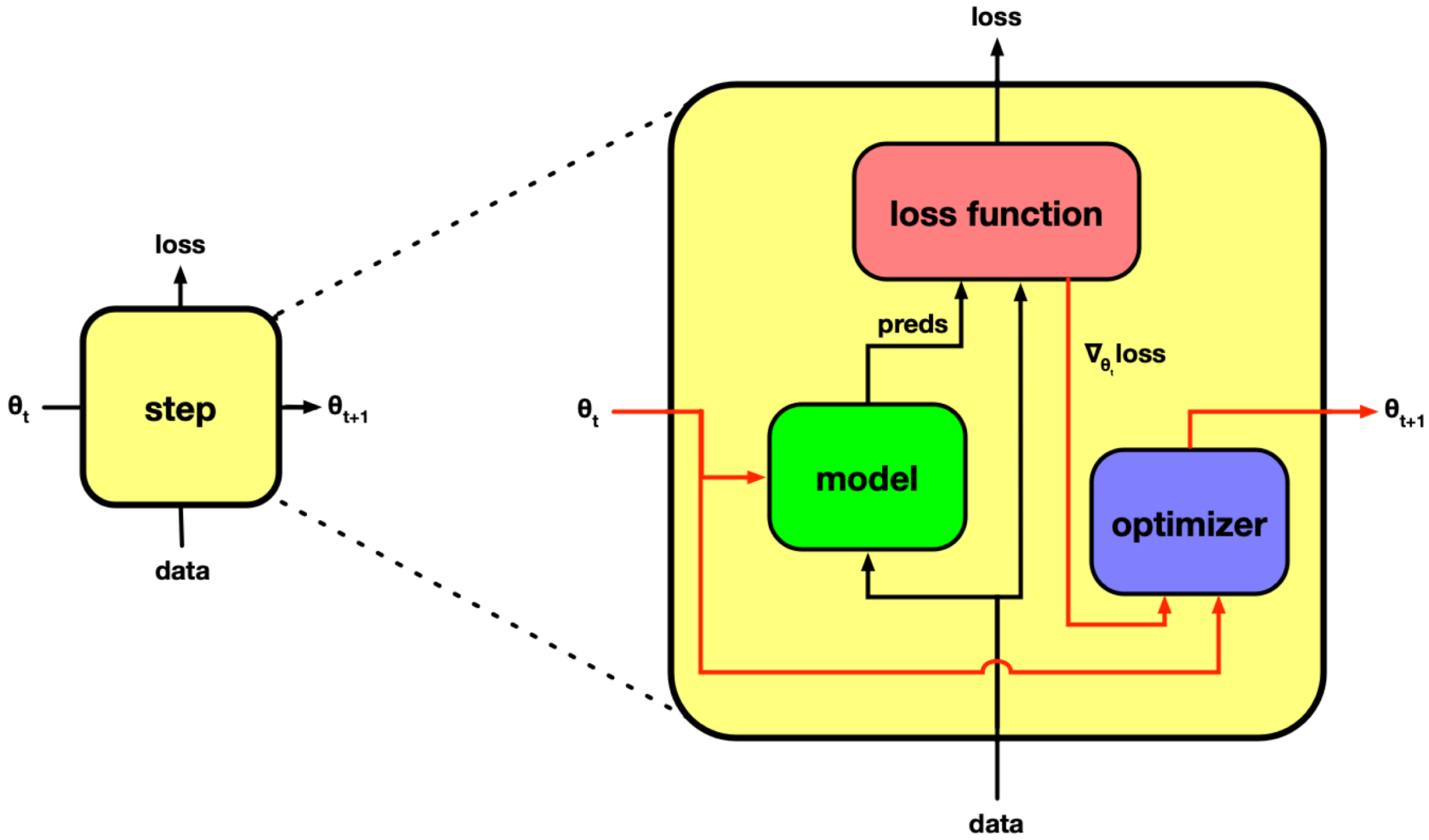
higher:
a pytorch metalearning library

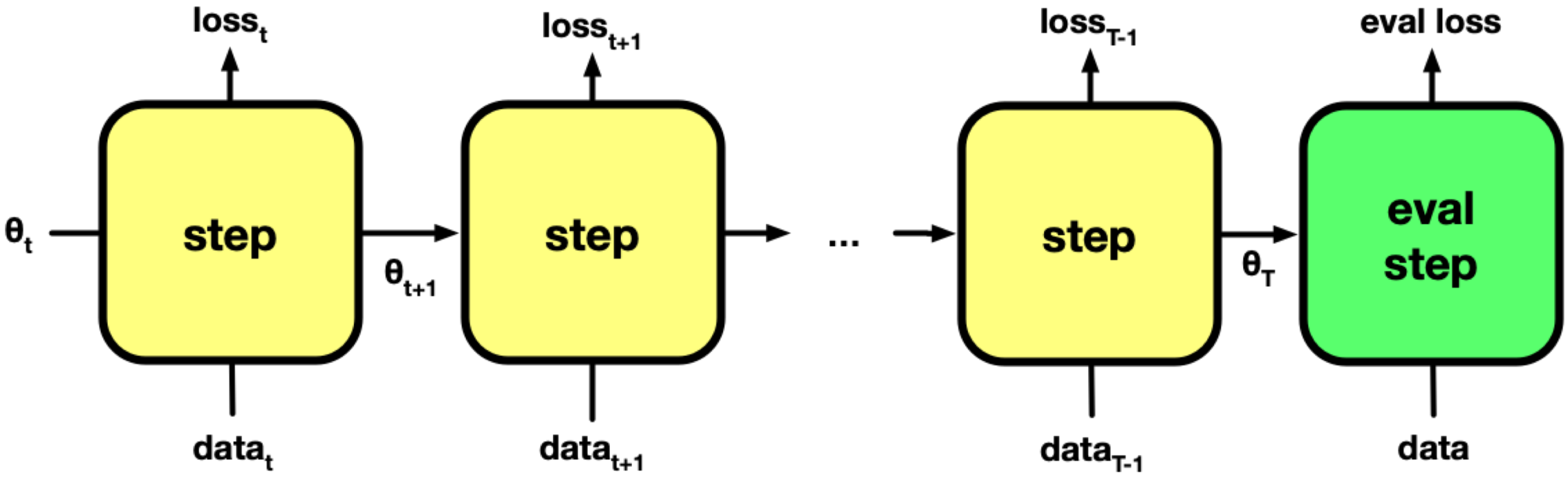
Edward Grefenstette

https://github.com/fairinternal/higher

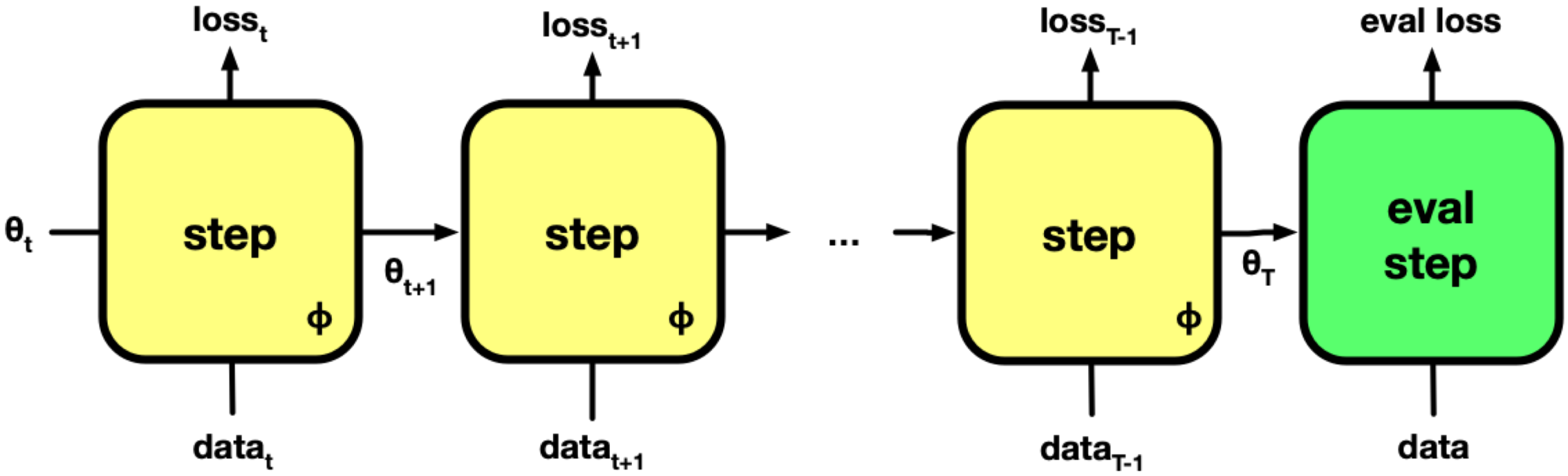
Anatomy of a training/eval step



An unrolled training + eval loop

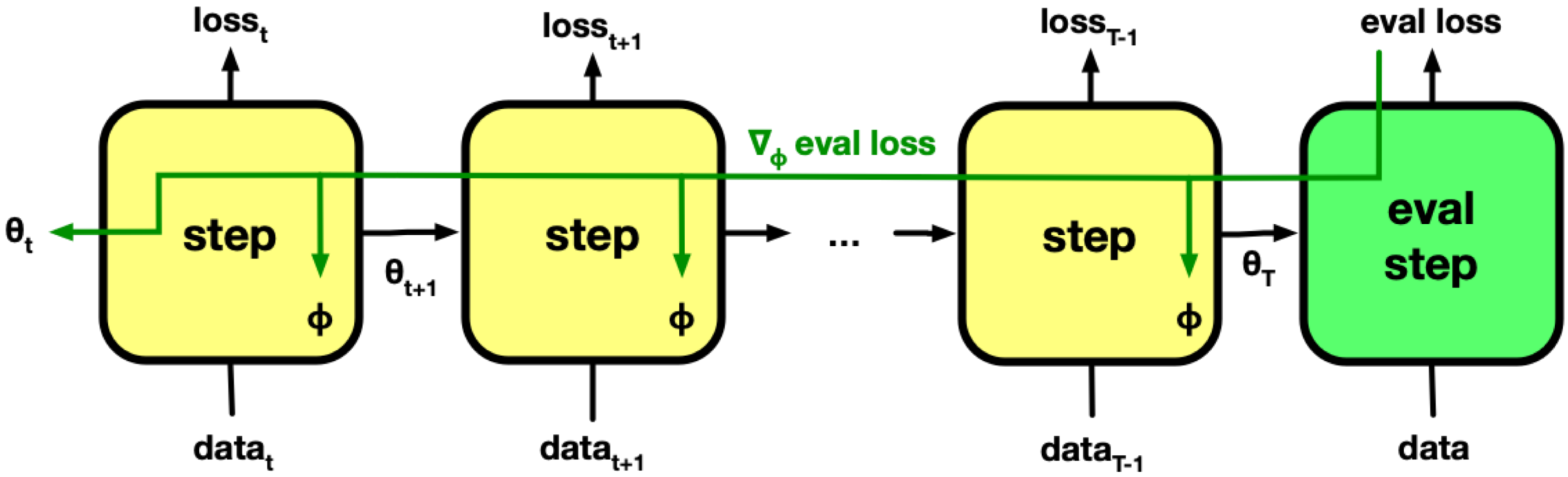


“Inner-loop” metalearning



φ: parameters corresponding to some aspect of your training loop (hyper-parameters, learned loss function, data mixture in multi-task learning, etc…)

“Inner-loop” metalearning



e.g. MAML == $\phi = \theta_t$

What you want to do

```
model = MyModel(…)  # arbitrarily complex model
opt = torch.optim.MyOptim(…)  # favourite optim
init_params = model.parameters()  # get view on initial model params
```

What you want to do

```
model = MyModel(…)  # arbitrarily complex model
opt = torch.optim.MyOptim(…)  # favourite optim
init_params = model.parameters()  # get view on initial model params

for inputs, labels in train_data:  # do some training
        opt.zero_grad()
        loss = loss_fn(model(inputs), labels, extra=phi)
        loss.backward()
        opt.step()
```

↑↑↑↑↑↑ == some additional parameters that form part of your training process, e.g. per-example importance weights, learned loss function, etc.

You might want to metalearn these, rather than optimize them against normal training loss...

What you want to do

```
model = MyModel(...)  # arbitrarily complex model
opt = torch.optim.MyOptim(...)  # favourite optim
init_params = model.parameters()  # get view on initial model params

for inputs, labels in train_data:  # do some training
        opt.zero_grad()
        loss = loss_fn(model(inputs), labels, extra=phi)
        loss.backward()
        opt.step()

meta_inputs, meta_labels = next(valid_data)  # get some non-training data
meta_loss = loss_fn(model(meta_inputs), meta_labels)  # measure loss
```

What you want to do

```
model = MyModel(...)  # arbitrarily complex model
opt = torch.optim.MyOptim(...)  # favourite optim
init_params = model.parameters()  # get view on initial model params

for inputs, labels in train_data:  # do some training
        opt.zero_grad()
        loss = loss_fn(model(inputs), labels, extra=phi)
        loss.backward()
        opt.step()

meta_inputs, meta_labels = next(valid_data)  # get some non-training data
meta_loss = loss_fn(model(meta_inputs), meta_labels)  # measure loss
meta_gradients = torch.autograd.grad(meta_loss, init_params)  # wrong grads!
```

↑↑↑↑↑↑ == $\nabla\theta_T$meta_loss != $\nabla\theta_t$meta_loss

↑↑↑↑↑↑ == model.parameters() # after training

What you want to do

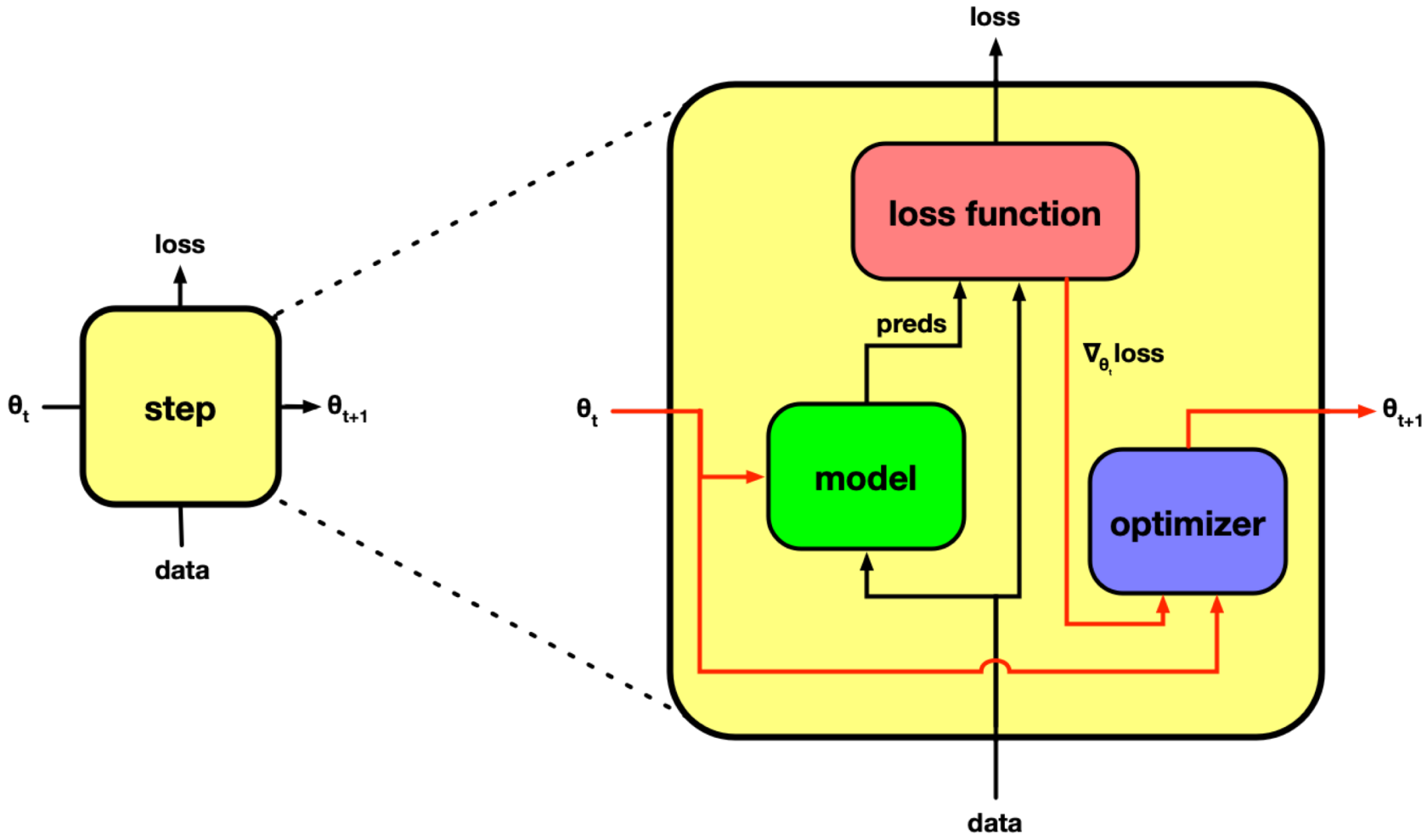
```
model = MyModel(...)  # arbitrarily complex model
opt = torch.optim.MyOptim(...)  # favourite optim
init_params = model.parameters()  # get view on initial model params

for inputs, labels in train_data:  # do some training
        opt.zero_grad()
        loss = loss_fn(model(inputs), labels, extra=phi)
        loss.backward()
        opt.step()

meta_inputs, meta_labels = next(valid_data)  # get some non-training data
meta_loss = loss_fn(model(meta_inputs), meta_labels)  # measure loss
meta_gradients = torch.autograd.grad(meta_loss, phi)  # also wrong grads!
```

↑↑↑↑↑↑ == only gradients w.r.t. phi at last timestep

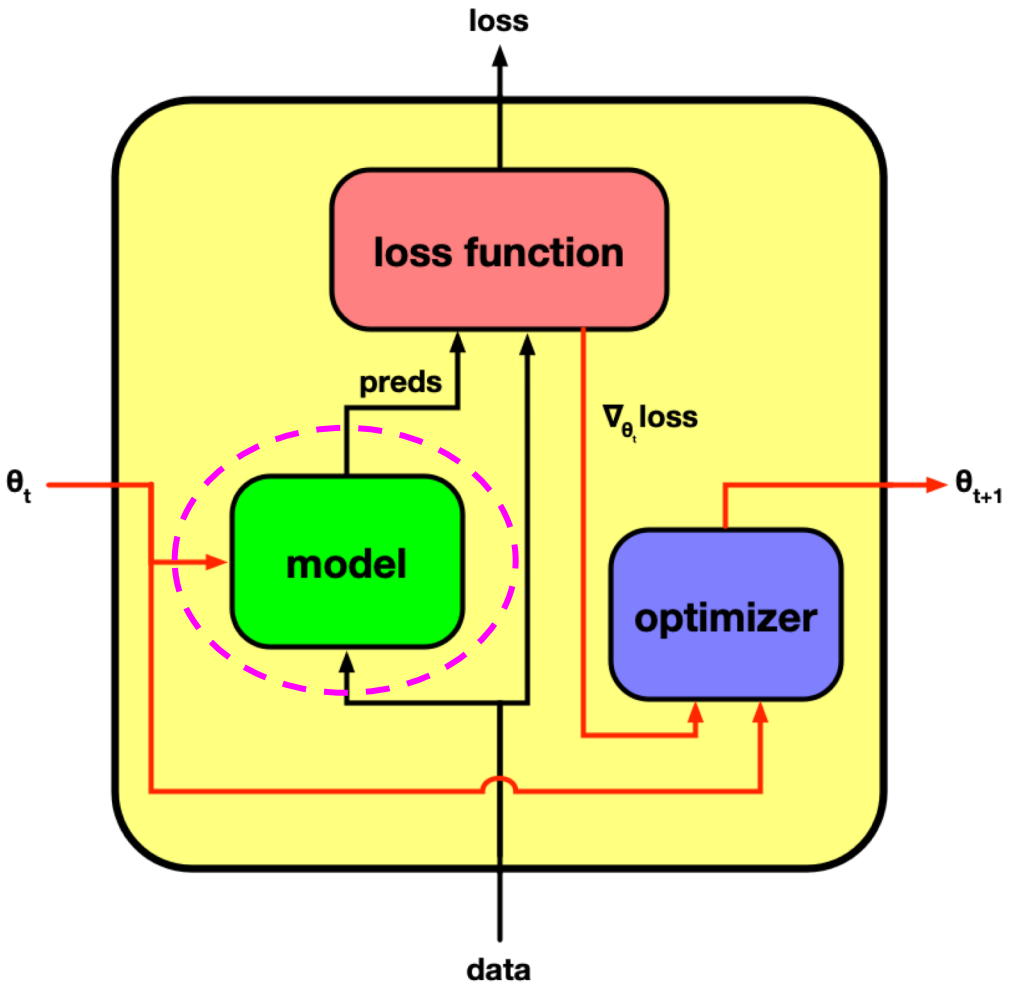
Why you can’t (easily) do it



Why you can't (easily) do it

We typically use **stateful** implementations of NN modules (torch.nn, tf.keras, etc).

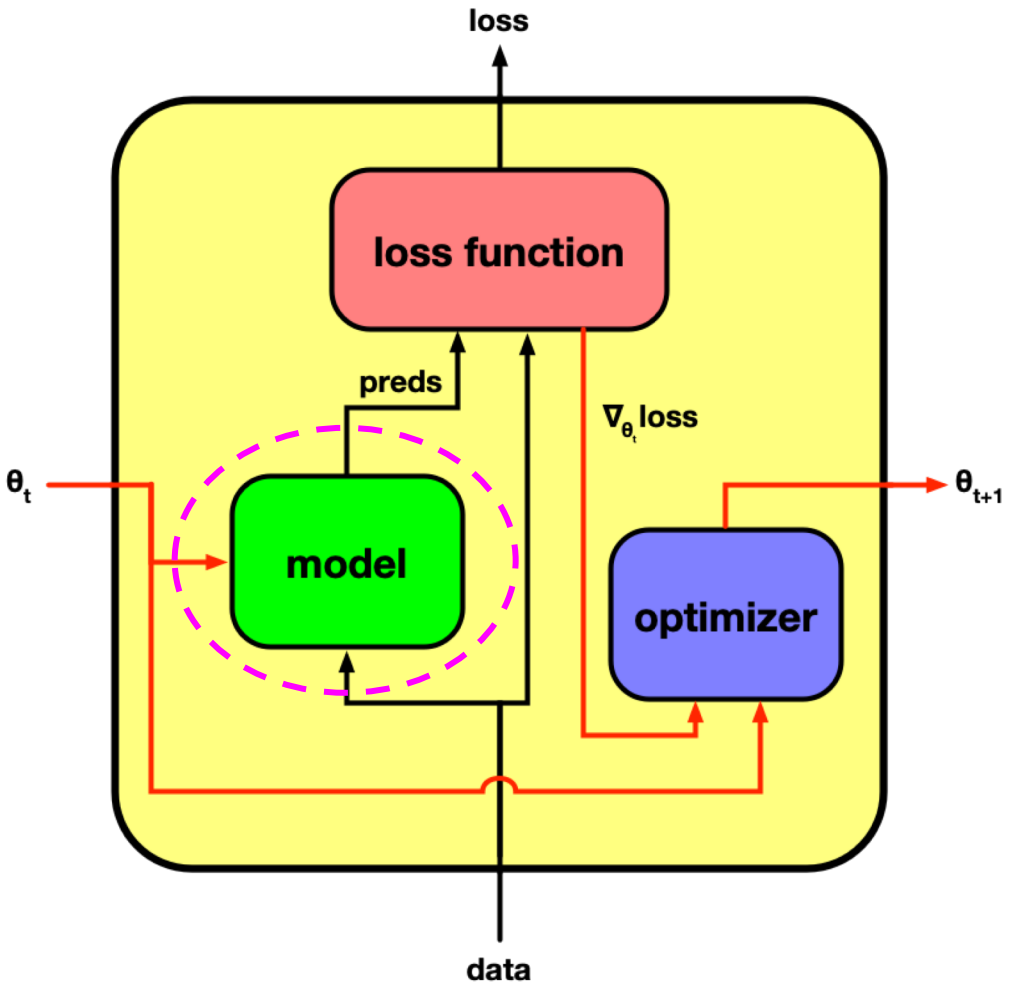
To track updates to params on the "gradient tape", we need to rewrite **functional** versions of our modules…



Why you can't (easily) do it

Re-implementing models functionally works but…

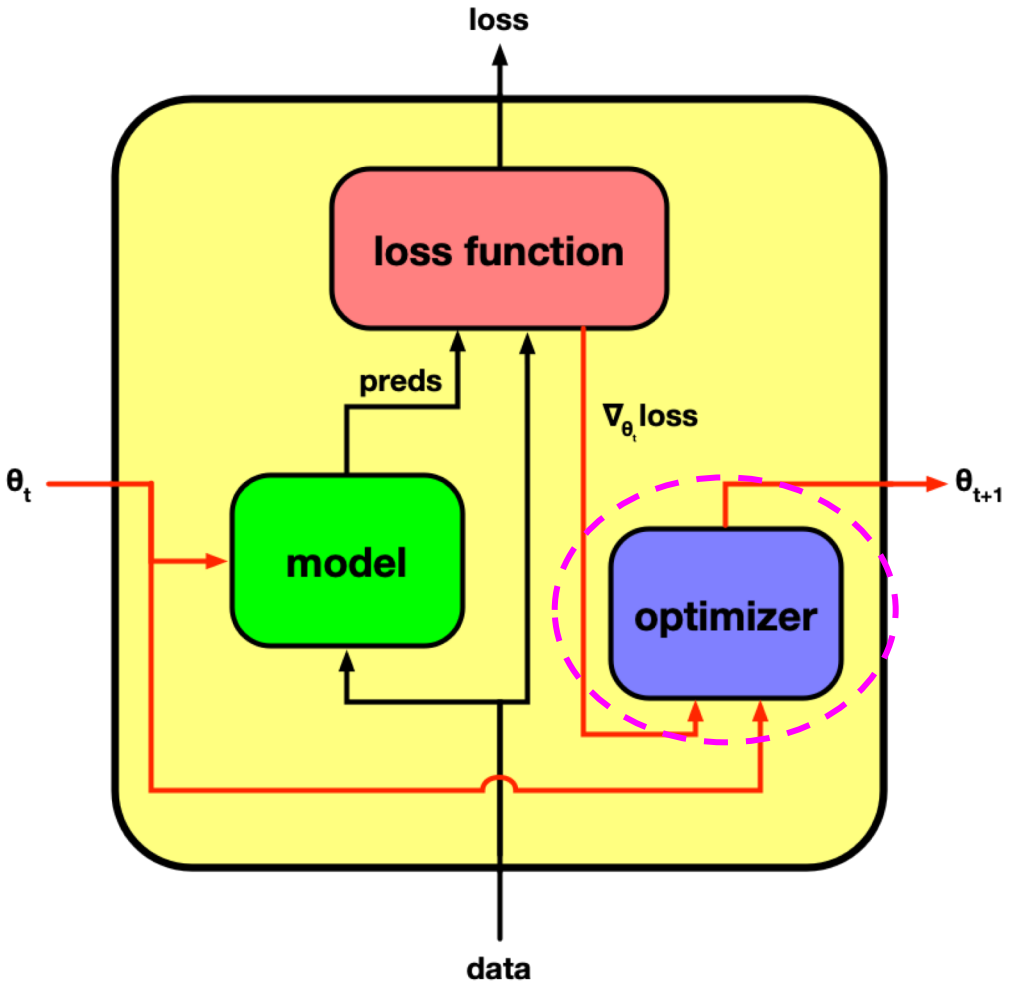
What if I want to metalearn with a pretrained BERT model, with saved weights, written by someone else, with a large and complicated codebase I don't want to mess with?



Why you can't (easily) do it

Optimizers apply grads to params **in place**, using ops that are differentiable, but **without tracking grad functions**.

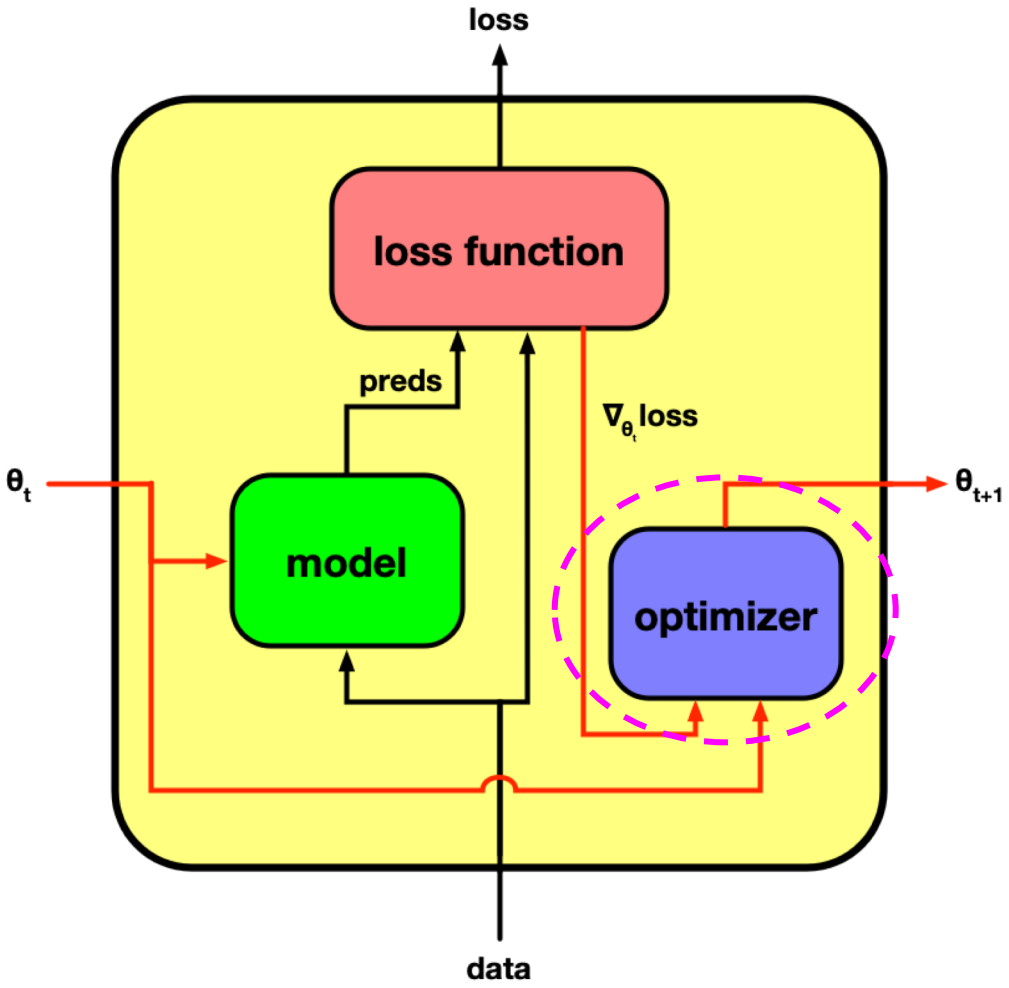
This is sensible to avoid **memory issues**, but we need to track these updates here…



Why you can't (easily) do it

Easy to write **vanilla SGD** in a differentiable way but…

What if I want to metalearn using a modern optimizer in my inner loop, or reflect my choice of outer loop optimizer?



What you want to do (redux)

```
model = MyModel(...)  # arbitrarily complex model
opt = torch.optim.MyOptim(...)  # favourite optim
init_params = model.parameters()  # get view on initial model params

for inputs, labels in train_data:  # do some training
        opt.zero_grad()
        loss = loss_fn(model(inputs), labels, extra=phi)
        loss.backward()
        opt.step()

meta_inputs, meta_labels = next(valid_data)  # get some non-training data
meta_loss = loss_fn(model(meta_inputs), meta_labels)  # measure loss
meta_gradients = torch.autograd.grad(meta_loss, init_params)  # wrong grads!
                                                              ↑↑↑↑↑↑ or w.r.t. phi
```

Just import higher!

```
model = MyModel(...)  # arbitrarily complex model
opt = torch.optim.MyOptim(...)  # favourite optim
init_params = model.parameters()  # get view on initial model params

for inputs, labels in train_data:  # do some training
        opt.zero_grad()
        loss = loss_fn(model(inputs), labels, extra=phi)
        loss.backward()
        opt.step()

meta_inputs, meta_labels = next(valid_data)  # get some non-training data
meta_loss = loss_fn(model(meta_inputs), meta_labels)  # measure loss
meta_gradients = torch.autograd.grad(meta_loss, init_params)  # wrong grads!
```

Just import higher!

```
import higher
model = MyModel(...)  # arbitrarily complex model
opt = torch.optim.MyOptim(...)  # favourite optim

for inputs, labels in train_data:  # do some training
        opt.zero_grad()
        loss = loss_fn(model(inputs), labels, extra=phi)
        loss.backward()
        opt.step()

meta_inputs, meta_labels = next(valid_data)  # get some non-training data
meta_loss = loss_fn(model(meta_inputs), meta_labels)  # measure loss
meta_gradients = torch.autograd.grad(meta_loss, init_params)  # wrong grads!
```

Just import higher!

```
import higher
model = MyModel(...)  # arbitrarily complex model
opt = torch.optim.MyOptim(...)  # favourite optim

for inputs, labels in train_data:  # do some training
        opt.zero_grad()
        loss = loss_fn(model(inputs), labels, extra=phi)
        loss.backward()
        opt.step()

meta_inputs, meta_labels = next(valid_data)  # get some non-training data
meta_loss = loss_fn(model(meta_inputs), meta_labels)  # measure loss
meta_gradients = torch.autograd.grad(meta_loss, init_params)  # wrong grads!
```

Just import higher!

```
import higher
model = MyModel(...)  # arbitrarily complex model
opt = torch.optim.MyOptim(...)  # favourite optim

with higher.innerloop_ctx(model, opt) as (fmodel, diffopt):
    for inputs, labels in train_data:  # do some training
        opt.zero_grad()
        loss = loss_fn(model(inputs), labels, extra=phi)
        loss.backward()
        opt.step()

    meta_inputs, meta_labels = next(valid_data)  # get some non-training data
    meta_loss = loss_fn(model(meta_inputs), meta_labels)  # measure loss
    meta_gradients = torch.autograd.grad(meta_loss, init_params)  # wrong grads!
```

Just import higher!

```
import higher
model = MyModel(...)  # arbitrarily complex model
opt = torch.optim.MyOptim(...)  # favourite optim

with higher.innerloop_ctx(model, opt) as (fmodel, diffopt):
    for inputs, labels in train_data:  # do some training
        opt.zero_grad()
        loss = loss_fn(model(inputs), labels, extra=phi)
        loss.backward()
        opt.step()

    meta_inputs, meta_labels = next(valid_data)  # get some non-training data
    meta_loss = loss_fn(model(meta_inputs), meta_labels)  # measure loss
    meta_gradients = torch.autograd.grad(meta_loss, init_params)  # wrong grads!
```

Just import higher!

```
import higher
model = MyModel(...)  # arbitrarily complex model
opt = torch.optim.MyOptim(...)  # favourite optim

with higher.innerloop_ctx(model, opt) as (fmodel, diffopt):
    for inputs, labels in train_data:  # do some training
        loss = loss_fn(fmodel(inputs), labels, extra=phi)
        diffopt.step(loss)

    meta_inputs, meta_labels = next(valid_data)  # get some non-training data
    meta_loss = loss_fn(model(meta_inputs), meta_labels)  # measure loss
    meta_gradients = torch.autograd.grad(meta_loss, init_params)  # wrong grads!
```

Just import higher!

```
import higher
model = MyModel(...)  # arbitrarily complex model
opt = torch.optim.MyOptim(...)  # favourite optim

with higher.innerloop_ctx(model, opt) as (fmodel, diffopt):
    for inputs, labels in train_data:  # do some training
        loss = loss_fn(fmodel(inputs), labels, extra=phi)
        diffopt.step(loss)

    meta_inputs, meta_labels = next(valid_data)  # get some non-training data
    meta_loss = loss_fn(model(meta_inputs), meta_labels)  # measure loss
    meta_gradients = torch.autograd.grad(meta_loss, init_params)  # wrong grads!
```

Just import higher!

```
import higher
model = MyModel(...)  # arbitrarily complex model
opt = torch.optim.MyOptim(...)  # favourite optim

with higher.innerloop_ctx(model, opt) as (fmodel, diffopt):
    for inputs, labels in train_data:  # do some training
        loss = loss_fn(fmodel(inputs), labels, extra=phi)
        diffopt.step(loss)

    meta_inputs, meta_labels = next(valid_data)  # get some non-training data
    meta_loss = loss_fn(fmodel(meta_inputs), meta_labels)  # measure loss
    meta_gradients = torch.autograd.grad(meta_loss, fmodel.parameters(time=0))
```

↑↑↑↑↑↑ == $\nabla_{\theta_{init}}$ meta_loss

↑↑↑↑↑↑ == θ_{init}

Just import higher!

```
import higher
model = MyModel(...)  # arbitrarily complex model
opt = torch.optim.MyOptim(...)  # favourite optim

with higher.innerloop_ctx(model, opt) as (fmodel, diffopt):
    for inputs, labels in train_data:  # do some training
        loss = loss_fn(fmodel(inputs), labels, extra=phi)
        diffopt.step(loss)

    meta_inputs, meta_labels = next(valid_data)  # get some non-training data
    meta_loss = loss_fn(fmodel(meta_inputs), meta_labels)  # measure loss
    meta_gradients = torch.autograd.grad(meta_loss, phi)
        ↑↑↑↑↑↑ == ∇φ meta_loss
```

How does this work?

Under the hood:

1. We **monkey-patch the torch module for the model** to use explicitly provided, or implicitly tracked, **fast weights** which can be updated through differentiable ops rather than in-place.

How does this work?

Under the hood:

1. We **monkey-patch the torch module for the model** to use explicitly provided, or implicitly tracked, **fast weights** which can be updated through differentiable ops rather than in-place.

2. **Differentiable versions of optimizers** wrap the regular optimizers, branching off their states, and perform updates on fast weights in a way which **yields grad functions**.

Patched modules

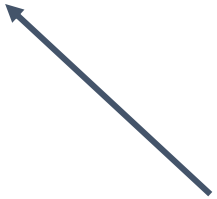
```
fmodel = higher.monkeypatch(
    module, device=None, copy_initial_weights=True)
```

the model you're unrolling.
Any torch.nn.Module instance
works...

Patched modules

```
fmodel = higher.monkeypatch(
    module, device=None, copy_initial_weights=True)
```

the model you're unrolling.
Any torch.nn.Module instance works...



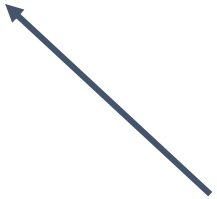
(optional) the device to do the unrolling on…

If None, use same device as module .

Patched modules

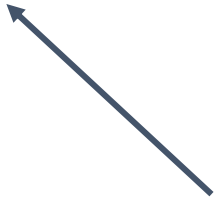
```
fmodel = higher.monkeypatch(
    module, device=None, copy_initial_weights=True)
```

the model you're unrolling. Any torch.nn.Module instance works...



(optional) the device to do the unrolling on…

If None, use same device as module .



If True (default), creates a safe copy of the model to unroll. i.e. the weights of module do not receive gradient or form part of grad funcs as we unroll...

Only set to False if doing MAML, or if you are disciplined about clearing gradients after your metalearning loop...

Patched modules

- fmodel(x, params=params) == model(x) (forward) but with explicitly provided “fast weights” params. Operations on these have grad funcs.

Patched modules

- fmodel(x, params=params) == model(x) (forward) but with explicitly provided “fast weights” params. Operations on these have grad funcs.
- fmodel tracks the latest fast weights, which are implicitly used when calling fmodel without the params kwarg.

Patched modules

- fmodel(x, params=params) == model(x) (forward) but with explicitly provided “fast weights” params. Operations on these have grad funcs.
- fmodel tracks the latest fast weights, which are implicitly used when calling fmodel without the params kwarg.
- fmodel.parameters() == latest_params
  fmodel.parameters(time=0) == init_params
  (i.e. model.parameters() when fmodel was created).
  The whole unroll history can be referenced this way…

Differentiable Optimizers

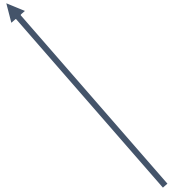
```
diffopt = higher.get_diff_optim(
    opt, reference_params, fmodel=None, device=None, track_higher_grads=True)
```

A torch.optim optimizer instance which will be mimicked. A safe copy of its state will be made…

Differentiable Optimizers

```
diffopt = higher.get_diff_optim(
    opt, reference_params, fmodel=None, device=None, track_higher_grads=True)
```

A torch.optim optimizer instance which will be mimicked. A safe copy of its state will be made…



Iterable of parameters from the model being optimized by opt. Assumed to be the output of model.parameters(). Must be the whole list even if only a subset is being used, e.g. through parameter groups.

Differentiable Optimizers

```
diffopt = higher.get_diff_optim(
    opt, reference_params, fmodel=None, device=None, track_higher_grads=True)
```

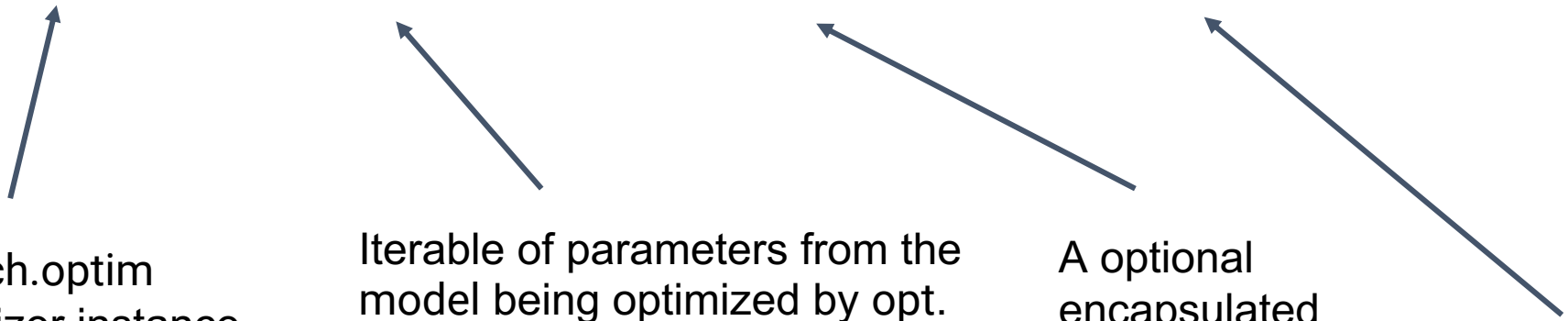
A torch.optim optimizer instance which will be mimicked. A safe copy of its state will be made…

Iterable of parameters from the model being optimized by opt. Assumed to be the output of model.parameters(). Must be the whole list even if only a subset is being used, e.g. through parameter groups.

A optional encapsulated fmodel. Providing this simplifies code by removing need to track/update fast weights explicitly.

Differentiable Optimizers

```
diffopt = higher.get_diff_optim(
    opt, reference_params, fmodel=None, device=None, track_higher_grads=True)
```



A torch.optim optimizer instance which will be mimicked. A safe copy of its state will be made…

Iterable of parameters from the model being optimized by opt. Assumed to be the output of model.parameters(). Must be the whole list even if only a subset is being used, e.g. through parameter groups.

A optional encapsulated fmodel. Providing this simplifies code by removing need to track/update fast weights explicitly.

(optional) the device to do the unrolling on…

If None, use same device as reference_params .

Differentiable Optimizers

```
diffopt = higher.get_diff_optim(
    opt, reference_params, fmodel=None, device=None, track_higher_grads=True)
```

A torch.optim optimizer instance which will be mimicked. A safe copy of its state will be made…

Iterable of parameters from the model being optimized by opt. Assumed to be the output of model.parameters(). Must be the whole list even if only a subset is being used, e.g. through parameter groups.

A optional encapsulated fmodel. Providing this simplifies code by removing need to track/update fast weights explicitly.

(optional) the device to do the unrolling on…

If None, use same device as reference_params .

Set to false to disable higher grad tracking (e.g. test mode)

Differentiable Optimizers

- diffopt.step(loss, params=params) **operates like opt.step()** after loss.backward() (which is not needed here), and returns new params as updated by the optimizer.

Differentiable Optimizers

- diffopt.step(loss, params=params) **operates like opt.step()** after loss.backward() (which is not needed here), and returns new params as updated by the optimizer.
- If fmodel is encapsulated by diffopt, it is **sufficient** to call diffopt.step(loss) as weights are **tracked** by fmodel.

Differentiable Optimizers

- diffopt.step(loss, params=params) **operates like opt.step()** after loss.backward() (which is not needed here), and returns new params as updated by the optimizer.
- If fmodel is encapsulated by diffopt, it is **sufficient** to call diffopt.step(loss) as weights are **tracked** by fmodel.
- Returned/tracked params have **grad functions attached**, allowing **backprop through optimization step**.

Differentiable Optimizers

- diffopt.step(loss, params=params) **operates like opt.step()** after loss.backward() (which is not needed here), and returns new params as updated by the optimizer.
- If fmodel is encapsulated by diffopt, it is **sufficient** to call diffopt.step(loss) as weights are **tracked** by fmodel.
- Returned/tracked params have **grad functions attached**, allowing **backprop through optimization step**.
- **No need to call zero_grad()** on diffopt instances.

Keep it simple with innerloop_ctx

```
with higher.innerloop_ctx(model, opt, **loop_kwargs) as (fmodel, diffopt):
        # write your inner loop here
# Warning: It is not recommended to reference fmodel/diffopt after context
# ends (risk of memory problems).
```

Produces a patched/diff version of model/opt, with optimizer encapsulating patched fmodel.

Keep it simple with innerloop_ctx

```
with higher.innerloop_ctx(model, opt, **loop_kwargs) as (fmodel, diffopt):
        # write your inner loop here
# Warning: It is not recommended to reference fmodel/diffopt after context
# ends (risk of memory problems).
```

Produces a patched/diff version of model/opt, with optimizer encapsulating patched fmodel.

loop_kwargs == (most) keyword args from monkeypatch and get_diff_optim

Keep it simple with innerloop_ctx

```
with higher.innerloop_ctx(model, opt, **loop_kwargs) as (fmodel, diffopt):
        # write your inner loop here
# Warning: It is not recommended to reference fmodel/diffopt after context
# ends (risk of memory problems).
```

Produces a patched/diff version of model/opt, with optimizer encapsulating patched fmodel.

loop_kwargs == (most) keyword args from monkeypatch and get_diff_optim

fmodel/diffopt branch from model/optim at context creation.

What's supported?

Any model that subclasses torch.nn.Module

What's supported?

Any model that subclasses torch.nn.Module

Nested modules, shared weights, state (e.g. batchnorm)

What's supported?

Any model that subclasses torch.nn.Module

Nested modules, shared weights, state (e.g. batchnorm)

Pre-trained weights, existing saved models

What's supported?

Any model that subclasses torch.nn.Module

Nested modules, shared weights, state (e.g. batchnorm)

Pre-trained weights, existing saved models

almost
↓
Every optimizer in torch.optim

(except SparseAdam and LBFGS, and some grad stability/correctness issues for Adagrad and RMSprop)

What's supported?

Any model that subclasses torch.nn.Module

Nested modules, shared weights, state (e.g. batchnorm)

Pre-trained weights, existing saved models

almost
↓
Every optimizer in torch.optim
(except SparseAdam and LBFGS, and some grad stability/correctness issues for Adagrad and RMSprop)

Metalearning on **GPU**(s)

What's the catch?

- **Space complexity** of inner loop is O(n) for n steps unrolled

What's the catch?

- **Space complexity** of inner loop is O(n) for n steps unrolled
- Thorough unit tests, but no blanket **correctness guarantees** for complex/exotic model architures. If in doubt, get in touch!

What's the catch?

- **Space complexity** of inner loop is O(n) for n steps unrolled
- Thorough unit tests, but no blanket **correctness guarantees** for complex/exotic model architures. If in doubt, get in touch!
- Possible **grad stability issues** for differentiable optimizers as number of unroll steps increases. Mitigated for Adam, SGD, but other opts still experimental. If in doubt, get in touch!

What's the catch?

- **Space complexity** of inner loop is O(n) for n steps unrolled
- Thorough unit tests, but no blanket **correctness guarantees** for complex/exotic model architures. If in doubt, get in touch!
- Possible **grad stability issues** for differentiable optimizers as number of unroll steps increases. Mitigated for Adam, SGD, but other opts still experimental. If in doubt, get in touch!
- Other bugs, memory issues may arise. It's alpha software...

When can I use it?

Now!

git clone git@github.com:fairinternal/higher.git
cd higher
source activate my_favorite_env
pip install .

In your code, just import higher and start metalearning! 😀

Thanks!

https://github.com/fairinternal/higher

Contact egrefen@fb.com for questions/comments

**Cheers to Phu Mon Htut for helping road test higher,
to Brandon Amos + Denis Yarats for writing examples (MAML, SPEN),
and to Adam Paszke for the original monkey patching gist…**