Why do we need to set the gradients manually to zero in pytorch?

pinocchio Rene Sandoval Jul '17

Why do we need to set the gradients manually to zero in pytorch? e.g:

`w1.grad.data.zero_()`

why do we need that? What happens if we don't use that?

It feels that needing to ask this question means there is something conceptual/fundamental about the design of pytorch that I dont understand.

LSTM CTC model not learning in PyTorch

ruotianluo Ruotian(RT) Luo Jul '17

Every time a variable is back propogated through, the gradient will be accumulated instead of being replaced. (This makes it easier for rnn, because each module will be back propogated through several times.)

pinocchio Rene Sandoval Jul '17

@ruotianluo I still don't understand. Though, is there a link I can just read to understand this?

Is there no link to understand how pytorch works and so I can form a mental model of it?

Like something like this seems very strange to someone coming from tensorflow.

samarth-robo Samarth Brahmbhatt Jul '17

Since the `backward()` function accumulates gradients, and you don't want to mix up gradients between minibatches, you have to zero them out at the start of a new minibatch. This is exactly like how a general (additive) accumulator variable is initialized to 0 in code.

By the way, the best practice is to use the **zero_grad()** function on the optimizer.

tom Thomas V Jul '17

A more explicit example in a similar direction as **@ruotianluo** is the ability to add gradients from several forward passes, for example in GANs:

> **How to use the backward functions for multiple losses?**
>
> Hello **@Nabarun_Goswami** , to try to clear this up, in the DCGAN example, you have (think about mathematical functions here, I left out everything not relevant). loss = criterion(netD(real, params))+criterion(netD(fake, params)) Spelling out the chain rule for the gradient of the loss w.r.t. the params: ∇params loss = ∇params netD(real, params)* ∇netD loss(netD (real,params)) + ∇params netD(fake, params)* ∇netD loss(netD (fake,params)), note how ∇params netD is evaluated at two different poin…

If you wanted, you could also achieve minibatches that are larger than fit in your memory by combining several sub-minibatches into one gradient step, but I have not really seen that done.

Best regards

Thomas

pinocchio Rene Sandoval Jul '17

I think I don't even know what "accumulating gradients" even means though. So I'm not sure what we are even talking about.

hughperkins Jul '17

You mean like, why doesnt `backward` just zero out the gradients before doing the back propagation? I somewhat agree on this point actually… I would have to really struggle to think of a time when I've called `backward` without first zeroing out the gradients.

hughperkins **Jul '17**

(like, it seems like we could have an option in backward to not zero out the gradients, like `backward(preserve_grads=True)`, but by default, seems like zeroing out the gradients could be the default action)

How to get gradient of a tensor wrt different losses

albanD **Jul '17**

Hi,

I think the big difference with tensorflow is the following.
Since you use a static graph, you define exactly what should be done to make one gradient computation/update. And then you just tell it to do it using a given input/target.

In pytorch, it is significantly more flexible as the autograd engine will just "remember" how to compute the gradient for a given variable while you are performing computations with this Variable. This means that you can get the gradients wrt a variable, then perform computation with it again, then recompute gradients corresponding to these new operations.
In this scheme, there is a not a single point where you stop performing "forward" operations and you know that the only thing that is left to be done is compute the gradients. So it is trickier to automatically set the gradients to 0 because you don't know when a computation end, and when a new starts.

An example where the gradient accumulation is useful is for example if you share some part of a network for two different tasks:

```
input = Variable(data)
# Get the features
features = feature_extractor(input)

# Compute first loss and get the gradients for it
loss1 = task1(features)
loss1.backward(retain_graph=True)
# This add the gradients wrt loss1 in both the "task1" net and the "feature_extract
# So each parameter "w" in "feature_extractor" has it gradient d(loss1)/dw

# Perform the second task and get the gradients for it as well
```

```
loss2 = task2(features)
loss2.backward()
# This will add gradients in "task2" and accumulate in "feature_extractor"
# Now each parameter in "feature_extractor" contains d(loss1)/dw + d(loss2)/dw
```

So the fact that the gradients are accumulated allows you to get the correct gradient for all the computations that you do with a given Variable even if you use it at multiple places in convoluted ways. The drawback here is that you have to manually reset the values to 0 so that the gradients computed previously do not interfere with the ones you are currently computing.

Why is "accumulating" the default mode of .gradient?

pinocchio Rene Sandoval **Jul '17**

thats interesting. I would have thought that the “sharing” would have been multiplicative since RNNs a composition of functions and not a mere addition (as in ur transfer learning example). I think I am more confused. 😦

albanD **Jul '17**

The addition comes from the rules of differentiation. Iif `f = f1 + f2`, then the gradients for a parameter in both branches is the sum of the contributions of each branch.

Jiawei_Zhuang **Oct '17**

`y.backward()` doesn’t just assign the value of y’(x) to `x.grad` (say `y` depends on `x`). It actually adds y’(x) to the current value of `x.grad` (think it as `x.grad += true_gradient`).

In the following example, `y.backward()` is called 5 times, so the final value of `x.grad` will be 5*cos(0)=5.

```
import torch
from torch.autograd import Variable

x = Variable(torch.Tensor([[0]]), requires_grad=True)

for t in range(5):
    y = x.sin()
    y.backward()

print(x.grad) # shows 5
```

Calling `x.grad.data.zero_()` before `y.backward()` can make sure `x.grad` is exactly the same as current y'(x), not a sum of y'(x) in all previous iterations.

```
x = Variable(torch.Tensor([[0]]), requires_grad=True)

for t in range(5):
    if x.grad is not None:
        x.grad.data.zero_()
    y = x.sin()
    y.backward()

print(x.grad) # shows 1
```

I also got confused by this "zeroing gradient" when first learning pytorch. The doc of `torch.autograd.backward` does mention that

> This function accumulates gradients in the leaves - you might need to zero them before calling it.

But this is quite hard to find and pretty confusing for (say) tensorflow users.

Official tutorials like **60 Minute Blitz** or **PyTorch with Examples** both say nothing about why one needs to call `grad.data.zero_()` during training. I think it would be useful to explain this a little more in beginner-level tutorials. RNN is a good example for why accumulating gradient (instead of refreshing) is useful, but I guess new users wouldn't even know that `backward()` is accumulating gradient 😅

---

jdhao **Nov '17**

@tom , since it is possible to accumulate the loss of several minibatch and do one parameter update. For example I want to update the parameter every 64 minibatch, I have the following code

```
total_loss = Variable(torch.zeros(1), requires_grad=True)

for idx, (data, target) in train_loader:

    data, target = Variable(data), Variable(target)
    output = model(data)
    loss = criterion(output, target)
    total_loss = total_loss + loss

    if (idx+1)%64 == 0:
        total_loss = total_loss/(64*batchsize)
        total_loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        total_loss = Variable(torch.zeros(1), requires_grad=True)
```

Is the above code correct to achieve the desired effect?

Question on Pytorch Tutorials about RNN and LSTM

tom Thomas V **Nov '17**

I'd do the backward on the (reweighted) loss in each run and not do total loss.

Best regards

Thomas

jdhao **Nov '17**

I find that doing the backward for each reweighted loss is much slower than accumulate the loss and then backward.

tom Thomas V **Nov '17**

Thanks for sharing this insight. I would have thought that you get similar speed at much less memory with separate backward calls. But clearly, the experiment proves my modest intuition to be wrong.

Best regards

Thomas

jdhao **Nov '17**

@albanD , I have some doubt about what the computation graph looks like in the case that we accumulate the loss manually.

Assume there are total 256 batches for the dataset. For the normal case (case1), for each batch in an epoch, a new computation graph is created and after the backward pass, the graph is freed. So 256 computation graphs are created and freed during one epoch.

In this case(case2), since we only do backward on every 64 batches. Does that mean only 4 graphs are created? The graph are created by composing 64 smaller graphs in case1 and the root node in the bigger graph is `total_loss`. The 64 smaller graph all have the same set of learnable parameters. If that is the case, the bigger graph will consume a lot of memory since it have 64 copies of the small graph.

Is that right? Do you have any ideas?

---

albanD **Nov '17**

Hi,

Indeed, in one case, you will create 256 graphs that work with one input.
In the second case, you will create only 4 graphs. but each of these 4 graphs is actually composed of 64 times the graph above and some `Add` operations at the end that sum the loss.

Indeed, in the second case you will use much more memory. Indeed, for the 64 iterations, you will create a single graph that just keep growing, and so you will use more and more memory.

Question on generator.zero_grad() in DCGAN

---

jdhao **Nov '17**

So We have to make sure that batchsize is not too large, or we will run out of memory.

---

albanD **Nov '17**

Here are three equivalent code, with different runtime/memory comsumption.
Assume that you want to run sgd with a batch size of 100.
(I didn’t run the code below there might be some typos, sorry in advance)

1: single batch of 100 (least runtime, more memory)

```
# some code
# Initialize dataset with batch size 100
for input, target in dataset:
    pred = net(input)
    loss = crit(pred, target)
    # one graph is created here
    opt.zero_grad()
    loss.backward()
    # graph is cleared here
    opt.step()
```

2: multiple small batches of 10 (more runtime, least memory)

```
# some code
# Initialize dataset with batch size 10
opt.zero_grad()
for i, (input, target) in enumerate(dataset):
    pred = net(input)
    loss = crit(pred, target)
    # one graph is created here
    loss.backward()
    # graph is cleared here
    if (i+1)%10 == 0:
        # every 10 iterations of batches of size 10
        opt.step()
        opt.zero_grad()
```

3: accumulate loss for multiple batches (more runtime, more memory)

```
# some code
# Initialize dataset with batch size 10
loss = 0
for i, (input, target) in enumerate(dataset):
    pred = net(input)
    current_loss = crit(pred, target)
    # current graph is appended to existing graph
    loss = loss + current_loss
    if (i+1)%10 == 0:
        # every 10 iterations of batches of size 10
        opt.zero_grad()
        loss.backward()
        # huge graph is cleared here
        opt.step()
```

It should be clear that case 3 is not what you want.
The choice between case 1 and 2 is a tradeoff between memory and speed so that depends on what you want to do.
Note that if you can fit a batch size of 50 in your memory, you can do a variation of case 2 with batch size of 50 and update every 2 iterations.

Accumulating Gradients

How to create a dataloader with variable-size input

How to increase the batch size but keep the gpu memory

Backpropagating through multiple optimizer steps

Getting different values for single batch vs accumulating gradients

7 more