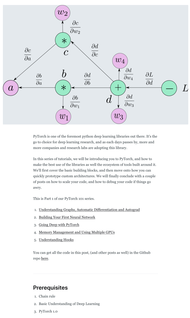
DEEP LEARNING

PyTorch 101, Part 1: Understanding Graphs, Automatic Differentiation and Autograd

In this article, we dive into how PyTorch's Autograd engine performs automatic differentiation.

AYOOSH KATHUROSIAN
Jan 31, 2019 · 10 min read



PyTorch is one of the foremost python deep learning libraries out there. It's the go to choice for deep learning research, and as such does pass by, more and more companies and research labs are adopting this library.

In this series of tutorials, we will be introducing you to PyTorch, and how to make the best use of the libraries as well the ecosystem of tools built around it. We'll first cover the basic building blocks, and then move onto how you can quickly prototype custom architectures. We will finally conclude with a couple of posts on how to scale your code, and how to delay your code if things go awry.

This is Part 1 of our PyTorch 101 series.

1. Understanding Graphs, Automatic Differentiation and Autograd
2. Building Your First Neural Network
3. Going Deep with PyTorch
4. Memory Management and Using Multiple GPUs
5. Understanding Hooks

You can get all the code in this post, (and other posts as well) in the Github repo here.

Prerequisites

1. Chain rule
2. Basic Understanding of Deep Learning
3. PyTorch 1.0

You can get all the code in this post, (and other posts as well) in the Github repo here.

Automatic Differentiation

A lot of tutorial series on PyTorch could start begin with a rudimentary discussion of what the basic structures are. However, I'd like to instead start by discussing automatic differentiation first.

Automatic Differentiation is a building block of not only PyTorch, but every DL library out there. In my opinion, PyTorch's automatic differentiation engine, called Autograd is a brilliant tool to understand how automatic differentiation works. This will not only help you understand PyTorch better, but also other DL libraries.

Modern neural network architectures can have millions of learnable parameters. From a computational point of view, training a neural network consists of two phases:

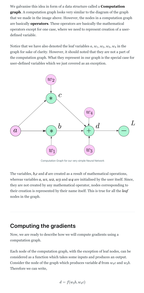
1. A forward pass to compute the value of the loss function.
2. A backward pass to compute the gradients of the learnable parameters.

The forward pass is pretty straight forward. The output of one layer is the input to the next and so forth.

Backward pass is a bit more complicated since it requires us to use the chain rule to compute the gradients of weights w.r.t to the loss function.

A toy example

Let us take an very simple neural network consisting of just 5 neurons. Our neural network looks like the following.



A Very Simple Neural Network

The following equations describe our neural network.

$$
\begin{aligned}
b &= w_1 * a \\
c &= w_2 * a \\
d &= w_3 * b + w_4 * c \\
L &= 10 - d
\end{aligned}
$$

Let us compute the gradients for each of the learnable parameters w.

$$
\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial b} * \frac{\partial b}{\partial w_1}
$$

$$
\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial c} * \frac{\partial c}{\partial w_2}
$$

$$
\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial w_3}
$$

$$
\frac{\partial L}{\partial w_4} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial w_4}
$$

All these gradients have been computed by applying the chain rule. Note that all the individual gradients on the right hand side of the equations mentioned above can be computed directly since the numerators of the gradients are explicit functions of the denominators.

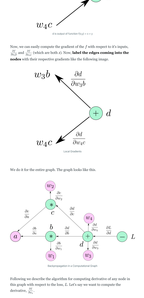
Computation Graphs

We could manually compute the gradients of our network as it was very simple. Imagine, what if you had a network with 150 layers. Or, if the network had multiple branches.

When we design software to implement neural networks, we want to come up with a way that can allow us to seamlessly compute the gradients, regardless of the architecture type so that the programmer doesn't have to manually compute gradients when changes are made to the network.

We galvanize this idea in form of a data structure called a **Computation graph**. A computation graph looks very similar to the diagram of the graph that we made in the image above. However, the nodes in a computation graph are very fundamentally primitive. These operations can be fundamentally as operations comparison ops, where we need to represent creation of a user defined variable.

Notice that we have also denoted the leaf variables, a, w_1, w_2, w_3, w_4 in the graph. We need to keep track of where these variables are, as they are a part of the computation graph. What they represent in our graph is the special case of user-defined variables which are just covered as an exception.



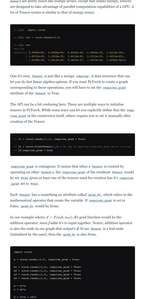
Computation Graph for our very simple Neural Network

Computing the gradients

Now, we are ready to describe how we will compute gradients using a computation graph.

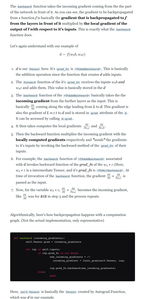
Each node of the computation graph, with the exception of leaf nodes, can be considered as a function which takes some inputs and produces an output. Consider the node of the graph which produces variable d from w_3 and b. Therefore we can write,

$$
d = f(w_3, b, w_4)
$$



Computation of Variable d in Graph

Now, we can easily compute the gradient of the f with respect to it's inputs, $\frac{\partial f}{\partial w_3}$ and $\frac{\partial f}{\partial b}$ (which are both 1). Now, **label the edges coming into the nodes** with their respective gradients like the following image.



Local Gradients

We do it for the entire graph. The graph looks like this.

Backpropagation in a Computational Graph

Following we describe the algorithm for computing derivatives of any node in this graph with respect to the loss, L. Let's say we want to compute the derivatives, $\frac{\partial L}{\partial n}$.

1. We first traverse all possible paths from d to w_1
2. There is only one such path.
3. We multiply all the edges along this path.

If you are, the product is precisely the same expression we derived using chain rule. If there is more than one path to a variable from L then, we multiply the edges along each path and then add them together. For example, $\frac{\partial L}{\partial c}$ is computed as

$$
\frac{\partial f}{\partial c} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial w_4 c} * \frac{\partial w_4 c}{\partial c}
$$

PyTorch Autograd

Now we get what a computational graph is, let's get back to PyTorch and understand how the above is implemented in PyTorch.

Tensor

`Tensor` is a data structure which is a fundamental building block of PyTorch. `Tensor`s are pretty much like numpy arrays, except that unlike numpy, tensors are designed to take advantage of parallel computation capabilities of a GPU. A lot of Tensor ops is similar to that of numpy.

```
>>> import torch
>>> this tensor = torch.Tensor(3,5)
>>
>>> this_tensor
tensor([[ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
        [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00],
        [ 0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00,  0.0000e+00]])
```

One it's own, `Tensor` is just like a numpy `ndarray`. A data structure that can let you do fast linear algebra options. If you want PyTorch to create a graph corresponding to these operations, you will have to set the `requires_grad` attribute of the `Tensor` to True.

The API can be a bit confusing here. There are multiple ways to initialize tensors in PyTorch. While some ways are for you explicitly define that the `requires_grad` in the constructor itself, others require you to set it manually after creation of the Tensor.

```
>>> x = torch.randn(5,5, requires_grad = True)
>>> y = torch.randn(5,5, requires_grad = True)
>>> z = torch.tensor([1.0,2.0,3.0], requires_grad=True)
>>> print(x,y,z)
```

`requires_grad` is contagious. It means that where a `Tensor` is created by operating on other `Tensor`s, the `requires_grad` of the resultant `Tensor` would also be set to True. Because one of the tensors used for creation has it's `requires_grad` set to True.

Each `Tensor` has a something an attribute called `grad_fn`, which refers to the mathematical operation that create the variable. If `requires_grad` is set to False, `grad_fn` would be None.

In our example where, $d = f(w_3 b, w_4 c)$, f's grad function would be the addition operator, since f adds it's to input together. Notice, addition operator is also the node in our graph that output of d from `Tensor`s is a leaf node (initialized by the user), then the `grad_fn` is also None.

```
>>> tensor.grad_fn
>>
>>> x.torch.cos(x.T), requires_grad = True)
>>> y.torch.sin(x.T), requires_grad = True)
>>> z.torch.tan(x.T), requires_grad = True)
>>> print(x.grad_fn, y.grad_fn, z.grad_fn)
>>
None None None
>>
>>> x.grad_fn
>>
>>> x.torch.cos(x.T), requires_grad = True)
>>> y.torch.sin(x.T), requires_grad = True)
>>> z.torch.tan(x.T), requires_grad = True)
>>> print(x.grad_fn, y.grad_fn, z.grad_fn)
>>
None None None
>>
>>> w = x*y
>>> print(w.grad_fn)
>>
>>> w.grad_fn
>>
>>> a = w + z
>>> print(a.grad_fn)
>>
<AddBackward0 object at 0x7f8d8c0ff7b0>
>>> print(a.grad_fn.next_functions)
>>
((<MulBackward0 object at 0x7f8d8c0ff7b0>, 0), (None, 0))
```

If you run the code above, you get the following output.

```
>>> grad_fn = a.grad_fn
>>> grad_fn.next_functions[0][0].next_functions
```

One can use the member function `.next_functions` to determine whether a variable is a leaf `Tensor` or not.

Function

All mathematical operations in PyTorch are implemented by the `torch.autograd.Function` class. This class has two important member functions we need to look at.

The first is the `forward` function, which simply computes the output using it's inputs.

The `backward` function takes the incoming gradient coming from the the part of the network in front of it. As you can see, the gradient to be backpropagated from this function is the **product of the local gradients of the function with the gradients from the layers in front of it** multiplied by **the local gradient of the output of Tensor with respect to it's inputs**. This is exactly what the `backward` function does.

Let's again understand with our example of

$$
d = f(w_3 b, w_4 c)
$$

1. d is our `Tensor` from 10's `grad_fn` is `<AddBackward0>`. This is basically the addition operation since the function that creates d adds inputs.
2. The `forward` function of the 10's `grad_fn` receives the inputs $w_3 b$ and $w_4 c$ with dims Then. This value is basically stored in the d.
3. The `backward` function of the `<AddBackward0>` basically takes the the incoming gradient from the layers in front of it, which is $\frac{\partial L}{\partial d}$ basically $\frac{\partial L}{\partial d}$ coming along the edge leading from L to d. This gradient is also the gradient of L w.r.t d and is stored in `grad` attribute of the d. It can be accessed by calling `d.grad`.
4. It then takes computes the local gradients $\frac{\partial d}{\partial w_3 b}$ and $\frac{\partial d}{\partial w_4 c}$.
5. Then the backward function multiplies the incoming gradient with the **locally computed gradients** respectively and "sends"the gradients to it's inputs by invoking the backward method of the `grad_fn` of its inputs.
6. For example, the `backward` function of `<MulBackward0>` associated with d sends backward function of the `grad_fn` of the $w_3 * b$ tensor. The incoming gradient value that the `<MulBackward0>` receives is the product of the local derivative of the `backward` function, the gradient $\frac{\partial L}{\partial d}$ it's inputs.
7. Now, for the variable w_3, $\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial d} * \frac{\partial d}{\partial w_3}$ becomes the incoming gradient. Like $\frac{\partial L}{\partial d}$ was for ∂d in step 5 and the process repeats.

Algorithmically, here's how backpropagation happens with a computation graph. Over the actual implementation, only representation)

```
def backward (incoming_gradients):
    self.tensor.grad = incoming_gradients

    for op in self.inputs:
        if op.grad_fn is not None:
            local_grads = self.grad_fn(*self.inputs)
            for (input, local_grad) in zip(self.inputs, local_grads):
                incoming_grad = incoming_gradients * local_grad
                input.grad += incoming_grad
                input.grad_fn.backward(input, incoming_gradients)
```

Here, `self` refers to `Tensor` created by Autograd Function, which was of an output.

Incoming gradients and local gradients have been described above.

In order to compute derivatives in our neural network, we generally call `.backw` on the `Tensor` representing our loss. Then, we backtrack through the graph starting from node representing the `grad_fn` of our loss.

As described above, the `backward` function is recursively called through out the graph. Each node calculates the local grads and feeds onto the `grad_fn` in it's inputs backwards through that path.

One thing to note here is that PyTorch gives an error if you call `backward()` on vector-valued Tensor. This means you can only call `backward` on a scalar valued Tensor. In our example, if we assume a to be a vector valued Tensor, we would need to pass in a gradient argument to `backward()`.

In order to compute derivatives in our neural network, we generally call `.backw` on the `Tensor` representing our loss. Then, we backtrack through the graph starting from node representing the `grad_fn` of our loss.

As described above, the `backward` function is recursively called through out the graph. Each node calculates the local grads and feeds onto the `grad_fn` in it's inputs backwards through that path.

One thing to note here is that PyTorch gives an error if you call `backward()` on vector-valued Tensor. This means you can only call `backward` on a scalar valued Tensor. In our example, if we assume a to be a vector valued Tensor, we would need to pass in a gradient argument to `backward()`.

```
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 `ba ckward` , 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 `Tenso rs` 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

Add speed and simplicity to your Machine Learning workflow today

GET STARTED CONTACT SALES

Weights & Biases for PyTorch

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 dl/da etc. Hope you will fix them.

brandonpo 8 Jan

Bumping up the typo already mentioned:

`dL/da` is written as `dl/dwd` etc. Hope you will fix them.

Abner 3 Feb

enrllent 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
22 Nov 2020 · 7 min read

Working With The Lambda Layer in Keras
9 Dec 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 18 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 · 27 JUN 2018 · 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,
JAMIE HOSEIN KARAMI · 2 MAY 2019 · 3 MIN READ

Paperspace Blog © 2020
Latest Posts · Facebook · Twitter · Ghost