```
import torch

a = torch.randn((3,3), requires_grad = True)

w1 = torch.randn((3,3), requires_grad = True)
w2 = torch.randn((3,3), requires_grad = True)
w3 = torch.randn((3,3), requires_grad = True)
w4 = torch.randn((3,3), requires_grad = True)

b = w1*a
c = w2*a

d = w3*b + w4*c

L = (10 - d)

L.backward()
```

Running the above snippet results in the following error.

```
RuntimeError: grad can be implicitly created only for scalar outputs
```

This is because gradients can be computed with respect to scalar values by definition. You can't exactly differentiate a vector with respect to another vector. The mathematical entity used for such cases is called a **Jacobian,** the discussion of which is beyond the scope of this article.

There are two ways to overcome this.

If you just make a small change in the above code setting `L` to be the sum of all the errors, our problem will be solved.

```
import torch

a = torch.randn((3,3), requires_grad = True)

w1 = torch.randn((3,3), requires_grad = True)
w2 = torch.randn((3,3), requires_grad = True)
w3 = torch.randn((3,3), requires_grad = True)
w4 = torch.randn((3,3), requires_grad = True)

b = w1*a
c = w2*a

d = w3*b + w4*c

# Replace L = (10 - d) by
L = (10 -d).sum()

L.backward()
```

Once that's done, you can access the gradients by calling the `grad` attribute of `Tensor`.

Second way is, for some reason have to absolutely call `backward` on a vector function, you can pass a `torch.ones` of size of shape of the tensor you are trying to call backward with.

```
# Replace L.backward() with
L.backward(torch.ones(L.shape))
```

Notice how `backward` used to take incoming gradients as it's input. Doing the above makes the `backward` think that incoming gradient are just Tensor of ones of same size as L, and it's able to backpropagate.

In this way, we can have gradients for every `Tensor` , and we can update them using Optimisation algorithm of our choice.

```
w1 = w1 - learning_rate * w1.grad
```

And so on.

How are PyTorch's graphs different from TensorFlow graphs

PyTorch creates something called a **Dynamic Computation Graph,** which means that the graph is generated on the fly.

Until the `forward` function of a Variable is called, there exists no node for the `Tensor` *(*it's `grad_fn` *)* in the graph.

```
a = torch.randn((3,3), requires_grad = True)   #No graph yet, as a is a leaf

w1 = torch.randn((3,3), requires_grad = True)  #Same logic as above

b = w1*a   #Graph with node `mulBackward` is created.
```

The graph is created as a result of `forward` function of many *Tensors* being invoked. Only then, the buffers for the non-leaf nodes allocated for the graph and intermediate values (used for computing gradients later. When you call `backward`, as the gradients are computed, these buffers (for non-leaf variables) are essentially freed, and the graph is *destroyed* (In a sense, you can't backpropagate through it since the buffers holding values to compute the gradients are gone).

Next time, you will call `forward` on the same set of tensors, **the leaf node buffers from the previous run will be shared, while the non-leaf nodes buffers will be created again.**

If you call `backward` more than once on a graph with non-leaf nodes, you'll be met with the following error.

```
RuntimeError: Trying to backward through the graph a second time, but the buff
```

This is because the non-leaf buffers gets destroyed the first time `backward()` is called and hence, there's no path to navigate to the leaves when `backward` is invoked the second time. You can undo this non-leaf buffer destroying behaviour by adding `retain_graph = True` argument to the `backward` function.

```
loss.backward(retain_graph = True)
```

If you do the above, you will be able to backpropagate again through the same graph and the gradients will be accumulated, i.e. the next you backpropagate, the gradients will be added to those already stored in the previous back pass.

This is in contrast to the **Static Computation Graphs**, used by TensorFlow where the graph is declared ***before*** running the program. Then the graph is "run" by feeding values to the predefined graph.

The dynamic graph paradigm allows you to make changes to your network architecture *during* runtime, as a graph is created only when a piece of code is run.

This means a graph may be redefined during the lifetime for a program since you don't have to define it beforehand.

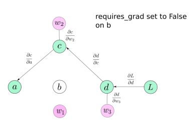
This, however, is not possible with static graphs where graphs are created before running the program, and merely executed later.

Dynamic graphs also make debugging way easier since it's easier to locate the source of your error.

Some Tricks of Trade

requires_grad

This is an attribute of the `Tensor` class. By default, it's False. It comes handy when you have to freeze some layers, and stop them from updating parameters while training. You can simply set the `requires_grad` to False, and these `Tensors` won't participate in the computation graph.



Thus, no gradient would be propagated to them, or to those layers which depend upon these layers for gradient flow `requires_grad`. When set to True, `requires_grad` is contagious meaning even if one operand of an operation has `requires_grad` set to True, so will the result.

torch.no_grad()

When we are computing gradients, we need to cache input values, and intermediate features as they maybe required to compute the gradient later.

The gradient of $b = w_1 * a$ w.r.t it's inputs w_1 and a is a and w_1 respectively. We need to store these values for gradient computation during the backward pass. This affects the memory footprint of the network.

While, we are performing inference, we don't compute gradients, and thus, don't need to store these values. Infact, no graph needs to be create during inference as it will lead to useless consumption of memory.

PyTorch offers a context manager, called `torch.no_grad` for this purpose.

```
with torch.no_grad:
        inference code goes here
```

No graph is defined for operations executed under this context manager.

Conclusion

Understanding how *Autograd* and computation graphs works can make life with PyTorch a whole lot easier. With our foundations rock solid, the next posts will detail how to create custom complex architectures, how to create custom data pipelines and more interesting stuff.

Further Reading

1. Chain Rule
2. Backpropagation

3 replies

Shubhankar Nov '19

Hey Ayoosh, it's a really succinct article. But I think there are a few typos in the equations where you describe the derivative using computational graph. dL/da is written as df/dw4 etc. Hope you will fix them.

brando90 8 Jan

Bumping up the typo already mentioned:

`dL/da is written as df/dw4 etc. Hope you will fix them.`

Abner 3 Feb

exellent blog, including essential concept for Computation Graphs.

Continue Discussion

MORE IN DEEP LEARNING

How AI And Deep Learning Are Shaping Education: An Interview With Terry Sejnowski
28 Feb 2020 – 7 min read

Working With The Lambda Layer in Keras
9 Feb 2020 – 11 min read

Understanding GauGAN Part 4: Debugging Training & Deciding If GauGAN Is Right For You
7 Feb 2020 – 13 min read

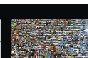
See all 58 posts →

PYTORCH

PyTorch 101, Part 2: Building Your First Neural Network

In this part, we will implement a neural network to classify CIFAR-10 images. We cover implementing the neural network, data loading pipeline and a decaying learning rate schedule.

AYOOSH KATHURIA
17 JUN 2019 · 13 MIN READ



TUTORIAL

Convert Full ImageNet Pre-trained Model from MXNet to PyTorch

Currently there are many available deep learning frameworks for researchers and engineers to implement their desired deep models. Each deep learning framework has its own advantages and disadvantages. For example,

AMIR HOSSEIN KARAMI
3 MAY 2019 · 8 MIN READ