->None:
        "Called after the forward pass and the loss has been computed, but before backprop."
        pass
    def on_backward_end(self, **kwargs:Any)->None:
        "Called after backprop but before optimizer step. Useful for true weight decay in AdamW."
        pass
    def on_step_end(self, **kwargs:Any)->None:
        "Called after the step of the optimizer but before the gradients are zeroed."
        pass
    def on_batch_end(self, **kwargs:Any)->None:
        "Called at the end of the batch."
        pass
    def on_epoch_end(self, **kwargs:Any)->None:
        "Called at the end of an epoch."
        pass
    def on_train_end(self, **kwargs:Any)->None:
        "Useful for cleaning up things and saving files/models."
        pass
```

As you saw, `GradientClipping` inherited from the `LearnerCallback` class that itself is a wrapper around the `Callback` class that adds some functionalities, like a reference to the `Learner`.

Let's see another example.

Early stopping

"Early stopping" is when you want to stop the training when something happens. In the following code there's some code omitted (namely the superclass `TrackerCallback` that implements the `get_monitor_value` method, to retrieve the monitored value) but everything needed to understand the callback is present:

```
class EarlyStoppingCallback(TrackerCallback):
    "A `TrackerCallback` that terminates training when monitored quantity stops improving."
    def __init__(self, learn:Learner, monitor:str='val_loss', mode:str='auto', min_delta:int=0, pa
        super().__init__(learn, monitor=monitor, mode=mode)
        self.min_delta,self.patience = min_delta,patience
        if self.operator == np.less:  self.min_delta *= -1

    def on_train_begin(self, **kwargs:Any)->None:
        "Initialize inner arguments."
        self.wait = 0
        super().on_train_begin(**kwargs)

    def on_epoch_end(self, epoch, **kwargs:Any)->None:
        "Compare the value monitored to its best score and maybe stop training."
        current = self.get_monitor_value()
        if current is None: return
        if self.operator(current - self.min_delta, self.best):
            self.best,self.wait = current,0
        else:
            self.wait += 1
            if self.wait > self.patience:
                print(f'Epoch {epoch}: early stopping')
                return {"stop_training":True}
```

As you can see, after the initialization that sets some values it's quite straightforward: on the beginning of the training you set the `wait` variable to 0. That variable will serve as a timer: if you've waited long enough, as defined by `patience`, you stop the training. Then, and the end of each epoch, you see if the monitored value is 'better' (whatever this can mean in your particular case: bigger, smaller?) than what you had before, and you update the storing variable `current` if so. You also reset `wait` back to 0. Otherwise, you increment `wait`, and if it exceeds the predefined `patience` you stop the training by returning `{"stop_training":True}` that will be interpreted by the `CallbackHandler` and stop the training.

Conclusion

I hope you can see now how powerful the callback system is in fastai. And there's plenty more! All of the following (state of the art) techniques can be (and are) implemented in fastai with callbacks:

- Learning rate finder: find the best learning rate by plotting the learning rate against the loss for a particular `Learner`.
- MixUp data augmentation: a technique of data augmentation for images that consists in mixing two images together and making the model predict the mix. For example, you can blend a cat image and a dog image (30% cat, 70% dog) and the model should predict 30% cat and 70& dog. It works surprisingly well!
- One Cycle training Schedule: schedule learning rate and momentum to start low, go up, and then back down over several epochs. It is one of the techniques to achieve super-convergence, and it is incredibly powerful.
- MixedPrecision training: on modern NVIDIA cards (the ones with tensor cores) you can do computations with half precision (16 bits instead of 32), which can considerably speed up the training of your model. However, you still need the full 32 bits precision for some of the computations, like the loss. So you need to switch between fp16 (float precision 16) and fp32 (float precision 32) during the different phases of the training.
- And many more! Please check the documentation of callbacks in fastai.

Once you've understood how callbacks work in fastai (and I hope I was clear enough so that you now do), you can also write your own callbacks! In fact I would encourage you to do so: the best learning is by doing. For instance, can you imagine how you would implement one of the above example? Try it!

I hope you liked my explanation of callbacks in fastai; feel free to reach out to me on Twitter or by email if you have any question or suggestion (or if you find a typo).

Special thanks to Sylvain Gugger: a lot of the inspiration, the code and the images of this post are taken from his talk. Also, thanks for being so awesome in the fastai community and helping everyone like you do!