FP16

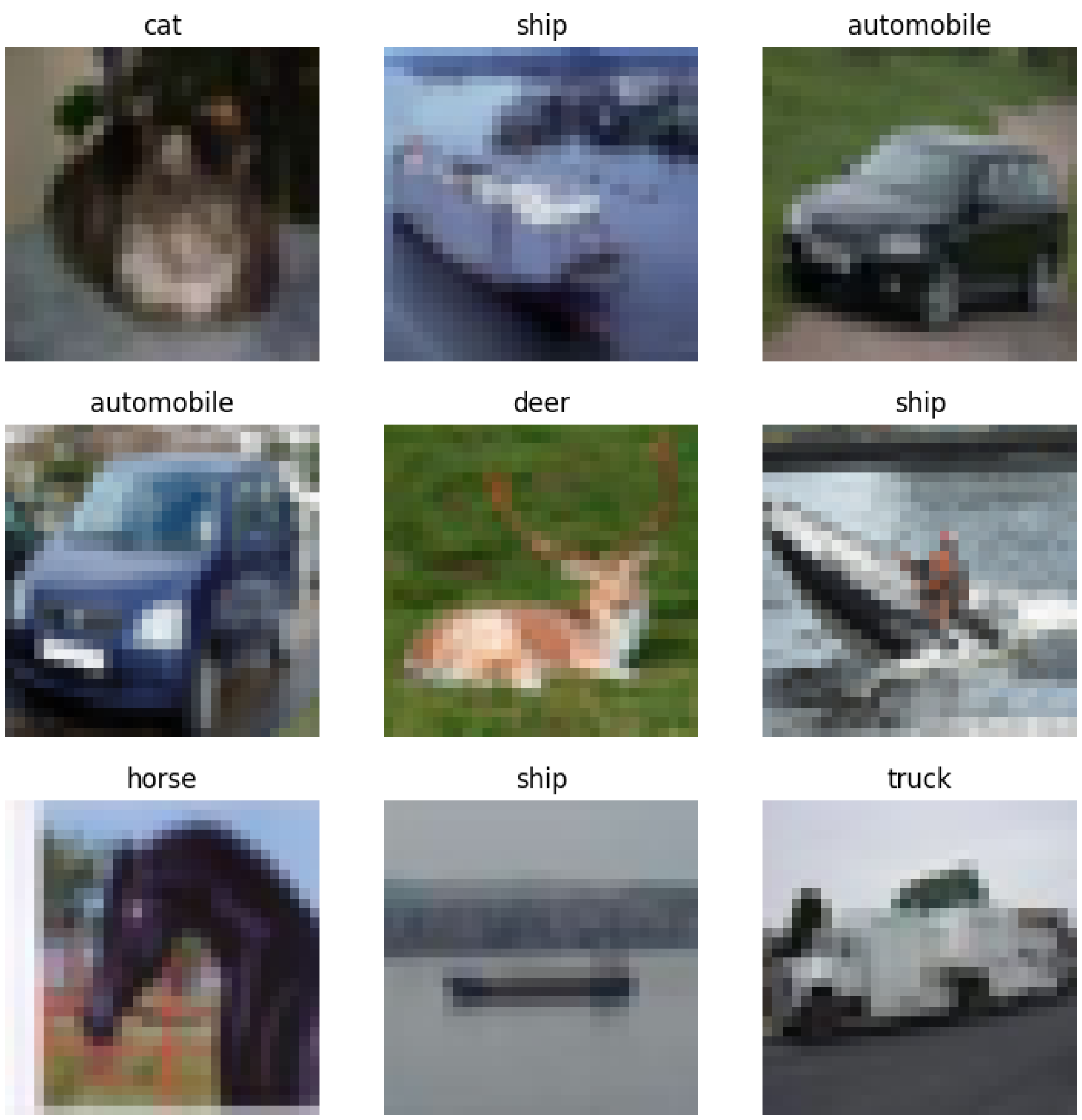
marii

We are going to take a look at how mixed precision or fp16 training works.

Getting setup

Lets start by looking at our data. We are working with low resolution cifar images.

```
bs=128
dls=DataBlock((ImageBlock(),
               CategoryBlock()),
          item_tfms=[Resize(32)],
          batch_tfms=(Normalize.from_stats(*cifar_stats)),
          get_items=get_image_files,
          get_y=parent_label
).dataloaders(path,bs=bs,val_bs=2*bs)
dls.show_batch()
```



We create a function to recreate our dataloader, because we will be doing it a lot for repeatability.

```
def create_dls():
    bs=128
    return DataBlock((ImageBlock(),
                      CategoryBlock()),
                item_tfms=[Resize(32)],
                batch_tfms=(Normalize.from_stats(*cifar_stats)),
                get_items=get_image_files,
                get_y=parent_label
    ).dataloaders(path,bs=bs,val_bs=2*bs)
```

We will just use a simple resnet to test our work.

```
learn=Learner(dls,resnet18(),opt_func=SGD)
```

We can make a tensor fp16 by calling `Tensor.half()`

```
torch.tensor(5.),torch.tensor(5.).half()
```

```
(tensor(5.), tensor(5., dtype=torch.float16))
```

Whole model half precision

We start by training our whole model using half precision. This includes converting our input data and out parameters to fp16.

```
class fp16Callback(Callback):
    def before_batch(self):
        self.learn.xb=(self.xb[0].half(),)
        self.learn.model=self.model.half()
    def after_batch(self):
        #fix loss for logging
        self.learn.loss=self.learn.loss.float()
```

```
with less_random():
    dls=create_dls()
    learn=Learner(dls,resnet18(),opt_func=SGD,cbs=[fp16Callback])
    learn.fit(6)
```

```
<IPython.core.display.HTML object>
```

epoch	train_loss	valid_loss	time
0	2.154748	2.054516	00:17

epoch	train_loss	valid_loss	time
1	1.803467	1.779198	00:15
2	1.646736	1.664151	00:15
3	1.547711	1.596234	00:15
4	1.487926	1.549276	00:15
5	1.433233	1.512745	00:15

How does this compare to training with fp32?

```
with less_random():
    dls=create_dls()
    learn=Learner(dls,resnet18(),opt_func=SGD)
    learn.fit(6)
```

```
<IPython.core.display.HTML object>
```

epoch	train_loss	valid_loss	time
0	2.084821	1.969784	00:15
1	1.742251	1.712177	00:14
2	1.575977	1.596902	00:14
3	1.465374	1.527497	00:14
4	1.398808	1.475065	00:15
5	1.337400	1.437802	00:14

Seems we are doing a bit worse…

Looking at the gradients.

We create a callback that collects the data needed to create a colorful deminsion graph of our gradients.

```
@patch
def after_backward(self:GradLogCallback):
    for n,p in learn.model.named_parameters():
        if p.numel()>10:
            if n not in self.log: self.log[n]=[]
            self.log[n]+=[p.cpu().abs().float().histc(100,0,1)]
```

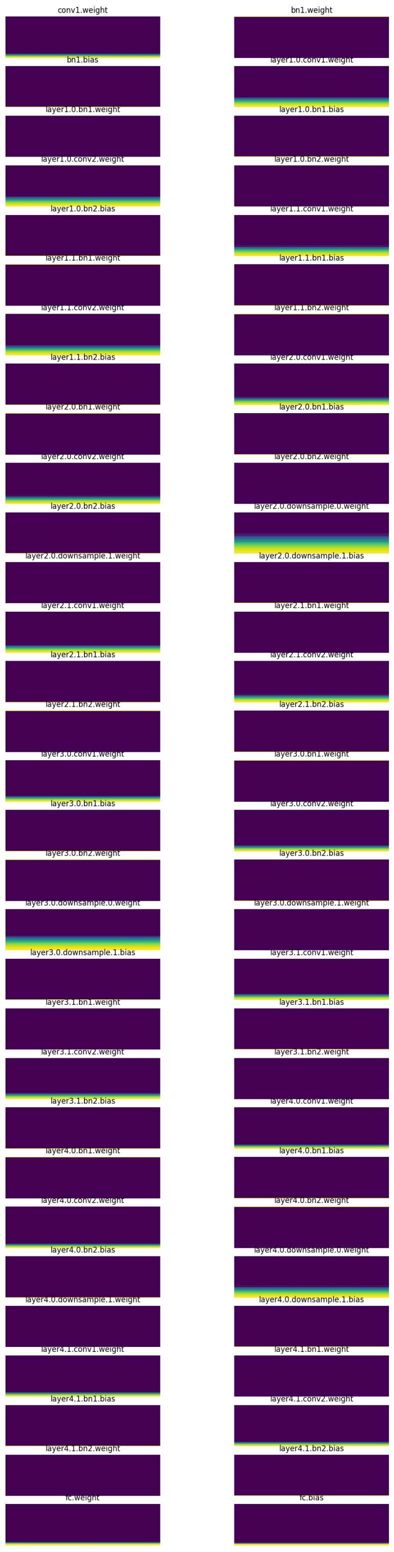
```python
with less_random():
    dls=create_dls()
    learn=Learner(dls,resnet18(),opt_func=SGD,cbs=[fp16Callback,GradLogCallback])
    learn.fit(6)
```

```
<IPython.core.display.HTML object>
```

epoch	train_loss	valid_loss	time
0	2.154748	2.054516	00:25
1	1.803467	1.779198	00:25
2	1.646736	1.664151	00:25
3	1.547711	1.596234	00:25
4	1.487926	1.549276	00:25
5	1.433233	1.512745	00:25

Most of the gradients are very close to zero. Just something to take note of, as the gradients don't have a standard deviation of 1 like the activations.

```python
show_colorful_grid(learn.cbs[-1].log)
```


conv1.weight
bn1.weight
bn1.bias
layer1.0.conv1.weight
layer1.0.bn1.weight
layer1.0.bn1.bias
layer1.0.conv2.weight
layer1.0.bn2.weight
layer1.0.bn2.bias
layer1.1.conv1.weight
layer1.1.bn1.weight
layer1.1.bn1.bias
layer1.1.conv2.weight
layer1.1.bn2.weight
layer1.1.bn2.bias
layer2.0.conv1.weight
layer2.0.bn1.weight
layer2.0.bn1.bias
layer2.0.conv2.weight
layer2.0.bn2.weight
layer2.0.bn2.bias
layer2.0.downsample.0.weight
layer2.0.downsample.1.weight
layer2.0.downsample.1.bias
layer2.1.conv1.weight
layer2.1.bn1.weight
layer2.1.bn1.bias
layer2.1.conv2.weight
layer2.1.bn2.weight
layer2.1.bn2.bias
layer3.0.conv1.weight
layer3.0.bn1.weight
layer3.0.bn1.bias
layer3.0.conv2.weight
layer3.0.bn2.weight
layer3.0.bn2.bias
layer3.0.downsample.0.weight
layer3.0.downsample.1.weight
layer3.0.downsample.1.bias
layer3.1.conv1.weight
layer3.1.bn1.weight
layer3.1.bn1.bias
layer3.1.conv2.weight
layer3.1.bn2.weight
layer3.1.bn2.bias
layer4.0.conv1.weight
layer4.0.bn1.weight
layer4.0.bn1.bias
layer4.0.conv2.weight
layer4.0.bn2.weight
layer4.0.bn2.bias
layer4.0.downsample.0.weight
layer4.0.downsample.1.weight
layer4.0.downsample.1.bias
layer4.1.conv1.weight
layer4.1.bn1.weight
layer4.1.bn1.bias
layer4.1.conv2.weight
layer4.1.bn2.weight
layer4.1.bn2.bias
fc.weight
fc.bias

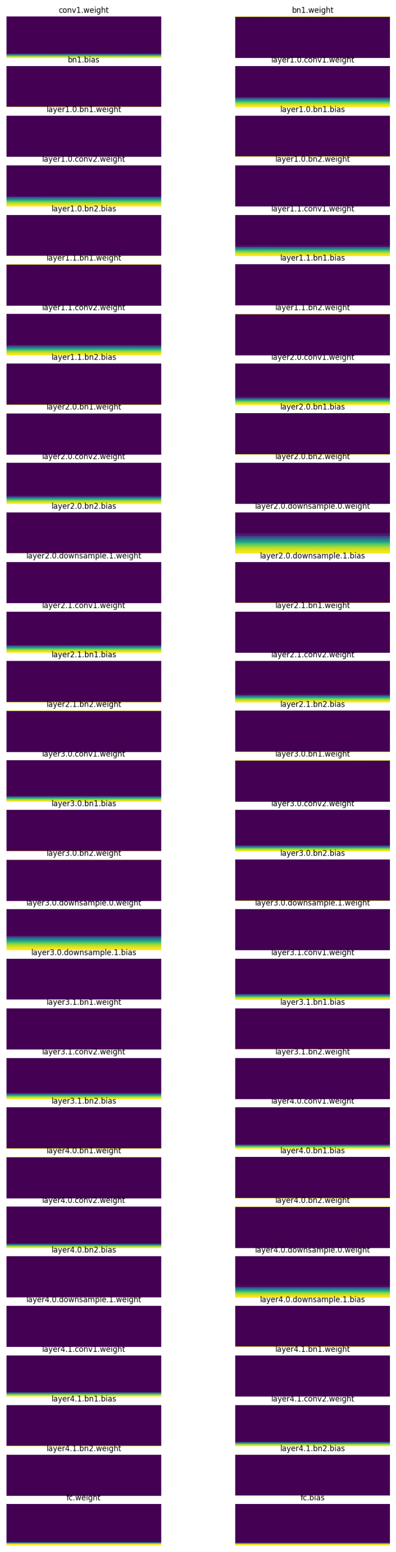
```
with less_random():
    dls=create_dls()
    learn=Learner(dls,resnet18(),opt_func=SGD,cbs=[GradLogCallback])
    learn.fit(6)
```

```
<IPython.core.display.HTML object>
```

epoch	train_loss	valid_loss	time
0	2.084821	1.969784	00:27
1	1.742251	1.712177	00:27
2	1.575977	1.596902	00:27
3	1.465374	1.527497	00:26
4	1.398808	1.475065	00:26
5	1.337400	1.437802	00:27

The fp32 gradients actually look fairly similar.

```
show_colorful_grid(learn.cbs[-1].log)
```


conv1.weight
bn1.weight
bn1.bias
layer1.0.conv1.weight
layer1.0.bn1.weight
layer1.0.bn1.bias
layer1.0.conv2.weight
layer1.0.bn2.weight
layer1.0.bn2.bias
layer1.1.conv1.weight
layer1.1.bn1.weight
layer1.1.bn1.bias
layer1.1.conv2.weight
layer1.1.bn2.weight
layer1.1.bn2.bias
layer2.0.conv1.weight
layer2.0.bn1.weight
layer2.0.bn1.bias
layer2.0.conv2.weight
layer2.0.bn2.weight
layer2.0.bn2.bias
layer2.0.downsample.0.weight
layer2.0.downsample.1.weight
layer2.0.downsample.1.bias
layer2.1.conv1.weight
layer2.1.bn1.weight
layer2.1.bn1.bias
layer2.1.conv2.weight
layer2.1.bn2.weight
layer2.1.bn2.bias
layer3.0.conv1.weight
layer3.0.bn1.weight
layer3.0.bn1.bias
layer3.0.conv2.weight
layer3.0.bn2.weight
layer3.0.bn2.bias
layer3.0.downsample.0.weight
layer3.0.downsample.1.weight
layer3.0.downsample.1.bias
layer3.1.conv1.weight
layer3.1.bn1.weight
layer3.1.bn1.bias
layer3.1.conv2.weight
layer3.1.bn2.weight
layer3.1.bn2.bias
layer4.0.conv1.weight
layer4.0.bn1.weight
layer4.0.bn1.bias
layer4.0.conv2.weight
layer4.0.bn2.weight
layer4.0.bn2.bias
layer4.0.downsample.0.weight
layer4.0.downsample.1.weight
layer4.0.downsample.1.bias
layer4.1.conv1.weight
layer4.1.bn1.weight
layer4.1.bn1.bias
layer4.1.conv2.weight
layer4.1.bn2.weight
layer4.1.bn2.bias
fc.weight
fc.bias

Looking near 0 for the Gradients

We look from 0 to 3 times the standard deviation of the first batch to get a closer look at the gradients

```
@patch
def after_backward(self:GradLogCallback):
    for n,p in learn.model.named_parameters():
        if p.numel()>10:
            if n not in self.log: self.log[n]=[3*p.cpu().abs().float().std()]
            self.log[n]+=[p.cpu().abs().float().histc(100,0,self.log[n][0].item())]
```

```
with less_random():
    dls=create_dls()
    learn=Learner(dls,resnet18(),opt_func=SGD,cbs=[fp16Callback,GradLogCallback])
    learn.fit(6)
```

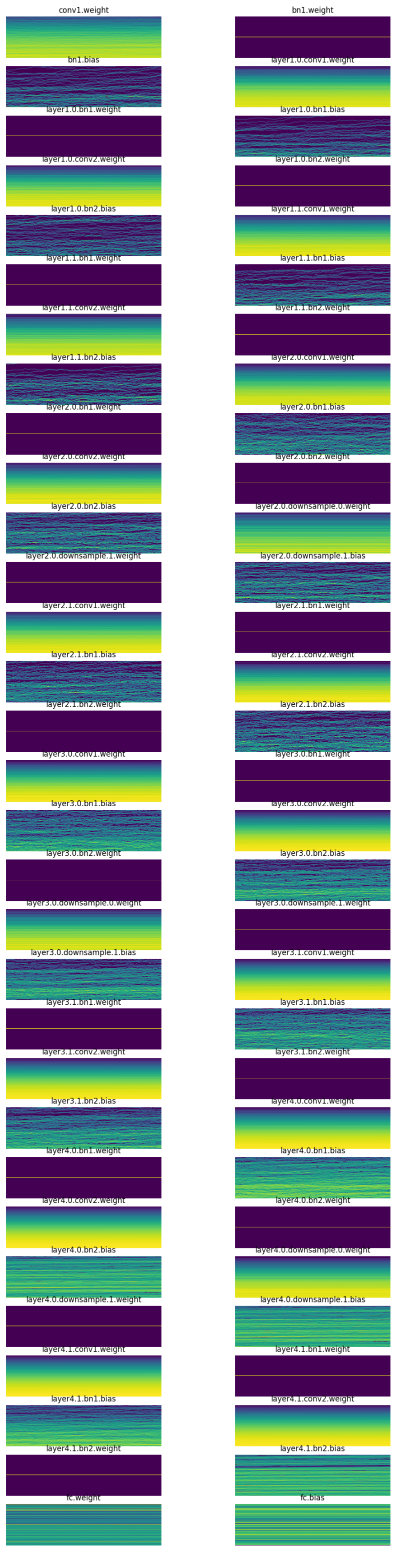
```
<IPython.core.display.HTML object>
```

epoch	train_loss	valid_loss	time
0	2.154748	2.054516	00:30
1	1.803467	1.779198	00:29
2	1.646736	1.664151	00:29
3	1.547711	1.596234	00:30
4	1.487926	1.549276	00:29
5	1.433233	1.512745	00:30

```
fp16_log=learn.grad_log.log
```

We can see something now! Pay a little bit of attention to the `bn.weight` graphs.

```
show_colorful_grid(learn.cbs[-1].log)
```


conv1.weight
bn1.weight
bn1.bias
layer1.0.conv1.weight
layer1.0.bn1.weight
layer1.0.bn1.bias
layer1.0.conv2.weight
layer1.0.bn2.weight
layer1.0.bn2.bias
layer1.1.conv1.weight
layer1.1.bn1.weight
layer1.1.bn1.bias
layer1.1.conv2.weight
layer1.1.bn2.weight
layer1.1.bn2.bias
layer2.0.conv1.weight
layer2.0.bn1.weight
layer2.0.bn1.bias
layer2.0.conv2.weight
layer2.0.bn2.weight
layer2.0.bn2.bias
layer2.0.downsample.0.weight
layer2.0.downsample.1.weight
layer2.0.downsample.1.bias
layer2.1.conv1.weight
layer2.1.bn1.weight
layer2.1.bn1.bias
layer2.1.conv2.weight
layer2.1.bn2.weight
layer2.1.bn2.bias
layer3.0.conv1.weight
layer3.0.bn1.weight
layer3.0.bn1.bias
layer3.0.conv2.weight
layer3.0.bn2.weight
layer3.0.bn2.bias
layer3.0.downsample.0.weight
layer3.0.downsample.1.weight
layer3.0.downsample.1.bias
layer3.1.conv1.weight
layer3.1.bn1.weight
layer3.1.bn1.bias
layer3.1.conv2.weight
layer3.1.bn2.weight
layer3.1.bn2.bias
layer4.0.conv1.weight
layer4.0.bn1.weight
layer4.0.bn1.bias
layer4.0.conv2.weight
layer4.0.bn2.weight
layer4.0.bn2.bias
layer4.0.downsample.0.weight
layer4.0.downsample.1.weight
layer4.0.downsample.1.bias
layer4.1.conv1.weight
layer4.1.bn1.weight
layer4.1.bn1.bias
layer4.1.conv2.weight
layer4.1.bn2.weight
layer4.1.bn2.bias
fc.weight
fc.bias

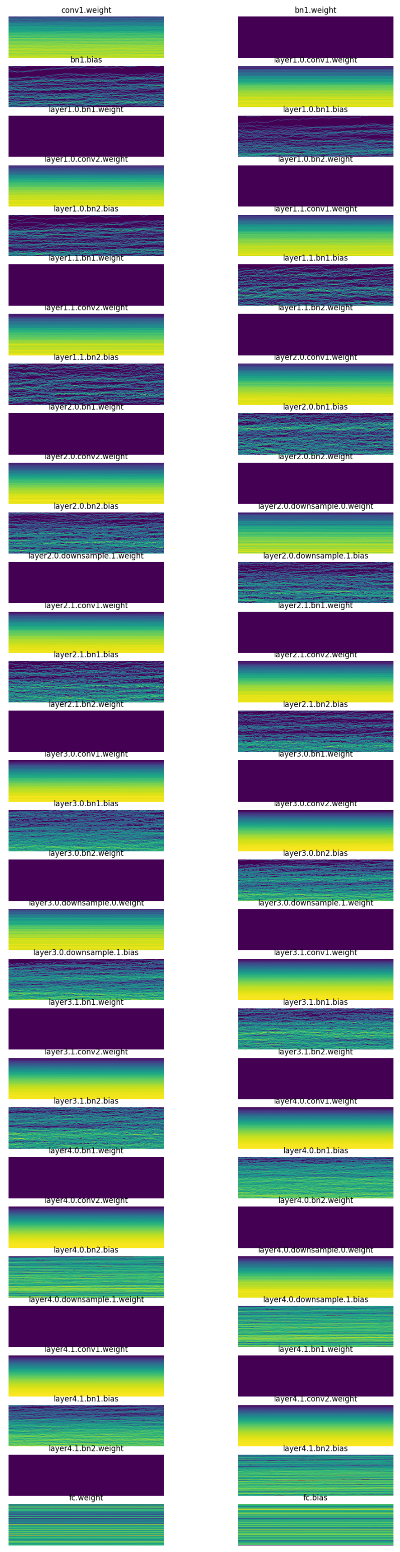
```
with less_random():
    dls=create_dls()
    learn=Learner(dls,resnet18(),opt_func=SGD,cbs=[GradLogCallback])
    learn.fit(6)
```

```
<IPython.core.display.HTML object>
```

epoch	train_loss	valid_loss	time
0	2.084821	1.969784	00:29
1	1.742251	1.712177	00:32
2	1.575977	1.596902	00:32
3	1.465374	1.527497	00:31
4	1.398808	1.475065	00:32
5	1.337400	1.437802	00:32

hm, these FP32 gradients mostly look the same, though we are not getting a horizontal line for the batchnorm weights.

```
show_colorful_grid(learn.cbs[-1].log)
```


conv1.weight
bn1.weight
bn1.bias
layer1.0.conv1.weight
layer1.0.bn1.weight
layer1.0.bn1.bias
layer1.0.conv2.weight
layer1.0.bn2.weight
layer1.0.bn2.bias
layer1.1.conv1.weight
layer1.1.bn1.weight
layer1.1.bn1.bias
layer1.1.conv2.weight
layer1.1.bn2.weight
layer1.1.bn2.bias
layer2.0.conv1.weight
layer2.0.bn1.weight
layer2.0.bn1.bias
layer2.0.conv2.weight
layer2.0.bn2.weight
layer2.0.bn2.bias
layer2.0.downsample.0.weight
layer2.0.downsample.1.weight
layer2.0.downsample.1.bias
layer2.1.conv1.weight
layer2.1.bn1.weight
layer2.1.bn1.bias
layer2.1.conv2.weight
layer2.1.bn2.weight
layer2.1.bn2.bias
layer3.0.conv1.weight
layer3.0.bn1.weight
layer3.0.bn1.bias
layer3.0.conv2.weight
layer3.0.bn2.weight
layer3.0.bn2.bias
layer3.0.downsample.0.weight
layer3.0.downsample.1.weight
layer3.0.downsample.1.bias
layer3.1.conv1.weight
layer3.1.bn1.weight
layer3.1.bn1.bias
layer3.1.conv2.weight
layer3.1.bn2.weight
layer3.1.bn2.bias
layer4.0.conv1.weight
layer4.0.bn1.weight
layer4.0.bn1.bias
layer4.0.conv2.weight
layer4.0.bn2.weight
layer4.0.bn2.bias
layer4.0.downsample.0.weight
layer4.0.downsample.1.weight
layer4.0.downsample.1.bias
layer4.1.conv1.weight
layer4.1.bn1.weight
layer4.1.bn1.bias
layer4.1.conv2.weight
layer4.1.bn2.weight
layer4.1.bn2.bias
fc.weight
fc.bias

```
fp32_log=learn.grad_log.log
```

One thing to note is that the minimum positive value for fp16 is a not as small as fp32.

```
torch.tensor(2**-24,dtype=torch.half),torch.tensor(2**-149)
```

```
(tensor(5.9605e-08, dtype=torch.float16), tensor(1.4013e-45))
```

hm, the batchnorm weight gradients standard deviations are **ZERO** for fp16!

```
[[k,fp16_log[k][0].item(),fp32_log[k][0].item()] for k in fp16_log if 'bn' in k]
```

```
[['bn1.weight', 0.0, 0.006029318086802959 4],
 ['bn1.bias', 0.004148084670305252, 0.003580566728487611],
 ['layer1.0.bn1.weight', 0.0, 0.005399828776717186],
 ['layer1.0.bn1.bias', 0.004158522468060255, 0.003490504808723926 5],
 ['layer1.0.bn2.weight', 0.0, 0.004458598792552948],
 ['layer1.0.bn2.bias', 0.0027511364314705133, 0.0024442975409328938],
 ['layer1.1.bn1.weight', 0.0, 0.0039586215279996395],
 ['layer1.1.bn1.bias', 0.002898486563935876, 0.002269925782456994],
 ['layer1.1.bn2.weight', 0.0, 0.0035925875417888165],
 ['layer1.1.bn2.bias', 0.001713279401883483, 0.0014464608393609524],
 ['layer2.0.bn1.weight', 0.0, 0.003240257501602173],
 ['layer2.0.bn1.bias', 0.0017084497958421707, 0.0014696172438561916],
 ['layer2.0.bn2.weight', 0.0, 0.002757459646090865],
 ['layer2.0.bn2.bias', 0.0018845133017748594, 0.0017825027462095022],
 ['layer2.1.bn1.weight', 0.0, 0.0026580658741295338],
 ['layer2.1.bn1.bias', 0.0015702887903898954, 0.0015195324085652828],
 ['layer2.1.bn2.weight', 0.0, 0.002109265886247158],
 ['layer2.1.bn2.bias', 0.0011786026880145073, 0.0011513214558362 96],
 ['layer3.0.bn1.weight', 0.0, 0.001799717196263373],
 ['layer3.0.bn1.bias', 0.0011105844751000404, 0.0009980872273445 13],
 ['layer3.0.bn2.weight', 0.0, 0.0017275793943554163],
 ['layer3.0.bn2.bias', 0.001101263682357967, 0.0009456288535147905],
 ['layer3.1.bn1.weight', 0.0, 0.001411611563526094],
 ['layer3.1.bn1.bias', 0.0008357313927263021, 0.0007302637677639723],
 ['layer3.1.bn2.weight', 0.0, 0.0011884564300999045],
 ['layer3.1.bn2.bias', 0.0006777657545171678, 0.0004925086977891624],
 ['layer4.0.bn1.weight', 0.0, 0.0009125272044911981],
 ['layer4.0.bn1.bias', 0.0005735242739319801, 0.0004978459910489619],
['layer4.0.bn2.weight', 0.0, 0.0017549480544403195],
['layer4.0.bn2.bias', 0.002376189222559333, 0.002254981081932783],
['layer4.1.bn1.weight', 0.0, 0.0008248933590948582],
['layer4.1.bn1.bias', 0.0006004928727634251, 0.0004860978224314749],
['layer4.1.bn2.weight', 0.0, 0.00195809337310493],
['layer4.1.bn2.bias', 0.0034636915661394596, 0.003284711856395006]]
```

Letting Batchnorm stay in FP32

We create a function here that just allows us to apply a funtion to both our module, and one of its parameters.

```
def apply_p(f,m):
    for module in m.children():
        apply_p(f,module)
        for n,p in module.named_parameters(recurse=False):
            f(module,n)
```

Here we allow BatchNorm weights to stay in fp32.

```
class fp16Callback(Callback):
    def before_fit(self):
        def f(m,n):
            getattr(m,n).data=getattr(m,n).data.float() if isinstance(m,nn.BatchNorm2d) el
        apply_p(f,self.learn.model)
    def before_batch(self):
        self.learn.xb=(self.xb[0].half(),)
    def after_pred(self):
        #loss should be calculated in fp32 as well
        self.learn.pred=self.learn.pred.float()
```

```
with less_random():
    dls=create_dls()
    learn=Learner(dls,resnet18(),opt_func=SGD,cbs=[fp16Callback,GradLogCallback])
    learn.fit(6)
```

```
<IPython.core.display.HTML object>
```

epoch	train_loss	valid_loss	time
0	2.160702	2.038649	00:29
1	1.807114	1.770107	00:29
2	1.642703	1.656874	00:28
3	1.542962	1.592482	00:29
4	1.486021	1.546489	00:29
5	1.425706	1.504864	00:29

hmm, we have slightly better results, but nothing too close to our fp32 model.

```
fp16_bn_log=learn.grad_log.log
```

Though, we have gotten the bn weights to have gradients!

```
[[k,fp16_log[k][0].item(),fp16_bn_log[k][0].item(),fp32_log[k][0].item()] for k in fp16_lo
```

```
[['bn1.weight', 0.0, 0.006746089085936546, 0.0060293180868029594],
 ['bn1.bias',
  0.004148084670305252,
  0.003786720335483551,
  0.003580566728487611],
 ['layer1.0.bn1.weight', 0.0, 0.005284080747514963, 0.005399828776717186],
 ['layer1.0.bn1.bias',
  0.004158522468060255,
  0.003640677547082305,
  0.0034905048087239265],
 ['layer1.0.bn2.weight', 0.0, 0.005027716979384422, 0.004458598792552948],
 ['layer1.0.bn2.bias',
  0.0027511364314705133,
  0.0025208923034369946,
  0.0024442975409328938],
 ['layer1.1.bn1.weight', 0.0, 0.00424396526068449, 0.0039586215279996395],
 ['layer1.1.bn1.bias',
  0.002898486563935876,
  0.002260061912238598,
  0.002269925782456994],
 ['layer1.1.bn2.weight', 0.0, 0.004007537383586168, 0.0035925875417888165],
 ['layer1.1.bn2.bias',
  0.001713279401883483,
  0.0016484896186739206,
  0.0014464608393609524],
['layer2.0.bn1.weight', 0.0, 0.0031620513182133436, 0.003240257501602173],
['layer2.0.bn1.bias',
 0.0017084497958421707,
 0.0016835747519508004,
 0.0014696172438561916],
['layer2.0.bn2.weight', 0.0, 0.0031509941909462214, 0.002757459646090865],
['layer2.0.bn2.bias',
 0.0018845133017748594,
 0.0020461969543248415,
 0.0017825027462095022],
['layer2.1.bn1.weight', 0.0, 0.003080494701862335, 0.0026580658741295338],
['layer2.1.bn1.bias',
 0.0015702887903898954,
 0.0016526294639334083,
 0.0015195324085652828],
['layer2.1.bn2.weight', 0.0, 0.0026030924636870623, 0.002109265886247158],
['layer2.1.bn2.bias',
 0.0011786026880145073,
 0.0013054630253463984,
 0.001151321455836296],
['layer3.0.bn1.weight', 0.0, 0.0020066993311047554, 0.001799717196263373],
['layer3.0.bn1.bias',
 0.0011105844751000404,
 0.0010541853262111545,
 0.000998087227344513],
['layer3.0.bn2.weight', 0.0, 0.0021246818359941244, 0.0017275793943554163],
['layer3.0.bn2.bias',
 0.001101263682357967,
 0.0010443663923069835,
 0.0009456288535147905],
['layer3.1.bn1.weight', 0.0, 0.001690064324066043, 0.001411611563526094],
['layer3.1.bn1.bias',
 0.0008357313927263021,
 0.0008858887013047934,
 0.0007302637677639723],
['layer3.1.bn2.weight', 0.0, 0.0014081220142543316, 0.0011884564300999045],
['layer3.1.bn2.bias',
 0.0006777657545171678,
 0.0006280804518610239,
 0.0004925086977891624],
['layer4.0.bn1.weight', 0.0, 0.0010809964733198285, 0.0009125272044911981],
['layer4.0.bn1.bias',
 0.0005735242739319801,
  0.0005982829607091844,
  0.0004978459910489619],
 ['layer4.0.bn2.weight', 0.0, 0.0018873803783208132, 0.0017549480544403195],
 ['layer4.0.bn2.bias',
  0.0023761892225593333,
  0.0023989235050976276,
  0.0022549810819327831],
 ['layer4.1.bn1.weight', 0.0, 0.0010339637519791722, 0.0008248933590948582],
 ['layer4.1.bn1.bias',
  0.0006004928727634251,
  0.0006056932033970952,
  0.0004860978224314749],
 ['layer4.1.bn2.weight', 0.0, 0.002108693355694413, 0.001958093373104930],
 ['layer4.1.bn2.bias',
  0.0034636915661394596,
  0.0034976028837263584,
  0.0032847118563950061]]
```

Looking at the gradients

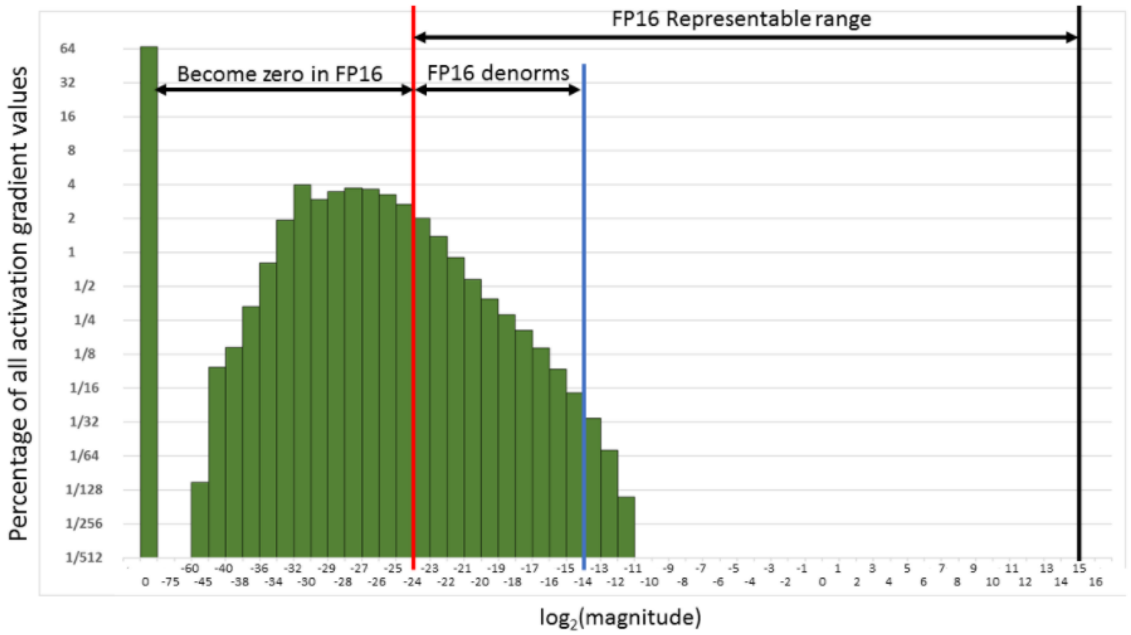


Figure 1: nv_grads.png

Image taken from here: https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html

Above we can see that some of the gradients will go to zero in fp16. This will happen more as the model trains and the gradients get smaller throughout training.

Here we are create seperate FP16 parameters for our model, while our optimizer uses the fp32 weights. We scale up the loss by **128** before our backwards pass, to increase the size of the gradients. We then descale it back **128** when converting back to fp32 for our optimizer step.

```
@patch
def before_fit(self:fp16Callback):
    self.opt_params= [p for p in self.model.parameters()]
    def f(m,n):
        value=getattr(m,n).data.float() if isinstance(m,nn.BatchNorm2d) else getattr(m,n).
        setattr(m,n,nn.Parameter(value))
    apply_p(f,self.learn.model)
@patch
def before_backward(self:fp16Callback):
    self.learn.loss_grad=128*self.learn.loss_grad
@patch
def after_backward(self:fp16Callback):
    for mp,op in zip(self.learn.model.parameters(),self.opt_params):
        op.grad=mp.grad.to(dtype=torch.float32)/128
@patch
def after_step(self:fp16Callback):
    for mp,op in zip(self.learn.model.parameters(),self.opt_params):
        mp.data= op.data.to(dtype=torch.float16) if(op.grad.dtype!=mp.grad.dtype) else op.
    self.learn.model.zero_grad()
```

```
with less_random():
    dls=create_dls()
    learn=Learner(dls,resnet18(),opt_func=SGD,cbs=[fp16Callback,GradLogCallback])
    learn.fit(6)
```

```
<IPython.core.display.HTML object>
```

epoch	train_loss	valid_loss	time
0	2.088479	1.967441	00:32
1	1.727627	1.706729	00:31
2	1.566641	1.595630	00:31

epoch	train_loss	valid_loss	time
3	1.463253	1.530451	00:31
4	1.398891	1.483463	00:32
5	1.338582	1.453968	00:32

And, now we have results comparable to fp32!

Can we do this in a non-manual way?

Well we start at the maximum possible multiple as the `loss_scale`. We multiply our gradients by this value. Yes, this value is definitely going to be too big so we need a strategy for decreasing the value now.

```
@patch
def before_fit(self:fp16Callback):
    self.loss_scale=2.**24
    self.count=0
    self.opt_params= [p for p in self.model.parameters()]
    def f(m,n):
        value=getattr(m,n).data.float() if isinstance(m,nn.BatchNorm2d) else getattr(m,n).
        setattr(m,n,nn.Parameter(value))
    apply_p(f,self.learn.model)
@patch
def before_backward(self:fp16Callback):
    self.learn.loss_grad=self.loss_scale*self.learn.loss_grad
```

This gets really messy, but we decrease by half if we overflow. We then skip the current batch and go to the next batch. This does mean we will hit a lot of skipped batches in the beginning of training. As our model trains, our gradients will probably get smaller, so we will want to increase our `loss_scale`. For this example we just test every 500 batches to see if we should increase our `loss_scale`.

```
@patch
def after_backward(self:fp16Callback):
    for mp in self.learn.model.parameters():
        if mp.grad is not None and test_overflow(mp.grad.data):
            self.learn.model.zero_grad()
            self.loss_scale/=2
            raise CancelBatchException()
    for mp,op in zip(self.learn.model.parameters(),self.opt_params):
       op.grad=mp.grad.to(dtype=torch.float32)/self.loss_scale
   self.count += 1
   if self.count == 500:
       self.count = 0
       self.loss_scale *= 2
```

```
with less_random():
    dls=create_dls()
    learn=Learner(dls,resnet18(),opt_func=SGD,cbs=[fp16Callback,GradLogCallback])
    learn.fit(6)
```

```
<IPython.core.display.HTML object>
```

epoch	train_loss	valid_loss	time
0	2.094611	1.968921	00:33
1	1.729887	1.698773	00:33
2	1.563014	1.592412	00:33
3	1.455867	1.526811	00:34
4	1.397211	1.486772	00:34
5	1.330738	1.447847	00:34

You can look more into all of this by checking out `NonNativeMixedPrecision`.