

Pytorch, what are the gradient arguments

Asked 5 years, 6 months ago Modified 10 months ago Viewed 26k times



I am reading through the documentation of PyTorch and found an example where they write

134



```
gradients = torch.FloatTensor([0.1, 1.0, 0.0001])
y.backward(gradients)
print(x.grad)
```



where x was an initial variable, from which y was constructed (a 3-vector). The question is, what are the 0.1, 1.0 and 0.0001 arguments of the gradients tensor ? The documentation is not very clear on that.

[neural-network](#) [gradient](#) [pytorch](#) [torch](#) [gradient-descent](#) [Edit tags](#)

[Share](#) [Edit](#) [Follow](#) [Close](#) [Flag](#)

edited Apr 8, 2019 at 16:00

asked Apr 17, 2017 at 12:04



[kmario23](#)

53k

13

148

146



[Qubix](#)

3,893

7

28

65

6 Answers

Sorted by:

[Reset to default](#)

Date modified (newest first)



The original code I haven't found on PyTorch website anymore.

22



```
gradients = torch.FloatTensor([0.1, 1.0, 0.0001])
y.backward(gradients)
print(x.grad)
```



The problem with the code above is there is no function based on how to calculate the gradients. This means we don't know how many parameters (arguments the function takes) and the dimension of parameters.

To fully understand this I created an example close to the original:

Example 1:

```
a = torch.tensor([1.0, 2.0, 3.0], requires_grad = True)
b = torch.tensor([3.0, 4.0, 5.0], requires_grad = True)
c = torch.tensor([6.0, 7.0, 8.0], requires_grad = True)

y=3*a + 2*b*b + torch.log(c)
```

```

gradients = torch.FloatTensor([0.1, 1.0, 0.0001])
y.backward(gradients,retain_graph=True)

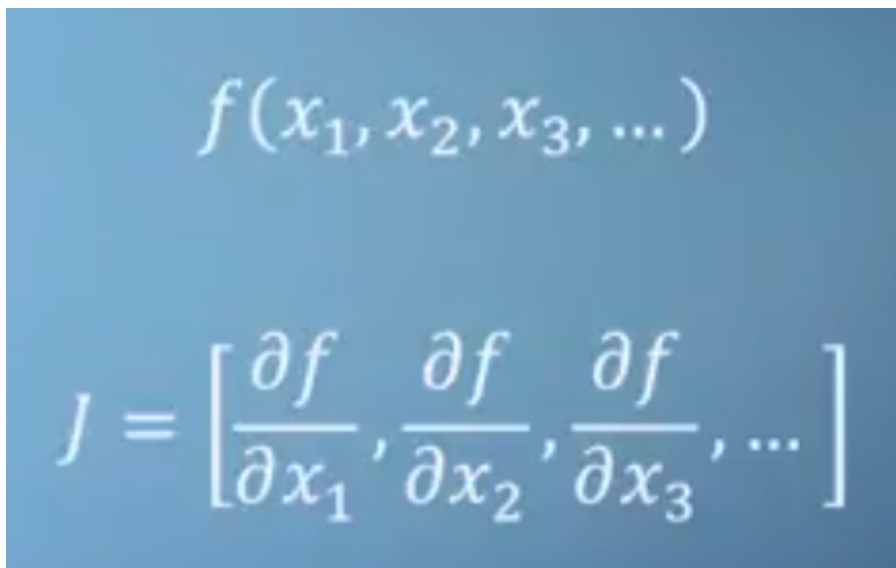
print(a.grad) # tensor([3.0000e-01, 3.0000e+00, 3.0000e-04])
print(b.grad) # tensor([1.2000e+00, 1.6000e+01, 2.0000e-03])
print(c.grad) # tensor([1.6667e-02, 1.4286e-01, 1.2500e-05])

```

I assumed our function is $y=3*a + 2*b*b + \text{torch.log}(c)$ and the parameters are tensors with three elements inside.

You can think of the `gradients = torch.FloatTensor([0.1, 1.0, 0.0001])` like this is the accumulator.

As you may hear, PyTorch autograd system calculation is equivalent to Jacobian product.



$$f(x_1, x_2, x_3, \dots)$$

$$J = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial x_3}, \dots \right]$$

In case you have a function, like we did:

```
y=3*a + 2*b*b + torch.log(c)
```

Jacobian would be $[3, 4*b, 1/c]$. However, this [Jacobian](#) is not how PyTorch is doing things to calculate the gradients at a certain point.

PyTorch uses forward pass and [backward mode automatic differentiation](#) (AD) in tandem.

There is no symbolic math involved and no numerical differentiation.

Numerical differentiation would be to calculate $\delta y / \delta b$, for $b=1$ and $b=1+\epsilon$ where ϵ is small.

If you don't use gradients in `y.backward()` :

Example 2

```

a = torch.tensor(0.1, requires_grad = True)
b = torch.tensor(1.0, requires_grad = True)
c = torch.tensor(0.1, requires_grad = True)
y=3*a + 2*b*b + torch.log(c)

y.backward()

print(a.grad) # tensor(3.)
print(b.grad) # tensor(4.)
print(c.grad) # tensor(10.)

```

You will simply get the result at a point, based on how you set your `a`, `b`, `c` tensors initially.

Be careful how you initialize your `a`, `b`, `c`:

Example 3:

```

a = torch.empty(1, requires_grad = True, pin_memory=True)
b = torch.empty(1, requires_grad = True, pin_memory=True)
c = torch.empty(1, requires_grad = True, pin_memory=True)

y=3*a + 2*b*b + torch.log(c)

gradients = torch.FloatTensor([0.1, 1.0, 0.0001])
y.backward(gradients)

print(a.grad) # tensor([3.3003])
print(b.grad) # tensor([0.])
print(c.grad) # tensor([inf])

```

If you use `torch.empty()` and don't use `pin_memory=True` you may have different results each time.

Also, note gradients are like accumulators so zero them when needed.

Example 4:

```

a = torch.tensor(1.0, requires_grad = True)
b = torch.tensor(1.0, requires_grad = True)
c = torch.tensor(1.0, requires_grad = True)
y=3*a + 2*b*b + torch.log(c)

y.backward(retain_graph=True)
y.backward()

print(a.grad) # tensor(6.)
print(b.grad) # tensor(8.)
print(c.grad) # tensor(2.)

```

Lastly few tips on terms PyTorch uses:

PyTorch creates a **dynamic computational graph** when calculating the gradients in forward pass. This looks much like a tree.

So you will often hear the *leaves* of this tree are **input tensors** and the *root* is **output tensor**.

Gradients are calculated by tracing the graph from the root to the leaf and multiplying every gradient in the way using the **chain rule**. This multiplying occurs in the backward pass.

Back some time I created [PyTorch Automatic Differentiation tutorial](#) that you may check interesting explaining all the tiny details about AD.

Share Edit Follow Flag

edited Dec 3, 2021 at 21:55

answered Aug 17, 2019 at 22:25



prosti

37.8k

10

169

144

-
- 1 Great answer! However, I don't think Pytorch does numerical differentiation ("For the previous function PyTorch would do for instance $\delta y / \delta b$, for $b=1$ and $b=1+\epsilon$ where ϵ is small. So there is nothing like symbolic math involved.") - I believe it does automatic differentiation. – [max_max_mir](#) Jan 27, 2020 at 4:12
-
- 1 Yes, it uses AD, or automatic differentiation, later I investigated AD further like in this [PDE](#), however, when I set this answer I was not quite informed. – [prosti](#) Jan 27, 2020 at 13:25
-
- E.g. example 2 gives RuntimeError: Mismatch in shape: grad_output[0] has a shape of torch.Size([3]) and output[0] has a shape of torch.Size([]). – [Andreas K.](#) Jul 5, 2020 at 21:10
-
- 1 @AndreasK., you were right, PyTorch introduced recently [zero sized tensors](#) and this had the impact on my previous examples. Removed since these examples were not crucial. – [prosti](#) Jul 11, 2020 at 12:54
-

Explanation

117



For neural networks, we usually use `loss` to assess how well the network has learned to classify the input image (or other tasks). The `loss` term is usually a scalar value. In order to update the parameters of the network, we need to calculate the gradient of `loss` w.r.t to the parameters, which is actually `leaf node` in the computation graph (by the way, these parameters are mostly the weight and bias of various layers such Convolution, Linear and so on).

According to chain rule, in order to calculate gradient of `loss` w.r.t to a leaf node, we can compute derivative of `loss` w.r.t some intermediate variable, and gradient of intermediate variable w.r.t to the leaf variable, do a dot product and sum all these up.

The `gradient` arguments of a `Variable`'s [backward\(.\)](#) method is used to **calculate a weighted sum of each element of a Variable w.r.t the leaf Variable**. These weight is just the derivate of final `loss` w.r.t each element of the intermediate variable.

A concrete example

Let's take a concrete and simple example to understand this.

```

from torch.autograd import Variable
import torch
x = Variable(torch.FloatTensor([[1, 2, 3, 4]]), requires_grad=True)
z = 2*x
loss = z.sum(dim=1)

# do backward for first element of z
z.backward(torch.FloatTensor([[1, 0, 0, 0]]), retain_graph=True)
print(x.grad.data)
x.grad.data.zero_() #remove gradient in x.grad, or it will be accumulated

# do backward for second element of z
z.backward(torch.FloatTensor([[0, 1, 0, 0]]), retain_graph=True)
print(x.grad.data)
x.grad.data.zero_()

# do backward for all elements of z, with weight equal to the derivative of
# loss w.r.t z_1, z_2, z_3 and z_4
z.backward(torch.FloatTensor([[1, 1, 1, 1]]), retain_graph=True)
print(x.grad.data)
x.grad.data.zero_()

# or we can directly backprop using loss
loss.backward() # equivalent to loss.backward(torch.FloatTensor([1.0]))
print(x.grad.data)

```

In the above example, the outcome of first print is

```

2 0 0 0
[torch.FloatTensor of size 1x4]

```

which is exactly the derivative of z_1 w.r.t to x .

The outcome of second print is :

```

0 2 0 0
[torch.FloatTensor of size 1x4]

```

which is the derivative of z_2 w.r.t to x .

Now if use a weight of $[1, 1, 1, 1]$ to calculate the derivative of z w.r.t to x , the outcome is

$1 \cdot dz_1/dx + 1 \cdot dz_2/dx + 1 \cdot dz_3/dx + 1 \cdot dz_4/dx$. So no surprisingly, the output of 3rd print is:

```

2 2 2 2
[torch.FloatTensor of size 1x4]

```

It should be noted that weight vector $[1, 1, 1, 1]$ is exactly derivative of $loss$ w.r.t to z_1, z_2, z_3 and z_4 . The derivative of $loss$ w.r.t to x is calculated as:

$$d(loss)/dx = d(loss)/dz_1 * dz_1/dx + d(loss)/dz_2 * dz_2/dx + d(loss)/dz_3 * dz_3/dx + d(loss)/dz_4 * dz_4/dx$$

So the output of 4th `print` is the same as the 3rd `print` :

```
2 2 2 2
[torch.FloatTensor of size 1x4]
```

Share Edit Follow Flag

edited Mar 19, 2019 at 2:53

answered Oct 31, 2017 at 2:03



eric
1,009 1 8 9



jdhaio
20.2k 13 122 232

-
- 1 just a doubt, why are we calculating `x.grad.data` for gradients for loss or `z`. – [Priyank Pathak](#) Jun 8, 2018 at 16:58
-
- 11 Maybe I missed something, but I feel like the official documentation really could have explained the `gradient` argument better. Thanks for your answer. – [protagonist](#) Aug 24, 2018 at 3:57
-
- 6 @jdhaio "It should be noted that weight vector `[1, 1, 1, 1]` is exactly derivative of `Loss` w.r.t to `z_1`, `z_2`, `z_3` and `z_4`." I think this statement is really key to the answer. When looking at the OP's code a big question mark is where do these [arbitrary\(magic\) numbers](#) for the gradient come from. In your concrete example I think it would be very helpful to point out the relation between the e.g. `[1, 0, 0 0]` tensor and the `loss` function right away so one can see that the values aren't arbitrary in this example. – [a_guest](#) Sep 3, 2018 at 9:13
-
- 2 @smwikipedia, that is not true. If we expand `loss = z.sum(dim=1)`, it will become `loss = z_1 + z_2 + z_3 + z_4`. If you know simple calculus, you will know that the derivative of `loss` w.r.t to `z_1`, `z_2`, `z_3`, `z_4` is `[1, 1, 1, 1]`. – [jdhaio](#) Feb 19, 2019 at 9:17
-
- 4 I love you. Solved my doubt! – [Black Jack 21](#) Jul 21, 2019 at 5:18
-

|



55



Typically, your computational graph has one scalar output says `loss`. Then you can compute the gradient of `loss` w.r.t. the weights (`w`) by `loss.backward()`. Where the default argument of `backward()` is `1.0`.

If your output has multiple values (e.g. `loss=[loss1, loss2, loss3]`), you can compute the gradients of loss w.r.t. the weights by `loss.backward(torch.FloatTensor([1.0, 1.0, 1.0]))`.

Furthermore, if you want to add weights or importances to different losses, you can use `loss.backward(torch.FloatTensor([-0.1, 1.0, 0.0001]))`.

This means to calculate $-0.1 \cdot d(\text{loss1})/dw$, $d(\text{loss2})/dw$, $0.0001 \cdot d(\text{loss3})/dw$ simultaneously.

Share Edit Follow Flag

edited Aug 20, 2017 at 7:23

answered Apr 19, 2017 at 9:26



Meta Fan
1,515 2 16 26

-
- 2 "if you want to add weights or importances to different losses, you can use `loss.backward(torch.FloatTensor([-0.1, 1.0, 0.0001]))`." -> This is true but somewhat misleading because the main reason why we pass `grad_tensors` is not to weigh them differently but they are gradients w.r.t. each element of corresponding tensors. – [aerin](#) Jan 12, 2019 at 23:26
-

Here, the output of forward(), i.e. y is a 3-vector.

30

The three values are the gradients at the output of the network. They are usually set to 1.0 if y is the final output, but can have other values as well, especially if y is part of a bigger network.

For eg. if x is the input, $y = [y_1, y_2, y_3]$ is an intermediate output which is used to compute the final output z,

Then,

$$dz/dx = dz/dy_1 * dy_1/dx + dz/dy_2 * dy_2/dx + dz/dy_3 * dy_3/dx$$

So here, the three values to backward are

$[dz/dy_1, dz/dy_2, dz/dy_3]$

and then backward() computes dz/dx

Share Edit Follow Flag

answered Apr 17, 2017 at 23:22



greenberet123

1,271 1 11 20

5 Thanks for the answer but how is this useful in practice? I mean where do we need $[dz/dy_1, dz/dy_2, dz/dy_3]$ other than hardcoding backprop? – hi15 May 21, 2017 at 21:07

Is it correct to say that the provided gradient argument is the gradient computed in the latter part of the network? – Khanetor Feb 2, 2018 at 21:31

This post is hidden. It was [deleted](#) 3 years ago by [Baum mit Augen](#) ♦.

-1

Check out this medium post on how pytorch backward() function works:

<https://medium.com/@mustafaghali11/how-pytorch-backward-function-works-55669b3b7c62>

Share Edit Follow **Undelete** Flag

answered Mar 24, 2019 at 21:37



Mustafa Alghali

51 1 3

2 A link to a solution is welcome, but please ensure your answer is useful without it: [add context around the link](#) so your fellow users will have some idea what it is and why it's there, then quote the most relevant part of the page you're linking to in case the target page is unavailable. [Answers that are little more than a link may be deleted.](#) – Zoe stands with Ukraine ♦ Mar 24, 2019 at 21:38

Comments disabled on deleted / locked posts / reviews

 This post is hidden. It was [deleted](#) 4 years ago by the post author.

-1

EDIT: Don't read this, I'm probably wrong, I'm still figuring it out...

I may have come across an example in which passing values to the gradient would be useful : when a part of your pipeline cannot be computed with PyTorch. If that is the case, you have to compute the gradient manually, and then you need to plug the gradient chain back together for that special part. Imagine this pipeline :

```
y = f(x)
z = g(y)
w = h(z)
```

But the function `g` has operations in it not supported by Pytorch so you can't have the automatic gradient of it, but you know how to compute it yourself. To have `w`'s gradient w.r.t `x`, you'll do:

```
x = Variable(torch.FloatTensor([1]),requires_grad=True)
y = f(x) # y is a Variable
np_z = g(y.data.numpy()) # Pass y as a Numpy array, on which you can apply g.
z = Variable(torch.FloatTensor(np_z),requires_grad=True)
w = h(z) # w is a Variable

# Get the grad through with chain rule
# dw/dx = dw/dz * dz/dy * dy/dx
w.backward() # Pass no arguments because z is scalar, but you could pass ones if it was
a vector
w_grad_z = z.grad.data
z_grad_y = grad_g(y.data.numpy()) # Your function that computes g's gradient
y.backward(w_grad_z * z_grad_y) # Pass the manually-computed gradient.
w_grad_x = x.grad # You directly get the gradient of w w.r.t to x
```

An example of such a `g` function could be a function that you didn't write yourself, and that accepts only Numpy arrays.

Share Edit Follow **Undelete** Flag [edited Dec 8, 2017 at 13:09](#)

answered Dec 1, 2017 at 8:37



[matthieu](#)

1,257 1 10 31



You must delete it! – [blitu12345](#) Dec 20, 2017 at 7:39

Comments disabled on deleted / locked posts / reviews