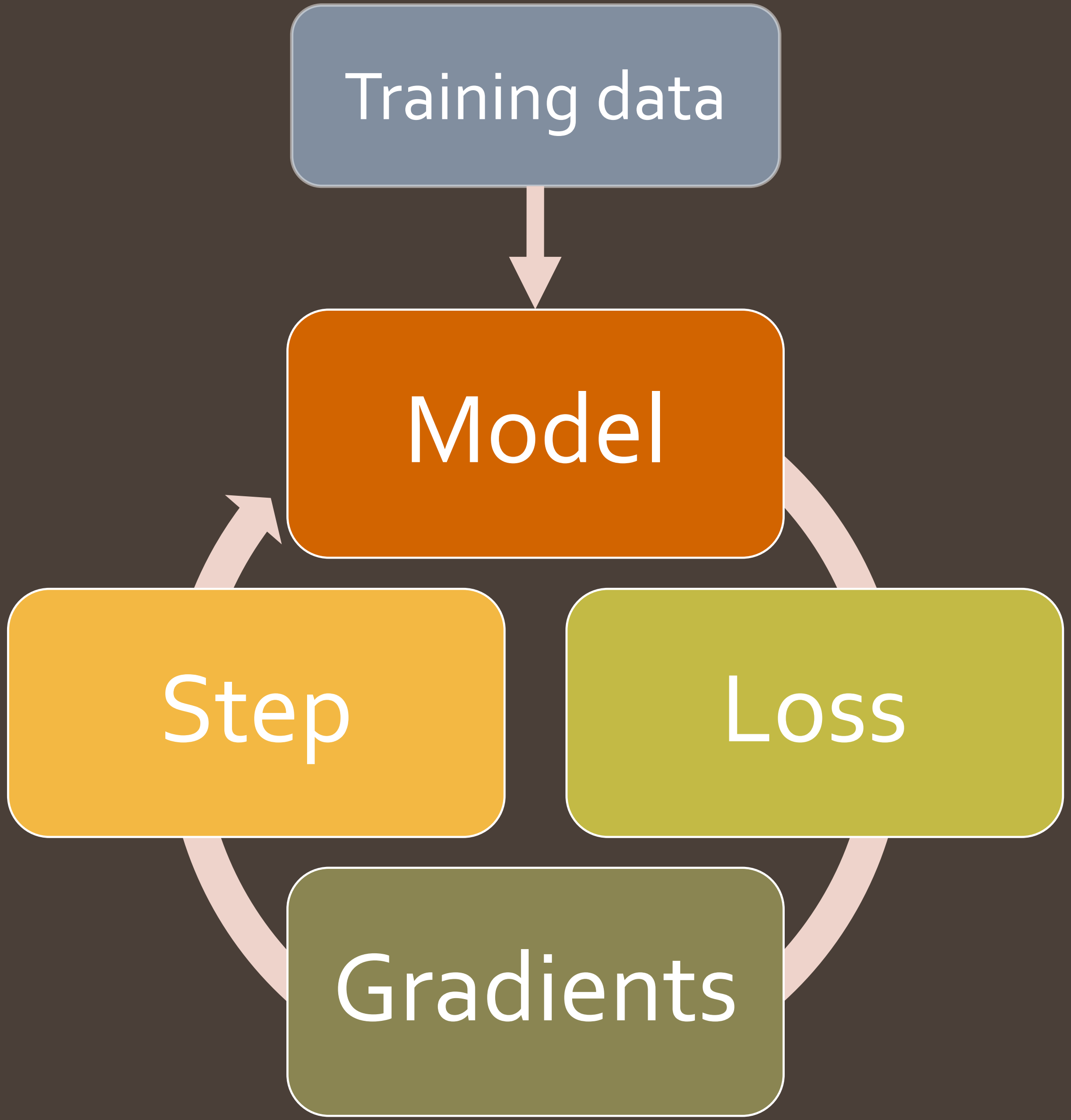
An infinitely customizable training loop

Sylvain Gugger

The basic training loop iterates repeatedly over 4 simple steps.

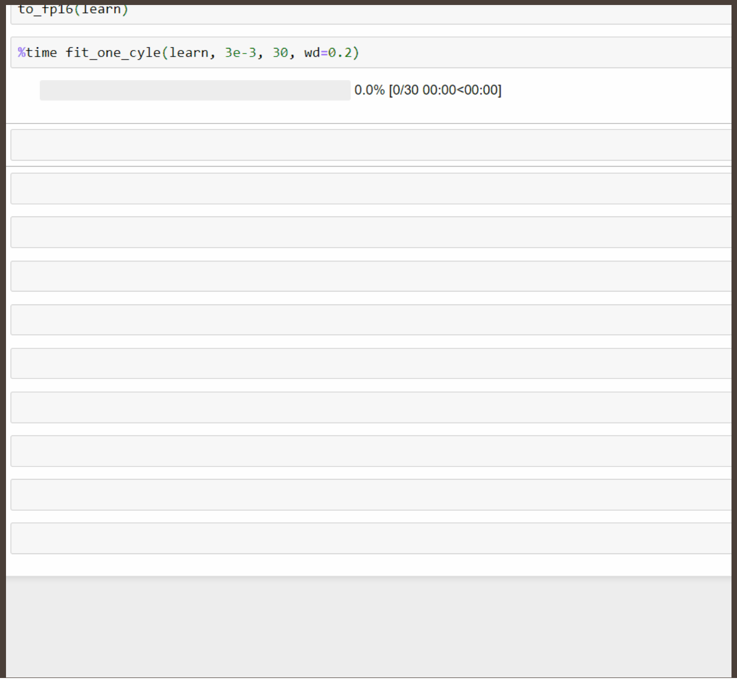


And in PyTorch, this fits in just five lines of codes.

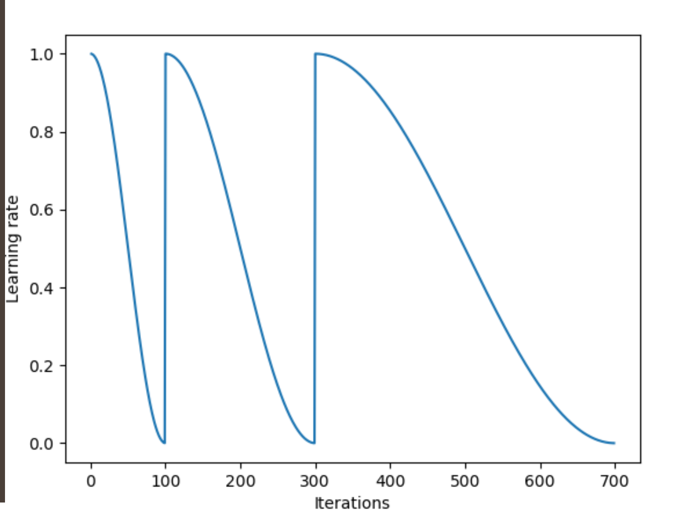
```
def train(train_dl, model, epoch, opt, loss_func):
    for _ in range(epoch):
        model.train()
        for xb,yb in train_dl:
            out = model(xb)
            loss = loss_func(out, yb)
            loss.backward()
            opt.step()
            opt.zero_grad()
```

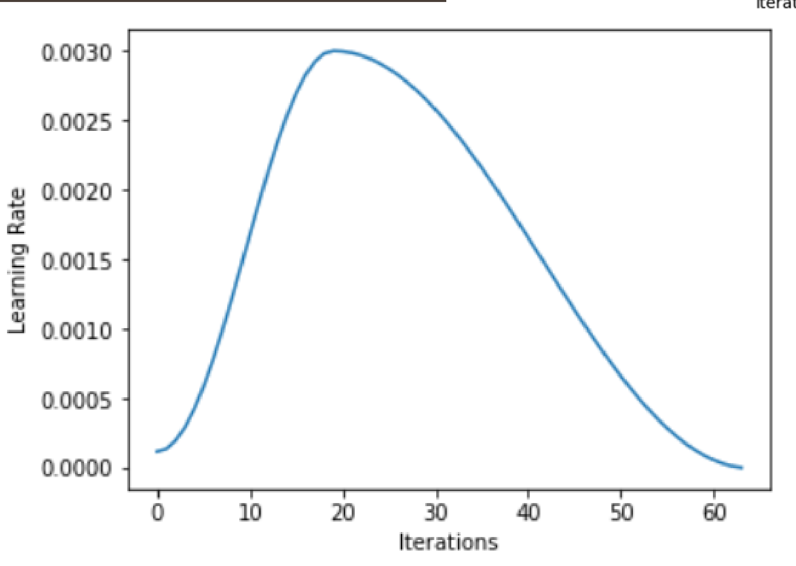
But there are multiple tweaks you can add to it.

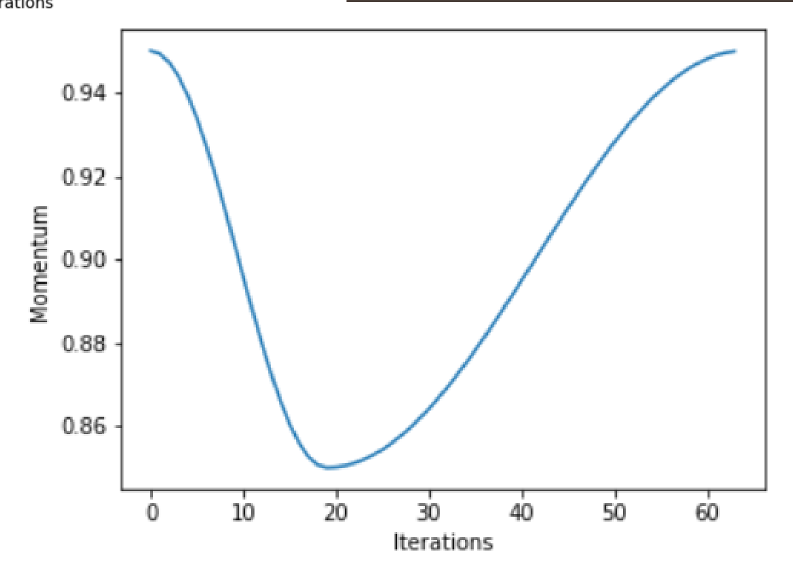
Keeping track of losses and metrics



Hyper-parameters schedules







Every regularization technique

Averaging Weights Leads to Wider Optima and Better Generalization

Pavel Izmailov[*1] **Dmitrii Podoprikhin**[*2,3] **Timur Garipov**[*4,5] **Dmitry Vetrov**[2,3] **Andrew Gordon Wilson**
[1]Cornell University, [2]Higher School of Economics, [3]Samsung-HSE Laboratory,
[4]Samsung AI Center in Moscow, [5]Lomonosov Moscow State University

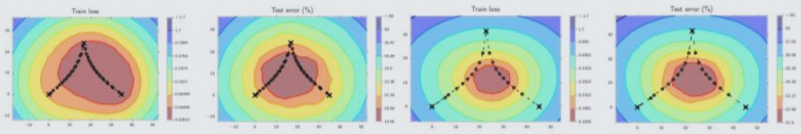


Figure 3: The L_2-regularized cross-entropy train loss and test error surfaces of a Preactivation ResNet-164 on CIFAR-100 in the plane containing the first, middle and last points (indicated by black crosses) in the trajectories with (**left two**) cyclical and (**right two**) constant learning rate schedules.

2.1. L_2 activation regularization (AR)

While L_2 regularization is traditionally used on the weights of machine learning models (L_2 weight decay), it could also be used on the activations. We define AR as

$$\alpha\, L_2(m \odot h_t)$$

where m is the dropout mask used by later parts of the model, $L_2(\cdot) = \|\cdot\|_2$ (L_2 norm), h_t is the output of the RNN at timestep t, and α is a scaling coefficient.

Temporal activation regularization (TAR) is a direct descendant of this slowness regularization, minimizing

$$\beta\, L_2(h_t - h_{t+1})$$

where $L_2(\cdot) = \|\cdot\|_2$ (L_2 norm), h_t is the output of the RNN at timestep t, and β is a scaling coefficient.

Decoupled Weight Decay Regularization

Ilya Loshchilov & Frank Hutter
University of Freiburg
Freiburg, Germany,
{ilya,fh}@cs.uni-freiburg.de

Algorithm 2 Adam with L_2 regularization and Adam with decoupled weight decay (AdamW)

given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
repeat
$t \leftarrow t + 1$
$\nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{\theta}_{t-1})$ ▷ select batch and return the corresponding gradient
$\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1})$ $+\lambda\boldsymbol{\theta}_{t-1}$
$\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t$ ▷ here and below all operations are element-wise
$\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2$
$\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t/(1-\beta_1^t)$ ▷ β_1 is taken to the power of t
$\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t/(1-\beta_2^t)$ ▷ β_2 is taken to the power of t
$\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) +\lambda\boldsymbol{\theta}_{t-1}\right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{\theta}_t$

mixup: Beyond Empirical Risk Minimization

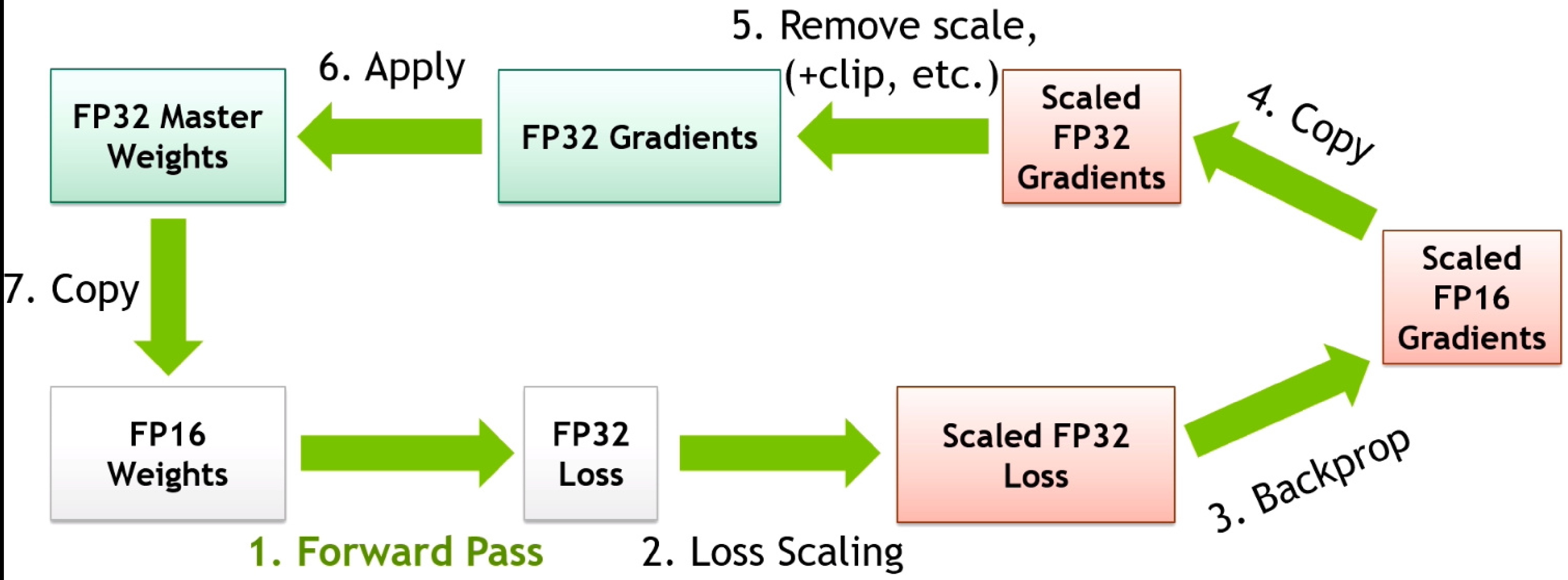
Hongyi Zhang
MIT

Moustapha Cisse, Yann N. Dauphin, David Lopez-Paz*
FAIR

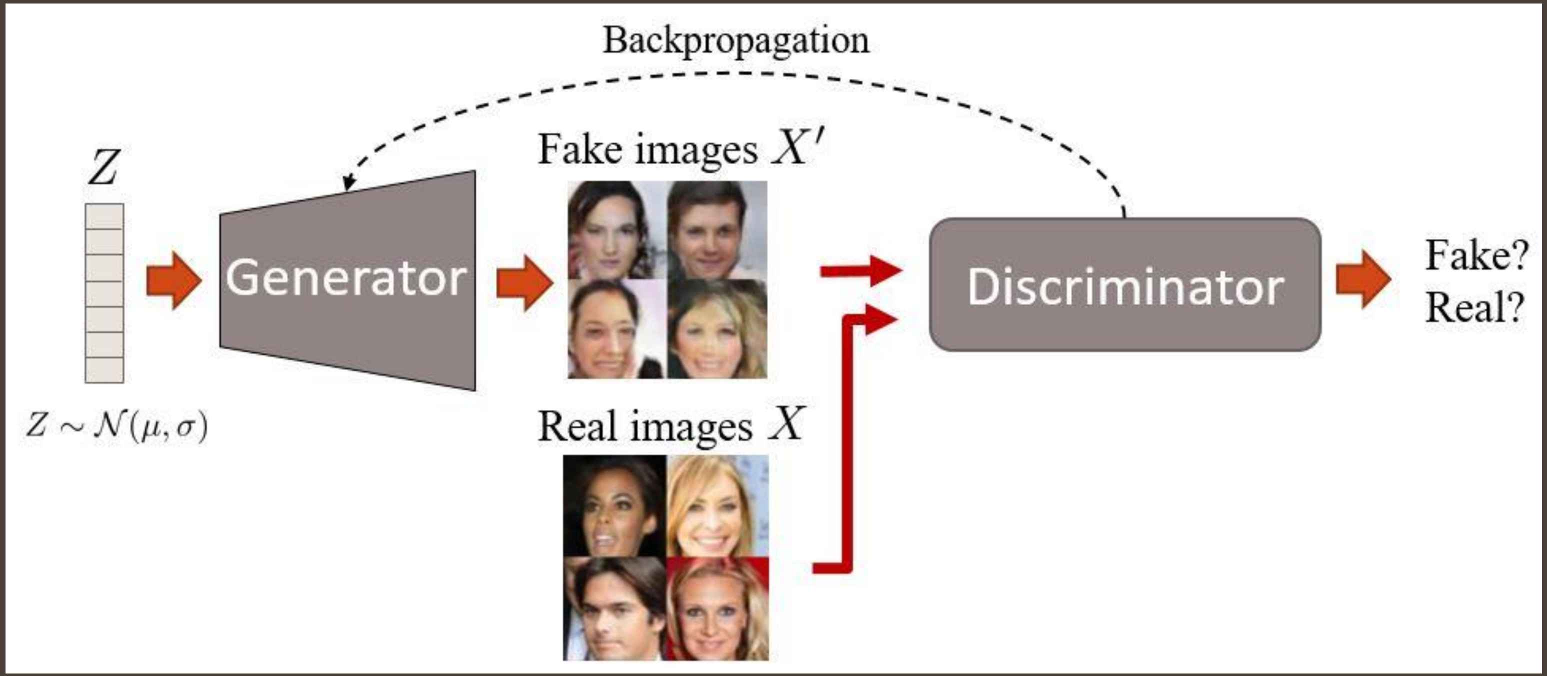
Contribution Motivated by these issues, we introduce a simple and data-agnostic data augmentation routine, termed *mixup* (Section 2). In a nutshell, *mixup* constructs virtual training examples

$$\tilde{x} = \lambda x_i + (1-\lambda)x_j, \quad \text{where } x_i, x_j \text{ are raw input vectors}$$
$$\tilde{y} = \lambda y_i + (1-\lambda)y_j, \quad \text{where } y_i, y_j \text{ are one-hot label encodings}$$

MIXED PRECISION TRAINING



Or more complex trainings like GANs



So you either have to rewrite a new loop for each new kind of training…

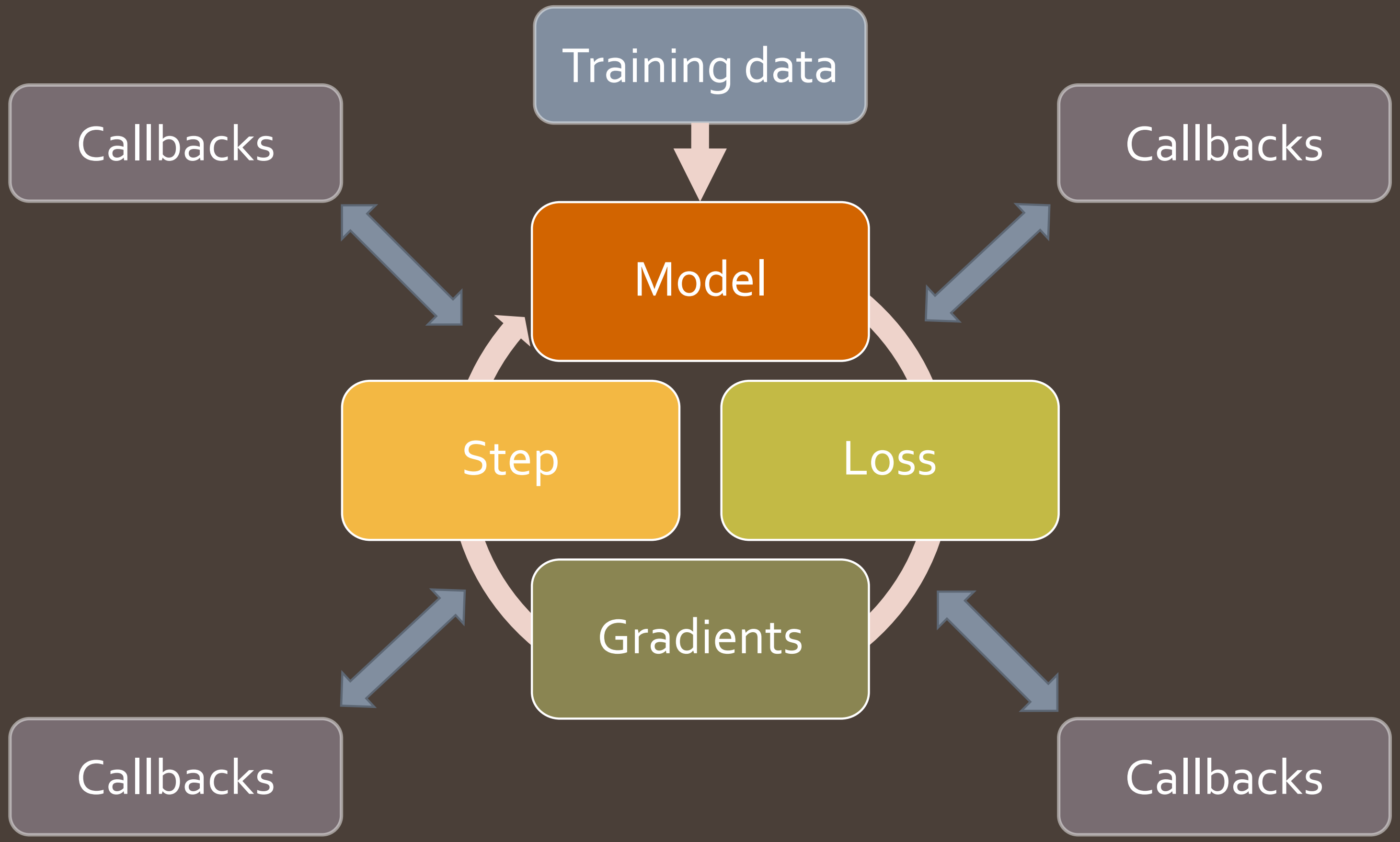
Or try to write something that incorporates everything you can think of.

```python
class Stepper():
    def __init__(self, m, opt, crit, clip=0, reg_fn=None, fp16=False, loss_scale=1):
        self.m,self.opt,self.crit,self.clip,self.reg_fn = m,opt,crit,clip,reg_fn
        self.fp16 = fp16
        self.reset(True)
        if self.fp16: self.fp32_params = copy_model_to_fp32(m, opt)
        self.loss_scale = loss_scale

    def reset(self, train=True):
        if train: apply_leaf(self.m, set_train_mode)
        else: self.m.eval()
        if hasattr(self.m, 'reset'):
            self.m.reset()
            if self.fp16: self.fp32_params = copy_model_to_fp32(self.m, self.opt)

    def step(self, xs, y, epoch):
        xtra = []
        output = self.m(*xs)
        if isinstance(output,tuple): output,*xtra = output
        if self.fp16: self.m.zero_grad()
```

Fortunately there is a way around this: Callbacks



Going back from our basic training loop…

```python
def train(train_dl, model, epoch, opt, loss_func):
    for _ in range(epoch):
        model.train()
        for xb,yb in train_dl:
            out = model(xb)
            loss = loss_func(out, yb)
            loss.backward()
            opt.step()
            opt.zero_grad()
```

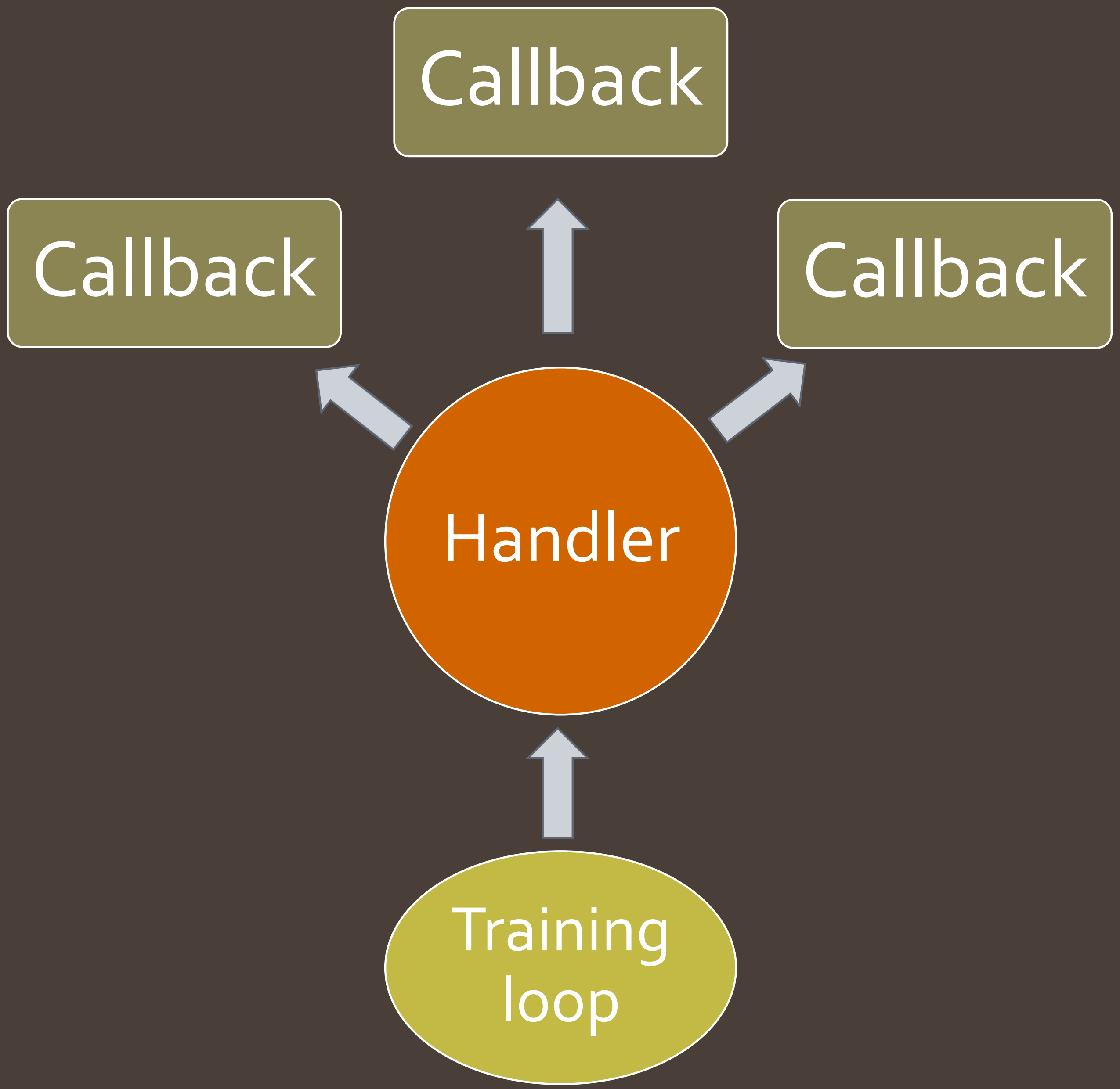
…we can just add callbacks everywhere.

```
def train(train_dl, model, epoch, opt, loss_func, callbacks):
    callbacks.on_train_begin()
    for _ in range(epoch):
        callbacks.on_epoch_begin()
        model.train()
        for xb,yb in train_dl:
            callbacks.on_batch_begin()
            out = model(xb)
            callbacks.on_loss_begin()
            loss = loss_func(out, yb)
            callbacks.on_loss_begin()
            loss.backward()
            callbacks.on_step_begin()
            opt.step()
            callbacks.on_step_end()
            opt.zero_grad()
            callbacks.on_batch_end()
        callbacks.on_epoch_end()
    callbacks.on_train_end()
```

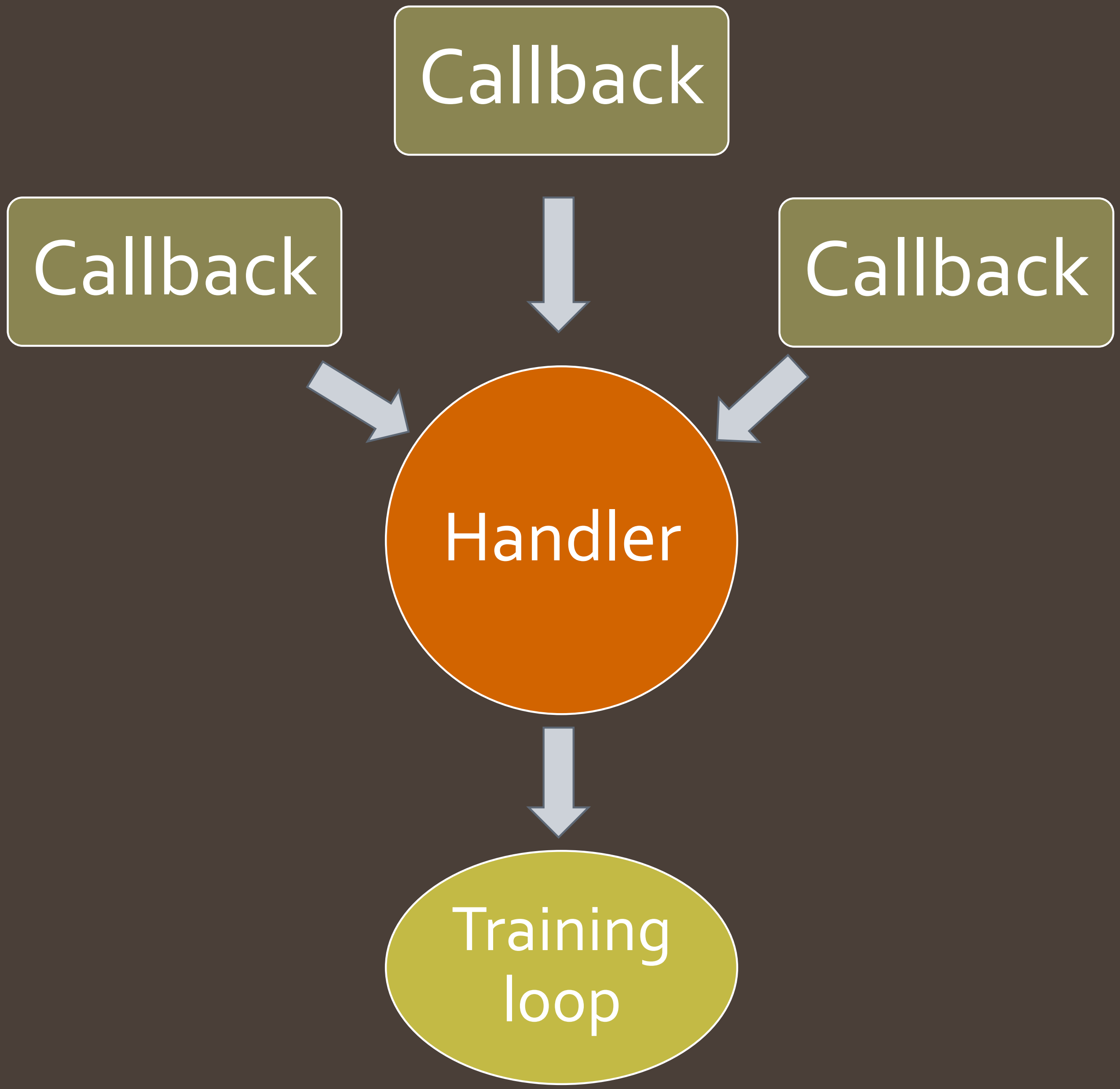
But for this to be infinitely flexible, each callback need to see everything that's going on…

… and be able to change anything in the training loop.

So we add a handler that stores all the information in the training loop to feed to each callback.



Which in turn can all update the state of the handler, which it gives back to the training loop.



The general context is provided by grouping the model, data, loss function and optimizer in a Learner class.

```
@dataclass
class DataBunch():
    train_dl:DataLoader
    valid_dl:DataLoader
```

```
@dataclass
class BasicLearner():
    model:nn.Module
    loss_func:LossFunction
    opt:optim.Optimizer
    data:DataBunch
```

Training loop

Those updates can be new values, or flags that skip steps or stop the training.

```
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()
        cb_handler.on_epoch_begin(epoch)
        for xb,yb in learn.data.train_dl:
            xb,yb = cb_handler.on_batch_begin(xb,yb)
            out = learn.model(xb)
            out = cb_handler.on_loss_begin(out)
            loss = learn.loss_func(out, yb)
            loss, skip_backward = callbacks.on_loss_begin(loss)
            if not skip_backward: loss.backward()
            if not cb_handler.on_step_begin(): learn.opt.step()
            if not cb_handler.on_step_end():   learn.opt.zero_grad()
            if not callbacks.on_batch_end():   break
        val_loss, mets = validate(learn.data.valid_dl, model, metrics)
        if not callbacks.on_epoch_end(val_loss, mets): break
    callbacks.on_train_end()
```

Then each tweak of the training loop can be entirely written in its own callback.

```python
class LRScheduler(LearnerCallback):

    def on_batch_begin(self, iteration, **kwargs):
        self.learn.opt.lr = my_function(iteration)
```

```python
class EarlyStopping(Callback):

    def on_epoch_end(self, last_metrics, **kwargs):
        if not_happy(last_metrics): return {'stop_training': True}
```

```python
class ParallelTrainer(LearnerCallback):
    _order = -20
    def on_train_begin(self, **kwargs):
        self.learn.model = DataParallel(self.learn.model)
    def on_train_end  (self, **kwargs):
        self.learn.model = self.learn.model.module
```

```python
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)
```

Even if they are a bit more complex.

```python
class AccumulateScheduler(LearnerCallback):
    "Does accumulated step every nth step by accumulating gradients"
    def __init__(self, learn:Learner, n_step:int = 1, drop_last:bool = False):
        super().__init__(learn)
        self.n_step,self.drop_last = n_step,drop_last
        if hasattr(self.loss_func, "reduction") and (self.loss_func.reduction != "sum"):
            warn("For better gradients consider 'reduction=sum'")

    def on_epoch_begin(self, **kwargs):
        "Init samples and batches."
        self.acc_samples, self.acc_batches = 0., 0.

    def on_batch_begin(self, last_input, last_target, **kwargs):
        "Accumulate samples and batches"
        self.acc_samples += last_input.shape[0]
        self.acc_batches += 1

    def on_backward_end(self, **kwargs):
        "Accumulated step and reset samples."
        if (self.acc_batches % self.n_step) == 0:
            for p in (self.learn.model.parameters()):
                if p.requires_grad: p.grad.div_(self.acc_samples)
            self.acc_samples = 0
        else: return {'skip_step':True, 'skip_zero':True}

    def on_epoch_end(self, **kwargs):
        "Step the rest of the accumulated grads if not perfectly divisible"
        for p in (self.learn.model.parameters()):
            if p.requires_grad: p.grad.div_(self.acc_samples)
        if not self.drop_last: self.learn.opt.step()
        self.learn.opt.zero_grad()
```

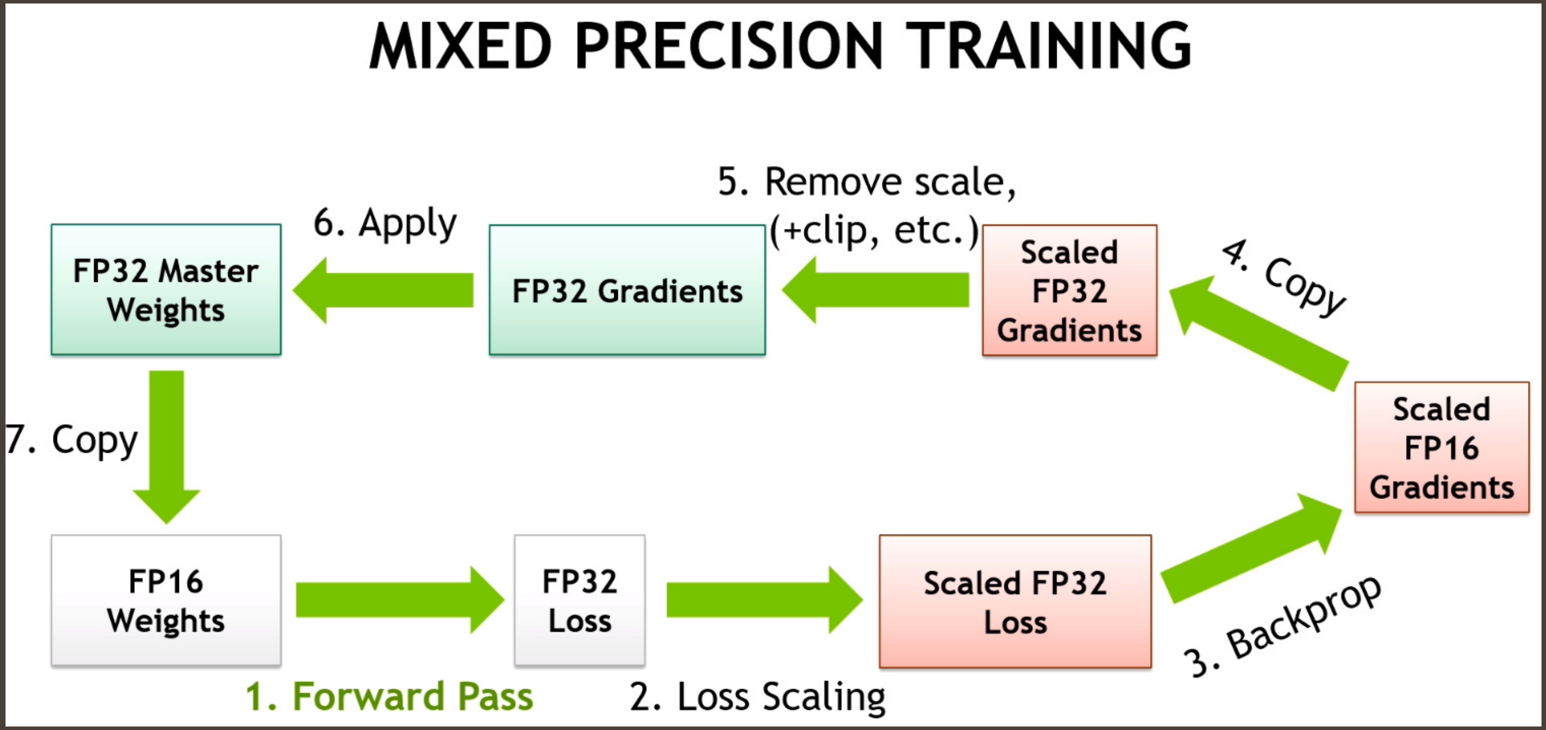
It’s then easy to mix and match those blocks together…

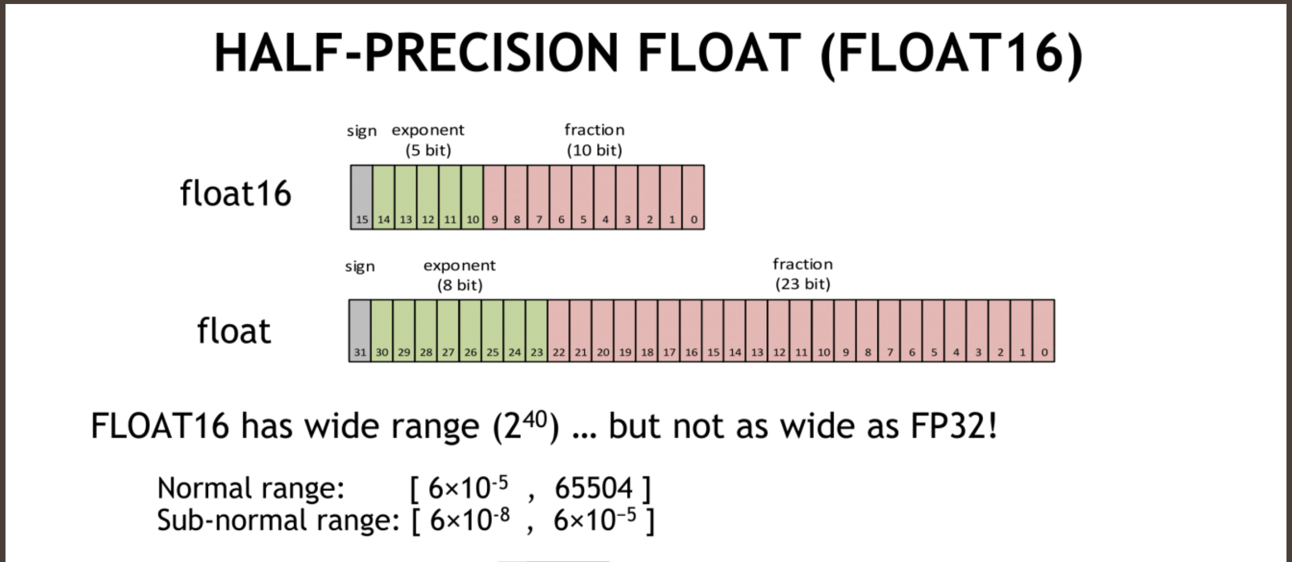
…and to add a new one!

SGDR	Record metrics	Multi-GPU	Mixed Precision
GAN	Gradient clipping	1cycle	SWA
Mixup	Save best model	Tensoboard	Early stopping
Gradient accumulation	Loss penalties	AdamW	Yours!

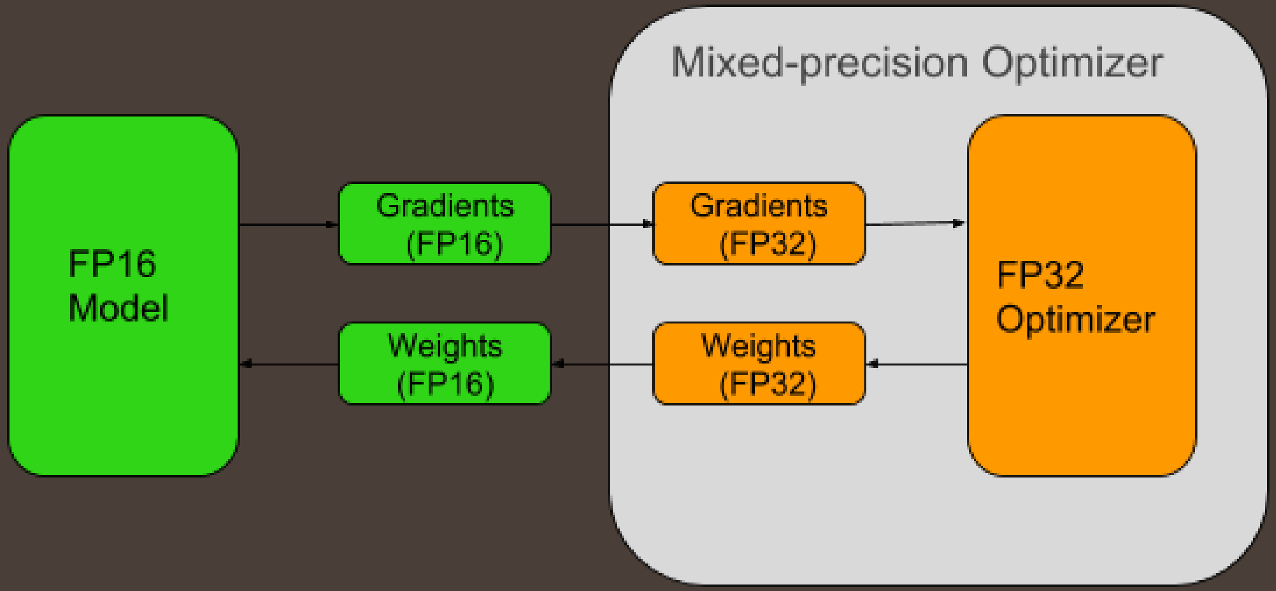
```
learn.fit(epochs, lr, wd, callbacks)
```

Case study: Mixed precision training





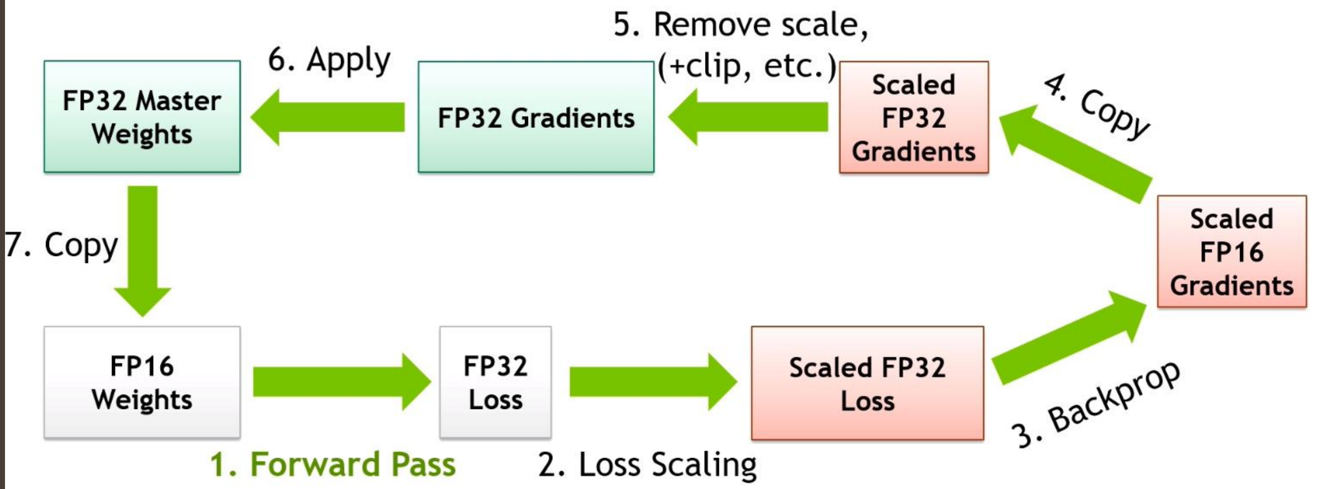
We define master parameters (in FP32) and model parameters (in FP16)




```python
class MixedPrecision(LearnerCallback):
    _order = 999 #Need to run after things that could call on_backward_begin and change the loss
    "Callback that handles mixed-precision training."
    def __init__(self, learn:Learner, loss_scale:float=None, clip:float=None,
                 flat_master:bool=False):
        super().__init__(learn)
        self.flat_master,self.loss_scale,self.clip = flat_master,loss_scale,clip
        assert torch.backends.cudnn.enabled, "Mixed precision training requires cudnn."

    def on_train_begin(self, **kwargs:Any)->None:
        "Prepare the master model."
        #Get a copy of the model params in FP32
        self.model_params, self.master_params = get_master(self.learn.layer_groups, self.flat_master)
        #Changes so that the optimization step is done in FP32.
        self.learn.opt = self.learn.opt.new_with_params(self.master_params)
```

And we can easily add the relevant parts in the appropriate event function.



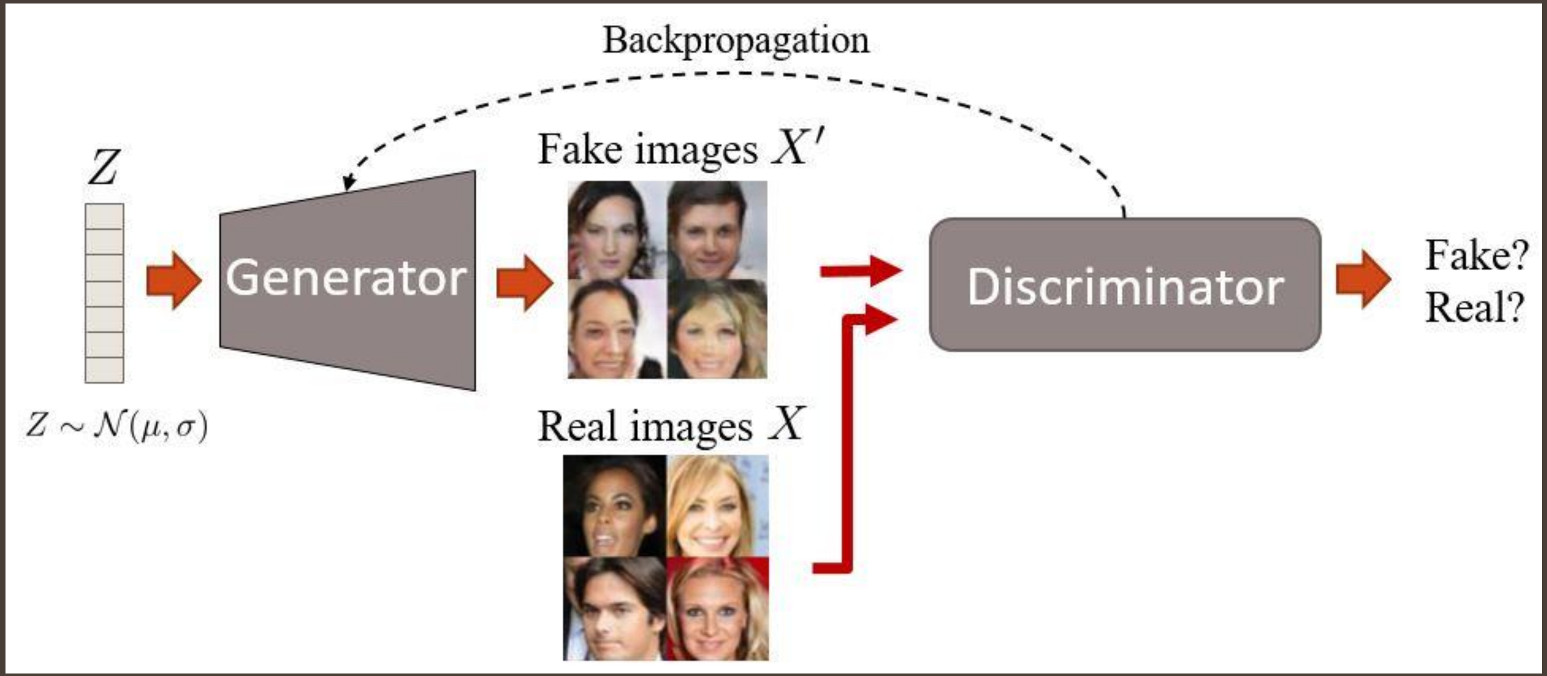

```python
def on_loss_begin(self, last_output:Tensor, **kwargs:Any) -> Tensor:
    "Convert half precision output to FP32 to avoid reduction overflow."
    return {'last_output': to_float(last_output)}

def on_backward_begin(self, last_loss:Rank0Tensor, **kwargs:Any) -> Rank0Tensor:
    "Scale gradients up by `self.loss_scale` to prevent underflow."
    #To avoid gradient underflow, we scale the gradients
    ret_loss = last_loss * self.loss_scale
    return {'last_loss': ret_loss}

def on_backward_end(self, **kwargs:Any)->None:
    "Convert the gradients back to FP32 and divide them by the scale."
    model_g2master_g(self.model_params, self.master_params, self.flat_master)
    for group in self.master_params:
        for param in group:
            if param.grad is not None: param.grad.div_(self.loss_scale)
    if self.clip is not None:
        for group in self.master_params: nn.utils.clip_grad_norm_(group, self.clip)

def on_step_end(self, **kwargs:Any)->None:
    "Update the params from master to model and zero grad."
    #Zeros the gradients of the model since the optimizer is disconnected.
    self.learn.model.zero_grad()
    master2model(self.model_params, self.master_params, self.flat_master)
```

Case study: GAN Training



We define a model and a loss function that wraps critic and generator with an inner switch.

```
class GANModule(nn.Module):
    "Wrapper around a `generator` and a `critic` to create a GAN."
    def __init__(self, generator:nn.Module=None, critic:nn.Module=None, gen_mode:bool=False):
        super().__init__()
        self.gen_mode = gen_mode
        if generator: self.generator,self.critic = generator,critic

    def forward(self, *args):
        return self.generator(*args) if self.gen_mode else self.critic(*args)

    def switch(self, gen_mode:bool=None):
        "Put the model in generator mode if `gen_mode`, in critic mode otherwise."
        self.gen_mode = (not self.gen_mode) if gen_mode is None else gen_mode
```

```
class GANLoss(GANModule):
    "Wrapper around `loss_funcC` (for the critic) and `loss_funcG` (for the generator)."
    def __init__(self, loss_funcG:Callable, loss_funcC:Callable, gan_model:GANModule):
        super().__init__()
        self.loss_funcG,self.loss_funcC,self.gan_model = loss_funcG,loss_funcC,gan_model

    def generator(self, output, target):
        """Evaluate the `output` with the critic then uses `self.loss_funcG` to combine
        it with `target`."""
        fake_pred = self.gan_model.critic(output)
        return self.loss_funcG(fake_pred, target, output)

    def critic(self, real_pred, input):
        """Create some `fake_pred` with the generator from `input` and compare them to `real_pred`
        in `self.loss_funcD`."""
        fake = self.gan_model.generator(input.requires_grad_(False)).requires_grad_(True)
        fake_pred = self.gan_model.critic(fake)
        return self.loss_funcC(real_pred, fake_pred)
```

Then the base of the GANTrainer Callback is to switch from the generator to the critic.

```python
class GANTrainer(LearnerCallback):
    "Handles GAN Training."
    _order=-20
    def __init__(self, learn:Learner, clip:float=None, gen_first:bool=False):
        super().__init__(learn)
        self.clip,self.gen_first = clip,gen_first
        self.generator,self.critic = self.model.generator,self.model.critic

    def _set_trainable(self):
        train_model = self.generator if     self.gen_mode else self.critic
        loss_model  = self.generator if not self.gen_mode else self.critic
        requires_grad(train_model, True)
        requires_grad(loss_model, False)

    def switch(self, gen_mode:bool=None):
        "Switch the model, if `gen_mode` is provided, in the desired mode."
        self.gen_mode = (not self.gen_mode) if gen_mode is None else gen_mode
        self.opt = self.opt_gen if self.gen_mode else self.opt_critic
        self._set_trainable()
        self.model.switch(gen_mode)
        self.loss_func.switch(gen_mode)
```

Then it's just a matter of making sure we have things fed to the model according to the right mode.

```python
def on_train_begin(self, **kwargs):
    "Create the optimizers for the generator and critic if necessary, initialize smootheners."
    if not getattr(self,'opt_gen',None):
        self.opt_gen = self.opt.new([nn.Sequential(*flatten_model(self.generator))])
    else: self.opt_gen.lr,self.opt_gen.wd = self.opt.lr,self.opt.wd
    if not getattr(self,'opt_critic',None):
        self.opt_critic = self.opt.new([nn.Sequential(*flatten_model(self.critic))])
    else: self.opt_critic.lr,self.opt_critic.wd = self.opt.lr,self.opt.wd
    self.gen_mode = self.gen_first
    self.switch(self.gen_mode)
    self.closses,self.glosses = [],[]
    self.recorder.add_metric_names(['gen_loss', 'disc_loss'])

def on_train_end(self, **kwargs):
    "Switch in generator mode for showing results."
    self.switch(gen_mode=True)

def on_batch_begin(self, last_input, last_target, **kwargs):
    "Clamp the weights with `self.clip` if it's not None, return the correct input."
    if self.clip is not None:
        for p in self.critic.parameters(): p.data.clamp_(-self.clip, self.clip)
    if self.gen_mode: return {'last_input':last_input, 'last_target':last_target}
    else:             return {'last_input':last_target,'last_target':last_input}

def on_backward_begin(self, last_loss, last_output, **kwargs):
    "Record `last_loss` in the proper list."
    if self.gen_mode: self.glosses.append(last_loss.detach().cpu())
    else:             self.closses.append(last_loss.detach().cpu())

def on_epoch_begin(self, epoch, **kwargs):
    "Put the critic or the generator back to eval if necessary."
    self.switch(self.gen_mode)

def on_epoch_end(self, pbar, epoch, last_metrics, **kwargs):
    "Put the various losses in the recorder and show a sample image."
    return add_metrics(last_metrics, self.glosses[-1], self.closses[-1])
```

Then another Callback is responsible for switching back and forth

```python
class FixedGANSwitcher(LearnerCallback):
    "Switcher to do `n_crit` iterations of the critic then `n_gen` iterations of the generator."
    def __init__(self, learn:Learner, n_crit:Union[int,Callable]=1, n_gen:Union[int,Callable]=1):
        super().__init__(learn)
        self.n_crit,self.n_gen = n_crit,n_gen

    def on_train_begin(self, **kwargs):
        "Initiate the iteration counts."
        self.n_c,self.n_g = 0,0

    def on_batch_end(self, iteration, **kwargs):
        "Switch the model if necessary."
        if self.learn.gan_trainer.gen_mode:
            self.n_g += 1
            n_iter,n_in,n_out = self.n_gen,self.n_c,self.n_g
        else:
            self.n_c += 1
            n_iter,n_in,n_out = self.n_crit,self.n_g,self.n_c
        target = n_iter if isinstance(n_iter, int) else n_iter(n_in)
        if target == n_out:
            self.learn.gan_trainer.switch()
            self.n_c,self.n_g = 0,0
```

In conclusion

A well-designed Callback system allows you to:

- keep the training loop as simple as possible
- have each tweak independently written in its own Callback
- easily mix and match, or perform ablation studies
- easily add a new experiment idea and debug it
- make it simple for contributors to add their own

You can use this system and all our Callbacks by installing fastai:

```
conda install -c pytorch -c fastai fastai
```