

Get started

Open in app

**towards**
data science

Follow

617K Followers



Understanding PyTorch with an example: a step-by-step tutorial



Daniel Godoy May 7, 2019 · 21 min read

Photo by [Allen Cai](#) on [Unsplash](#)

Update (May 18th, 2021): Today I've finished my book: [Deep Learning with PyTorch Step-by-Step: A Beginner's Guide](#).

Introduction

PyTorch is the **fastest growing** Deep Learning framework and it is also used by **Fast.ai** in its MOOC, [Deep Learning for Coders](#) and its [library](#).

PyTorch is also very *pythonic*, meaning, it feels more natural to use it if you already are a Python developer.

Besides, using PyTorch may even *improve your health*, according to [Andrej Karpathy](#). :-)

Motivation

There are *many many* PyTorch tutorials around and its documentation is quite complete and extensive. So, **why** should you keep reading this step-by-step tutorial?

[Get started](#)[Open in app](#)

In this post, I will guide you through the *main reasons* why PyTorch makes it much **easier** and more **intuitive** to build a Deep Learning model in Python — **autograd**, **dynamic computation graph**, **model classes** and more — and I will also show you **how to avoid** some **common pitfalls** and **errors** along the way.

Moreover, since this is quite a **long** post, I built a *Table of Contents* to make navigation easier, should you use it as a **mini-course** and work your way through the content one topic at a time.

Table of Contents

- [A Simple Regression Problem](#)
- [Gradient Descent](#)
- [Linear Regression in Numpy](#)
- [PyTorch](#)
- [Autograd](#)
- [Dynamic Computation Graph](#)
- [Optimizer](#)
- [Loss](#)
- [Model](#)
- [Dataset](#)
- [DataLoader](#)
- [Evaluation](#)

A Simple Regression Problem

Most tutorials start with some nice and pretty *image classification problem* to illustrate how to use PyTorch. It may seem cool, but I believe it **distracts** you from the **main goal**: **how PyTorch works?**

For this reason, in this tutorial, I will stick with a **simple** and **familiar** problem: a **linear regression with a single feature x** ! It doesn't get much simpler than that...

$$y = a + bx + \epsilon$$

Simple Linear Regression model

Data Generation

Let's start **generating** some synthetic data: we start with a vector of 100 points for our **feature x** and create our **labels** using **$a = 1$** , **$b = 2$** and some Gaussian noise.

Next, let's **split** our synthetic data into **train** and **validation** sets, shuffling the array of indices and using the first 80 shuffled points for training.

Get started

Open in app



```

4  y = 1 + 2 * x + .1 * np.random.randn(100, 1)
5
6  # Shuffles the indices
7  idx = np.arange(100)
8  np.random.shuffle(idx)
9
10 # Uses first 80 random indices for train
11 train_idx = idx[:80]
12 # Uses the remaining indices for validation
13 val_idx = idx[80:]
14
15 # Generates train and validation sets
16 x_train, y_train = x[train_idx], y[train_idx]
17 x_val, y_val = x[val_idx], y[val_idx]

```

torch101_data_gen.py hosted with ❤ by GitHub

view raw

Generating synthetic train and validation sets for a linear regression

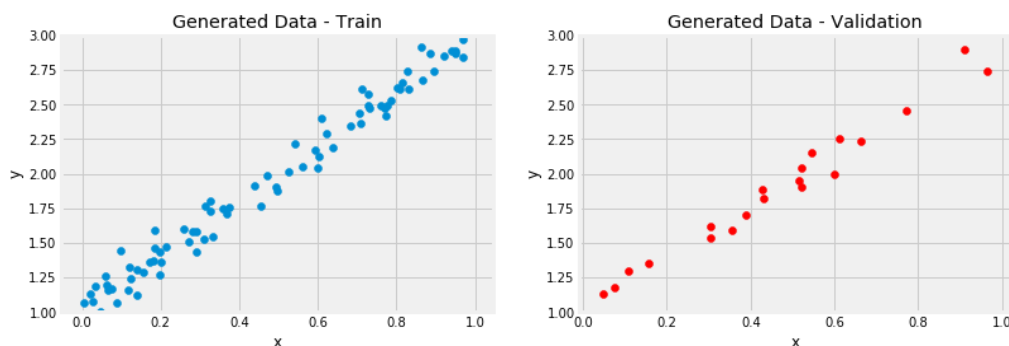


Figure 1: Synthetic data — Train and Validation sets

We **know** that $a = 1$ and $b = 2$, but now let's see how close we can get to the true values by using **gradient descent** and the 80 points in the **training set**...

Gradient Descent

If you are comfortable with the inner workings of gradient descent, *feel free to skip* this section. It goes beyond the scope of this post to fully explain how gradient descent works, but I'll cover the **four basic steps** you'd need to go through to compute it.

Step 1: Compute the Loss

For a regression problem, the **loss** is given by the **Mean Square Error (MSE)**, that is, the average of all squared differences between **labels** (y) and **predictions** ($a + bx$).

*It is worth mentioning that, if we use **all points** in the training set (N) to compute the loss, we are performing a **batch** gradient descent. If we were to use a **single point** at each time, it would be a **stochastic** gradient descent. Anything else (n) **in-between 1 and N** characterizes a **mini-batch** gradient descent.*

$$MSE = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

$$MSE = \frac{1}{N} \sum_{i=1}^N (y_i - a - bx_i)^2$$

Get started

Open in app



Step 2: Compute the Gradients

A **gradient** is a **partial derivative** — *why partial?* Because one computes it with respect to (w.r.t.) a **single parameter**. We have two parameters, **a** and **b**, so we must compute two partial derivatives.

A **derivative** tells you *how much a given quantity changes* when you *slightly vary* some **other quantity**. In our case, how much does our **MSE loss** change when we vary **each one of our two parameters**?

The *right-most* part of the equations below is what you usually see in implementations of gradient descent for a simple linear regression. In the **intermediate step**, I show you **all elements** that pop-up from the application of the chain rule, so you know how the final expression came to be.

$$\frac{\partial MSE}{\partial a} = \frac{\partial MSE}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial a} = \frac{1}{N} \sum_{i=1}^N 2(y_i - a - bx_i) \cdot (-1) = -2 \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)$$

$$\frac{\partial MSE}{\partial b} = \frac{\partial MSE}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial b} = \frac{1}{N} \sum_{i=1}^N 2(y_i - a - bx_i) \cdot (-x_i) = -2 \frac{1}{N} \sum_{i=1}^N x_i (y_i - \hat{y}_i)$$

Computing gradients w.r.t coefficients a and b

Step 3: Update the Parameters

In the final step, we **use the gradients to update** the parameters. Since we are trying to **minimize** our **losses**, we **reverse the sign** of the gradient for the update.

There is still another parameter to consider: the **learning rate**, denoted by the *Greek letter eta* (that looks like the letter *n*), which is the **multiplicative factor** that we need to apply to the gradient for the parameter update.

$$a = a - \eta \frac{\partial MSE}{\partial a}$$

$$b = b - \eta \frac{\partial MSE}{\partial b}$$

Updating coefficients a and b using computed gradients and a learning rate

How to **choose** a learning rate? That is a topic on its own and beyond the scope of this post as well.

Step 4: Rinse and Repeat!

Now we use the **updated parameters** to go back to **Step 1** and restart the process.

*An epoch is complete whenever every point has been already used for computing the loss. For **batch** gradient descent, this is trivial, as it uses all points for computing the loss — one epoch is the same as one update. For **stochastic** gradient descent, one epoch means **N updates**, while for **mini-batch** (of size *n*), one epoch has **N/n updates**.*

[Get started](#)[Open in app](#)

Linear Regression in Numpy

It's time to implement our linear regression model using gradient descent using **Numpy only**.

Wait a minute... I thought this tutorial was about PyTorch!

Yes, it is, but this serves **two purposes**: *first*, to introduce the **structure** of our task, which will remain largely the same and, *second*, to show you the main **pain points** so you can fully appreciate how much PyTorch makes your life easier :-)

For training a model, there are **two initialization steps**:

- Random initialization of parameters/weights (we have only two, *a* and *b*) — lines 3 and 4;
- Initialization of hyper-parameters (in our case, only *learning rate* and *number of epochs*) — lines 9 and 11;

Make sure to *always initialize your random seed* to ensure **reproducibility** of your results. As usual, the random seed is 42, the *least random* of all random seeds one could possibly choose :-)

For each epoch, there are **four training steps**:

- Compute model's predictions — this is the **forward pass** — line 15;
- Compute the loss, using *predictions* and *labels* and the appropriate **loss function** for the task at hand — lines 18 and 20;
- Compute the **gradients** for every parameter — lines 23 and 24;
- **Update** the parameters — lines 27 and 28;

Just keep in mind that, if you *don't* use batch gradient descent (our example does), you'll have to write an **inner loop** to perform the **four training steps** for either each **individual point (stochastic)** or ***n* points (mini-batch)**. We'll see a mini-batch example later down the line.

```
1 # Initializes parameters "a" and "b" randomly
2 np.random.seed(42)
3 a = np.random.randn(1)
4 b = np.random.randn(1)
5
6 print(a, b)
7
8 # Sets learning rate
9 lr = 1e-1
10 # Defines number of epochs
11 n_epochs = 1000
12
13 for epoch in range(n_epochs):
14     # Computes our model's predicted output
15     yhat = a + b * x_train
16
17     # How wrong is our model? That's the cost
```

Get started

Open in app



```

20     loss = (error ** 2).mean()
21
22     # Computes gradients for both "a" and "b" parameters
23     a_grad = -2 * error.mean()
24     b_grad = -2 * (x_train * error).mean()
25
26     # Updates parameters using gradients and the learning rate
27     a = a - lr * a_grad
28     b = b - lr * b_grad
29
30     print(a, b)
31
32     # Sanity Check: do we get the same results as our gradient descent?
33     from sklearn.linear_model import LinearRegression
34     linr = LinearRegression()
35     linr.fit(x_train, y_train)
36     print(linr.intercept_, linr.coef_[0])

```

torch101_numpy.py hosted with ❤ by GitHub

[view raw](#)

Implementing gradient descent for linear regression using Numpy

Just to make sure we haven't done any mistakes in our code, we can use *Scikit-Learn's Linear Regression* to fit the model and compare the coefficients.

```

# a and b after initialization
[0.49671415] [-0.1382643]
# a and b after our gradient descent
[1.02354094] [1.96896411]
# intercept and coef from Scikit-Learn
[1.02354075] [1.96896447]

```

They **match** up to 6 decimal places — we have a *fully working implementation of linear regression* using Numpy.

Time to **TORCH** it :-)

PyTorch

First, we need to cover a **few basic concepts** that may throw you off-balance if you don't grasp them well enough before going full-force on modeling.

In Deep Learning, we see **tensors** everywhere. Well, Google's framework is called *TensorFlow* for a reason! *What is a tensor, anyway?*

Tensor

In *Numpy*, you may have an **array** that has **three dimensions**, right? That is, technically speaking, a **tensor**.

A **scalar** (a single number) has **zero** dimensions, a **vector** has **one** dimension, a **matrix** has **two** dimensions and a **tensor** has **three or more** dimensions. That's it!

But, to keep things simple, it is commonplace to call vectors and matrices tensors as well — so, from now on, **everything is either a scalar or a tensor**.

Scalar

Vector

Matrix

Tensor

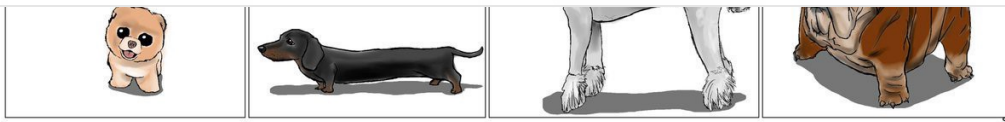
[Get started](#)[Open in app](#)

Figure 2: Tensors are just higher-dimensional matrices :-)

[Source](#)

Loading Data, Devices and CUDA

“How do we go from Numpy’s arrays to PyTorch’s tensors”, you ask? That’s what `from numpy` is good for. It returns a **CPU tensor**, though.

“But I want to use my fancy GPU...”, you say. No worries, that’s what `to()` is good for. It sends your tensor to whatever **device** you specify, including your **GPU** (referred to as `cuda` or `cuda:0`).

“What if I want my code to fallback to CPU if no GPU is available?”, you may be wondering... PyTorch got your back once more — you can use `cuda.is_available()` to find out if you have a GPU at your disposal and set your device accordingly.

You can also easily **cast** it to a lower precision (32-bit float) using `float()`.

```
1 import torch
2 import torch.optim as optim
3 import torch.nn as nn
4 from torchviz import make_dot
5
6 device = 'cuda' if torch.cuda.is_available() else 'cpu'
7
8 # Our data was in Numpy arrays, but we need to transform them into PyTorch's Tensors
9 # and then we send them to the chosen device
10 x_train_tensor = torch.from_numpy(x_train).float().to(device)
11 y_train_tensor = torch.from_numpy(y_train).float().to(device)
12
13 # Here we can see the difference - notice that .type() is more useful
14 # since it also tells us WHERE the tensor is (device)
15 print(type(x_train), type(x_train_tensor), x_train_tensor.type())
```

torch101_imports.py hosted with ❤ by GitHub

[view raw](#)

Loading data: turning Numpy arrays into PyTorch tensors

If you compare the **types** of both variables, you’ll get what you’d expect: `numpy.ndarray` for the first one and `torch.Tensor` for the second one.

But where does your nice tensor “live”? In your CPU or your GPU? You can’t say... but if you use PyTorch’s `type()`, it will reveal its **location** — `torch.cuda.FloatTensor` — a GPU tensor in this case.

We can also go the other way around, turning tensors back into Numpy arrays, using `numpy()`. It should be easy as `x_train_tensor.numpy()` **but...**

```
TypeError: can't convert CUDA tensor to numpy. Use Tensor.cpu() to copy the tensor to host memory first.
```

[Get started](#)[Open in app](#)

Creating Parameters

What distinguishes a *tensor* used for *data* — like the ones we've just created — from a **tensor** used as a (*trainable*) **parameter/weight**?

The latter tensors require the **computation of its gradients**, so we can **update** their values (the parameters' values, that is). That's what the `requires_grad=True` argument is good for. It tells PyTorch we want it to compute gradients for us.

You may be tempted to create a simple tensor for a parameter and, later on, send it to your chosen device, as we did with our data, right? Not so fast...

```

1  # FIRST
2  # Initializes parameters "a" and "b" randomly, ALMOST as we did in Numpy
3  # since we want to apply gradient descent on these parameters, we need
4  # to set REQUIRES_GRAD = TRUE
5  a = torch.randn(1, requires_grad=True, dtype=torch.float)
6  b = torch.randn(1, requires_grad=True, dtype=torch.float)
7  print(a, b)
8
9  # SECOND
10 # But what if we want to run it on a GPU? We could just send them to device, right?
11 a = torch.randn(1, requires_grad=True, dtype=torch.float).to(device)
12 b = torch.randn(1, requires_grad=True, dtype=torch.float).to(device)
13 print(a, b)
14 # Sorry, but NO! The to(device) "shadows" the gradient...
15
16 # THIRD
17 # We can either create regular tensors and send them to the device (as we did with our c
18 a = torch.randn(1, dtype=torch.float).to(device)
19 b = torch.randn(1, dtype=torch.float).to(device)
20 # and THEN set them as requiring gradients...
21 a.requires_grad_()
22 b.requires_grad_()
23 print(a, b)

```

torch101_bad_inits.py hosted with ❤ by GitHub

[view raw](#)

Trying to create variables for the coefficients...

The first chunk of code creates two nice tensors for our parameters, gradients and all. But they are **CPU** tensors.

```

# FIRST
tensor([-0.5531], requires_grad=True)
tensor([-0.7314], requires_grad=True)

```

In the second chunk of code, we tried the **naïve** approach of sending them to our GPU. We succeeded in sending them to another device, but we **"lost"** the **gradients** somehow...

```

# SECOND
tensor([0.5158], device='cuda:0', grad_fn=<CopyBackwards>)
tensor([0.0246], device='cuda:0', grad_fn=<CopyBackwards>)

```


[Get started](#)[Open in app](#)

method to set its `requires_grad` to `True` in place.

```
# THIRD
tensor([-0.8915], device='cuda:0', requires_grad=True)
tensor([0.3616], device='cuda:0', requires_grad=True)
```

*In PyTorch, every method that **ends** with an **underscore** (`_`) makes changes **in-place**, meaning, they will **modify** the underlying variable.*

Although the last approach worked fine, it is much better to **assign** tensors to a **device** at the moment of their **creation**.

```
1 # We can specify the device at the moment of creation - RECOMMENDED!
2 torch.manual_seed(42)
3 a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
4 b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
5 print(a, b)
```

torch101_good_init.py hosted with ❤ by GitHub

[view raw](#)

Actually creating variables for the coefficients :-)

```
tensor([0.6226], device='cuda:0', requires_grad=True)
tensor([1.4505], device='cuda:0', requires_grad=True)
```

Much easier, right?

Now that we know how to create tensors that require gradients, let's see how PyTorch handles them — that's the role of the...

Autograd

Autograd is PyTorch's *automatic differentiation package*. Thanks to it, we **don't need to worry** about *partial derivatives*, *chain rule* or anything like it.

So, how do we tell PyTorch to do its thing and **compute all gradients**? That's what `backward()` is good for.

Do you remember the **starting point** for **computing the gradients**? It was the **loss**, as we computed its partial derivatives w.r.t. our parameters. Hence, we need to invoke the `backward()` method from the corresponding Python variable, like, `loss.backward()`.

What about the **actual values** of the **gradients**? We can inspect them by looking at the `grad` attribute of a tensor.

If you check the method's documentation, it clearly states that **gradients are accumulated**. So, every time we use the **gradients** to **update** the parameters, we need to **zero the gradients afterwards**. And that's what `zero_()` is good for.

What does the **underscore** (`_`) at the **end of the method name** mean? Do you remember? If not, scroll back to the previous section and find out.

[Get started](#)[Open in app](#)

That's it? Well, pretty much... but, there is always a **catch**, and this time it has to do with the **update** of the **parameters**...

```

1  lr = 1e-1
2  n_epochs = 1000
3
4  torch.manual_seed(42)
5  a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
6  b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
7
8  for epoch in range(n_epochs):
9      yhat = a + b * x_train_tensor
10     error = y_train_tensor - yhat
11     loss = (error ** 2).mean()
12
13     # No more manual computation of gradients!
14     # a_grad = -2 * error.mean()
15     # b_grad = -2 * (x_tensor * error).mean()
16
17     # We just tell PyTorch to work its way BACKWARDS from the specified loss!
18     loss.backward()
19     # Let's check the computed gradients...
20     print(a.grad)
21     print(b.grad)
22
23     # What about UPDATING the parameters? Not so fast...
24
25     # FIRST ATTEMPT
26     # AttributeError: 'NoneType' object has no attribute 'zero_'
27     # a = a - lr * a.grad
28     # b = b - lr * b.grad
29     # print(a)
30
31     # SECOND ATTEMPT
32     # RuntimeError: a leaf Variable that requires grad has been used in an in-place operation
33     # a -= lr * a.grad
34     # b -= lr * b.grad
35
36     # THIRD ATTEMPT
37     # We need to use NO_GRAD to keep the update out of the gradient computation
38     # Why is that? It boils down to the DYNAMIC GRAPH that PyTorch uses...
39     with torch.no_grad():
40         a -= lr * a.grad
41         b -= lr * b.grad
42
43     # PyTorch is "clingy" to its computed gradients, we need to tell it to let it go...
44     a.grad.zero_()
45     b.grad.zero_()
46
47     print(a, b)

```

torch101_updating_parms.py hosted with ❤ by GitHub

[view raw](#)

In the first attempt, if we use the same update structure as in our *Numpy* code, we'll get the weird **error** below... but we can get a *hint* of what's going on by looking at the tensor itself — once again we “lost” the **gradient** while reassigning the update results to our parameters. Thus, the **grad** attribute turns out to be **None** and it raises the error...

[Get started](#)[Open in app](#)

```
AttributeError: 'NoneType' object has no attribute 'zero_'
```

We then change it slightly, using a familiar **in-place Python assignment** in our second attempt. And, once again, PyTorch complains about it and raises an **error**.

```
# SECOND ATTEMPT
RuntimeError: a leaf Variable that requires grad has been used in an
in-place operation.
```

*Why?! It turns out to be a case of “**too much of a good thing**”. The culprit is PyTorch’s ability to build a **dynamic computation graph** from every **Python operation** that involves any **gradient-computing tensor** or its **dependencies**.*

We’ll go deeper into the inner workings of the dynamic computation graph in the next section.

So, how do we tell PyTorch to “**back off**” and let us **update our parameters** without messing up with its *fancy dynamic computation graph*? That’s what `torch.no_grad()` is good for. It allows us to **perform regular Python operations on tensors, independent of PyTorch’s computation graph**.

Finally, we managed to successfully run our model and get the **resulting parameters**. Surely enough, they **match** the ones we got in our *Numpy*-only implementation.

```
# THIRD ATTEMPT
tensor([1.0235], device='cuda:0', requires_grad=True)
tensor([1.9690], device='cuda:0', requires_grad=True)
```

Dynamic Computation Graph

“Unfortunately, no one can be told what the dynamic computation graph is. You have to see it for yourself.”
Morpheus

How great was “*The Matrix*”? Right, right? But, jokes aside, I want **you to see the graph for yourself** too!

The `PyTorchViz` package and its `make_dot(variable)` method allows us to easily visualize a graph associated with a given Python variable.

So, let’s stick with the **bare minimum**: two (*gradient computing*) **tensors** for our parameters, predictions, errors and loss.

```
1 torch.manual_seed(42)
2 a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
3 b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
4
5 yhat = a + b * x_train_tensor
```

Get started

Open in app



torch101_dyn_graph.py hosted with ❤ by GitHub

view raw

Computing MSE in three steps

If we call `make_dot(yhat)` we'll get the **left-most graph** on Figure 3 below:

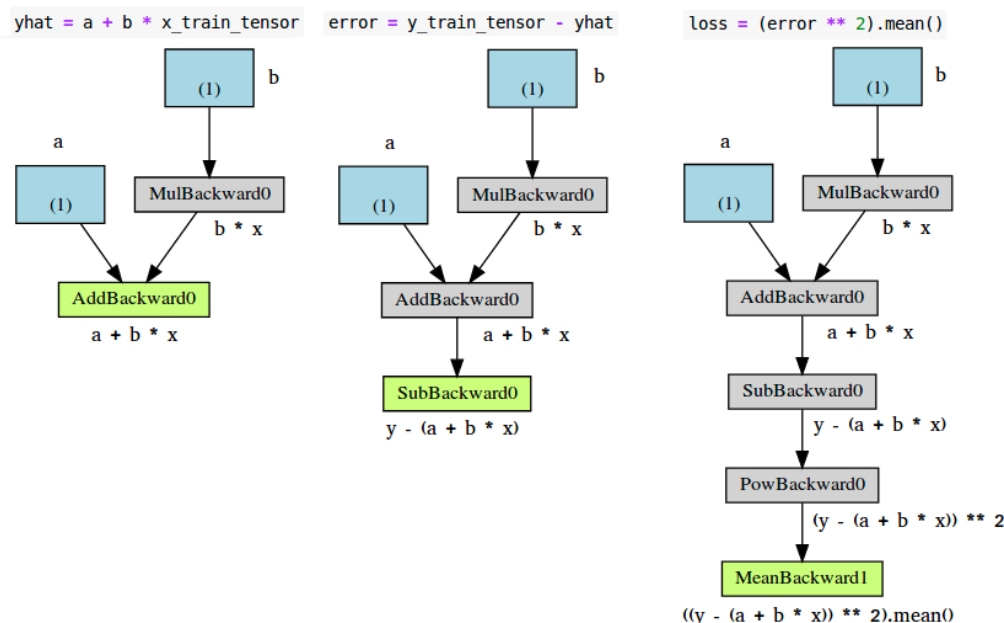


Figure 3: Computation graph for every step in computing MSE

Let's take a closer look at its components:

- **blue boxes:** these correspond to the **tensors** we use as **parameters**, the ones we're asking PyTorch to **compute gradients** for;
- **gray box:** a **Python operation** that involves a **gradient-computing tensor** or its **dependencies**;
- **green box:** the same as the gray box, except it is the **starting point for the computation** of gradients (assuming the `backward()` method is called from the **variable used to visualize** the graph)— they are computed from the **bottom-up** in a graph.

If we plot graphs for the `error` (center) and `loss` (right) **variables**, the **only difference** between them and the first one is the number of **intermediate steps** (gray boxes).

Now, take a closer look at the **green box** of the **left-most graph**: there are **two arrows** pointing to it, since it is **adding up two variables**, `a` and `b*x`. Seems obvious, right?

Then, look at the **gray box** of the same graph: it is performing a **multiplication**, namely, `b*x`. But there is only one arrow pointing to it! The arrow comes from the **blue box** that corresponds to our **parameter b**.

Why don't we have a box for our data x? The answer is: we **do not compute gradients** for it! So, even though there are *more* tensors involved in the operations performed by the computation graph, it **only shows gradient-computing tensors and its dependencies**.



```
a_nograd = torch.randn(1, requires_grad=False, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

yhat = a_nograd + b * x_train_tensor
```

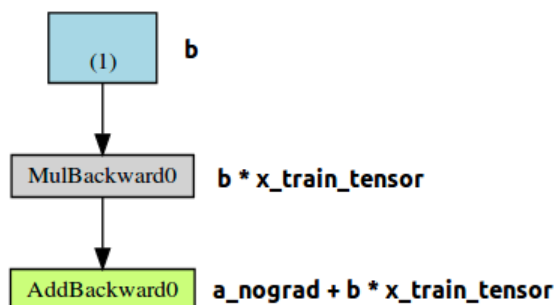


Figure 4: now variable *a* does NOT have its gradient computed anymore. But it is STILL used in computation

Unsurprisingly, the **blue box** corresponding to the **parameter *a*** is no more! Simple enough: **no gradients, no graph**.

The **best** thing about the *dynamic computing graph* is the fact that you can make it **as complex as you want** it. You can even use *control flow statements* (e.g., *if* statements) to **control the flow of the gradients** (obviously!) :-)

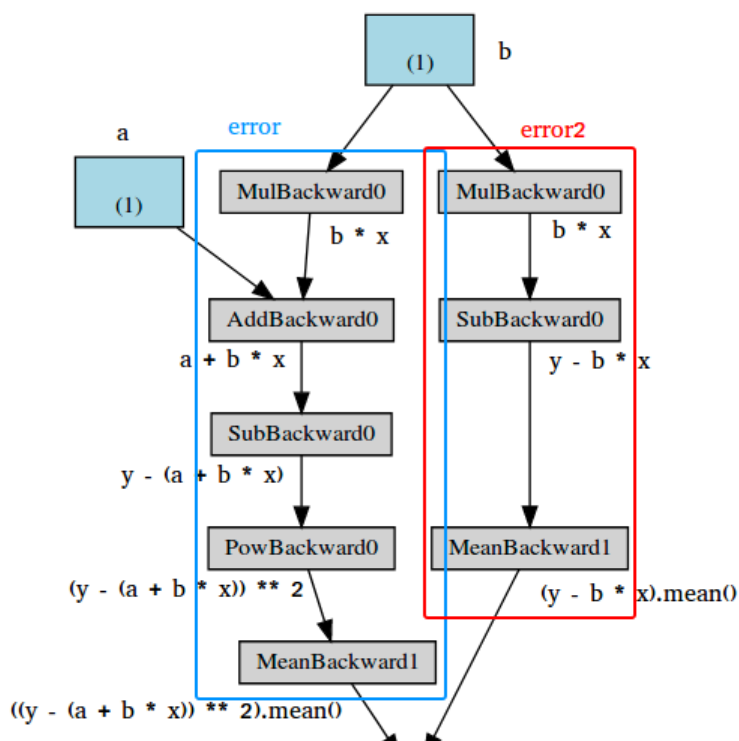
Figure 5 below shows an example of this. And yes, I do know that the computation itself is *completely nonsense*...

```
yhat = a + b * x_train_tensor
error = y_train_tensor - yhat

loss = (error ** 2).mean()

if loss > 0:
    yhat2 = b * x_train_tensor
    error2 = y_train_tensor - yhat2

    loss += error2.mean()
```



[Get started](#)[Open in app](#)

Figure 5: Complex computation graph just to make a point :-)

Optimizer

So far, we've been **manually** updating the parameters using the computed gradients. That's probably fine for *two parameters*... but what if we had a **whole lot of them**?! We use one of PyTorch's **optimizers**, like **SGD** or **Adam**.

An optimizer takes the **parameters** we want to update, the **learning rate** we want to use (and possibly many other hyper-parameters as well!) and **performs the updates** through its `step()` method.

Besides, we also don't need to zero the gradients one by one anymore. We just invoke the optimizer's `zero_grad()` method and that's it!

In the code below, we create a *Stochastic Gradient Descent* (SGD) optimizer to update our parameters **a** and **b**.

*Don't be fooled by the **optimizer**'s name: if we use **all training data** at once for the update — as we are actually doing in the code — the optimizer is performing a **batch** gradient descent, despite of its name.*

```

1 torch.manual_seed(42)
2 a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
3 b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
4 print(a, b)
5
6 lr = 1e-1
7 n_epochs = 1000
8
9 # Defines a SGD optimizer to update the parameters
10 optimizer = optim.SGD([a, b], lr=lr)
11
12 for epoch in range(n_epochs):
13     yhat = a + b * x_train_tensor
14     error = y_train_tensor - yhat
15     loss = (error ** 2).mean()
16
17     loss.backward()
18
19     # No more manual update!
20     # with torch.no_grad():
21     #     a -= lr * a.grad
22     #     b -= lr * b.grad
23     optimizer.step()
24
25     # No more telling PyTorch to let gradients go!
26     # a.grad.zero_()
27     # b.grad.zero_()
28     optimizer.zero_grad()
29
30 print(a, b)

```

torch101_optimizer.py hosted with ❤ by GitHub

[view raw](#)

PyTorch's optimizer in action — no more manual update of parameters!

[Get started](#)[Open in app](#)

```
# BEFORE: a, b
tensor([0.6226], device='cuda:0', requires_grad=True)
tensor([1.4505], device='cuda:0', requires_grad=True)
# AFTER: a, b
tensor([1.0235], device='cuda:0', requires_grad=True)
tensor([1.9690], device='cuda:0', requires_grad=True)
```

Cool! We've *optimized* the **optimization** process :-) What's left?

Loss

We now tackle the **loss computation**. As expected, PyTorch got us covered once again. There are many loss functions to choose from, depending on the task at hand. Since ours is a regression, we are using the Mean Square Error (MSE) loss.

*Notice that `nn.MSELoss` actually **creates a loss function** for us — **it is NOT the loss function itself**. Moreover, you can specify a **reduction method** to be applied, that is, **how do you want to aggregate the results for individual points** — you can average them (`reduction='mean'`) or simply sum them up (`reduction='sum'`).*

We then **use** the created loss function later, at line 20, to compute the loss given our **predictions** and our **labels**.

Our code looks like this now:

```
1 torch.manual_seed(42)
2 a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
3 b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
4 print(a, b)
5
6 lr = 1e-1
7 n_epochs = 1000
8
9 # Defines a MSE loss function
10 loss_fn = nn.MSELoss(reduction='mean')
11
12 optimizer = optim.SGD([a, b], lr=lr)
13
14 for epoch in range(n_epochs):
15     yhat = a + b * x_train_tensor
16
17     # No more manual loss!
18     # error = y_tensor - yhat
19     # loss = (error ** 2).mean()
20     loss = loss_fn(y_train_tensor, yhat)
21
22     loss.backward()
23     optimizer.step()
24     optimizer.zero_grad()
25
26 print(a, b)
```

torch101_loss.py hosted with ❤ by GitHub

[view raw](#)

PyTorch's loss in action — no more manual loss computation!

[Get started](#)[Open in app](#)

Model

In PyTorch, a **model** is represented by a regular **Python class** that inherits from the **`Module`** class.

The most fundamental methods it needs to implement are:

- `__init__(self)` : it defines the parts that make up the model—in our case, two parameters, **a** and **b**.

*You are **not** limited to defining **parameters**, though... **models can contain other models (or layers) as its attributes** as well, so you can easily nest them. We'll see an example of this shortly as well.*

- `forward(self, x)` : it performs the **actual computation**, that is, it **outputs a prediction**, given the input **x**.

*You should **NOT** call the `forward(x)` method, though. You should **call the whole model itself**, as in `model(x)` to perform a forward pass and output predictions.*

Let's build a proper (yet simple) model for our regression task. It should look like this:

```

1  class ManualLinearRegression(nn.Module):
2      def __init__(self):
3          super().__init__()
4          # To make "a" and "b" real parameters of the model, we need to wrap them with nn
5          self.a = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))
6          self.b = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))
7
8      def forward(self, x):
9          # Computes the outputs / predictions
10         return self.a + self.b * x

```

torch101_manual_model.py hosted with ❤ by GitHub

[view raw](#)

Building our "Manual" model, creating parameter by parameter!

In the `__init__` method, we define our **two parameters, a and b**, using the `Parameter()` class, to tell PyTorch these **tensors should be considered parameters of the model they are an attribute of**.

Why should we care about that? By doing so, we can use our model's `parameters()` method to retrieve **an iterator over all model's parameters, even** those parameters of **nested models**, that we can use to feed our optimizer (instead of building a list of parameters ourselves!).

Moreover, we can get the **current values for all parameters** using our model's `state_dict()` method.

IMPORTANT: we need to **send our model to the same device where the data is**. If our data is made of GPU tensors, our model must "live" inside the GPU as well.

Get started

Open in app



```

1  torch.manual_seed(42)
2
3  # Now we can create a model and send it at once to the device
4  model = ManualLinearRegression().to(device)
5  # We can also inspect its parameters using its state_dict
6  print(model.state_dict())
7
8  lr = 1e-1
9  n_epochs = 1000
10
11 loss_fn = nn.MSELoss(reduction='mean')
12 optimizer = optim.SGD(model.parameters(), lr=lr)
13
14 for epoch in range(n_epochs):
15     # What is this?!?
16     model.train()
17
18     # No more manual prediction!
19     # yhat = a + b * x_tensor
20     yhat = model(x_train_tensor)
21
22     loss = loss_fn(y_train_tensor, yhat)
23     loss.backward()
24     optimizer.step()
25     optimizer.zero_grad()
26
27 print(model.state_dict())

```

torch101_model.py hosted with ❤ by GitHub

[view raw](#)

PyTorch's model in action — no more manual prediction/forward step!

Now, the printed statements will look like this — final values for parameters **a** and **b** are still the same, so everything is ok :-)

```

OrderedDict([('a', tensor([0.3367], device='cuda:0')), ('b',
tensor([0.1288], device='cuda:0'))])
OrderedDict([('a', tensor([1.0235], device='cuda:0')), ('b',
tensor([1.9690], device='cuda:0'))])

```

I hope you noticed one particular statement in the code, to which I assigned a comment “What is this?!?” — `model.train()`.

*In PyTorch, models have a `train()` method which, somewhat disappointingly, **does NOT perform a training step**. Its only purpose is to **set the model to training mode**. Why is this important? Some models may use mechanisms like **Dropout**, for instance, which have **distinct behaviors in training and evaluation phases**.*

Nested Models

In our model, we manually created two parameters to perform a linear regression. Let's use PyTorch's **Linear** model as an attribute of our own, thus creating a nested model.

Even though this clearly is a contrived example, as we are pretty much wrapping the underlying model without adding anything useful (or, at all!) to it, it illustrates well the

[Get started](#)[Open in app](#)

in the `__init__` method, we created an attribute that contains our nested `Linear` model.

In the `forward()` method, we call the nested model itself to perform the forward pass (notice, we are **not** calling `self.linear.forward(x)`!).

```
1 class LayerLinearRegression(nn.Module):
2     def __init__(self):
3         super().__init__()
4         # Instead of our custom parameters, we use a Linear layer with single input and s
5         self.linear = nn.Linear(1, 1)
6
7     def forward(self, x):
8         # Now it only takes a call to the layer to make predictions
9         return self.linear(x)
```

torch101_layer_model.py hosted with ❤ by GitHub

[view raw](#)

Building a model using PyTorch's Linear layer

Now, if we call the `parameters()` method of this model, **PyTorch will figure the parameters of its attributes in a recursive way**. You can try it yourself using something like: `[*LayerLinearRegression().parameters()]` to get a list of all parameters. You can also add new `Linear` attributes and, even if you don't use them at all in the forward pass, they will **still** be listed under `parameters()`.

Sequential Models

Our model was simple enough... You may be thinking: “*why even bother to build a class for it?!*” Well, you have a point...

For **straightforward models**, that use **run-of-the-mill layers**, where the output of a layer is sequentially fed as an input to the next, we can use a, er... **Sequential** model :-)

In our case, we would build a Sequential model with a single argument, that is, the `Linear` layer we used to train our linear regression. The model would look like this:

```
# Alternatively, you can use a Sequential model
model = nn.Sequential(nn.Linear(1, 1)).to(device)
```

Simple enough, right?

Training Step

So far, we've defined an **optimizer**, a **loss function** and a **model**. Scroll up a bit and take a quick look at the code *inside the loop*. Would it **change** if we were using a **different optimizer**, or **loss**, or even **model**? If not, how can we make it **more generic**?

Well, I guess we could say all these lines of code **perform a training step**, given those **three elements** (*optimizer, loss and model*), the **features** and the **labels**.

So, how about **writing a function that takes those three elements and returns another function that performs a training step**, taking a set of features and labels as arguments and returning the corresponding loss?

[Get started](#)[Open in app](#)

training loop is now?

```
1 def make_train_step(model, loss_fn, optimizer):
2     # Builds function that performs a step in the train loop
3     def train_step(x, y):
4         # Sets model to TRAIN mode
5         model.train()
6         # Makes predictions
7         yhat = model(x)
8         # Computes loss
9         loss = loss_fn(y, yhat)
10        # Computes gradients
11        loss.backward()
12        # Updates parameters and zeroes gradients
13        optimizer.step()
14        optimizer.zero_grad()
15        # Returns the loss
16        return loss.item()
17
18    # Returns the function that will be called inside the train loop
19    return train_step
20
21 # Creates the train_step function for our model, loss function and optimizer
22 train_step = make_train_step(model, loss_fn, optimizer)
23 losses = []
24
25 # For each epoch...
26 for epoch in range(n_epochs):
27     # Performs one train step and returns the corresponding loss
28     loss = train_step(x_train_tensor, y_train_tensor)
29     losses.append(loss)
30
31 # Checks model's parameters
32 print(model.state_dict())
```

torch101_train_step.py hosted with ❤ by GitHub

[view raw](#)

Building a function to perform one step of training!

Let's give our training loop a rest and focus on our **data** for a while... so far, we've simply used our *Numpy arrays* turned **PyTorch tensors**. But we can do better, we can build a...

Dataset

In PyTorch, a **dataset** is represented by a regular **Python class** that inherits from the **Dataset** class. You can think of it as a kind of a Python **list of tuples**, each tuple corresponding to **one point (features, label)**.

The most fundamental methods it needs to implement are:

- `__init__(self)` : it takes **whatever arguments** needed to build a **list of tuples** — it may be the name of a CSV file that will be loaded and processed; it may be *two tensors*, one for features, another one for labels; or anything else, depending on the task at hand.

*There is **no need to load the whole dataset in the constructor method** (`__init__`). If your **dataset is big** (tens of thousands of image files, for instance), loading it at once would*

Get started

Open in app



- `__getitem__(self, index)` : it allows the dataset to be **indexed**, so it can work **like a list** (`dataset[i]`) — it must **return a tuple (features, label)** corresponding to the requested data point. We can either return the **corresponding slices** of our **pre-loaded** dataset or tensors or, as mentioned above, **load them on demand** (like in this [example](#)).
- `__len__(self)` : it should simply return the **size** of the whole dataset so, whenever it is sampled, its indexing is limited to the actual size.

Let's build a simple custom dataset that takes two tensors as arguments: one for the features, one for the labels. For any given index, our dataset class will return the corresponding slice of each of those tensors. It should look like this:

```

1  from torch.utils.data import Dataset, TensorDataset
2
3  class CustomDataset(Dataset):
4      def __init__(self, x_tensor, y_tensor):
5          self.x = x_tensor
6          self.y = y_tensor
7
8      def __getitem__(self, index):
9          return (self.x[index], self.y[index])
10
11     def __len__(self):
12         return len(self.x)
13
14     # Wait, is this a CPU tensor now? Why? Where is .to(device)?
15     x_train_tensor = torch.from_numpy(x_train).float()
16     y_train_tensor = torch.from_numpy(y_train).float()
17
18     train_data = CustomDataset(x_train_tensor, y_train_tensor)
19     print(train_data[0])
20
21     train_data = TensorDataset(x_train_tensor, y_train_tensor)
22     print(train_data[0])

```

torch101_dataset.py hosted with ❤ by GitHub

[view raw](#)

Creating datasets using train tensors

Once again, you may be thinking “*why go through all this trouble to wrap a couple of tensors in a class?*”. And, once again, you do have a point... if a dataset is nothing else but a **couple of tensors**, we can use PyTorch's **TensorDataset** class, which will do pretty much what we did in our custom dataset above.

*Did you notice we built our **training tensors** out of Numpy arrays but we **did not send them to a device**? So, they are **CPU** tensors now! **Why**?*

*We **don't** want our **whole training data** to be loaded into **GPU tensors**, as we have been doing in our example so far, because it **takes up space** in our precious **graphics card's RAM**.*

OK, fine, but then again, **why** are we building a dataset anyway? We're doing it because we want to use a...

[Get started](#)[Open in app](#)

batch gradient descent all along. This is fine for our *ridiculously small dataset*, sure, but if we want to go serious about all this, we **must** use **mini-batch** gradient descent. Thus, we need mini-batches. Thus, we need to **slice** our dataset accordingly. Do you want to do it *manually*?! Me neither!

So we use PyTorch's **DataLoader** class for this job. We tell it which **dataset** to use (the one we just built in the previous section), the desired **mini-batch size** and if we'd like to **shuffle** it or not. That's it!

Our **loader** will behave like an **iterator**, so we can **loop over it** and **fetch a different mini-batch** every time.

```
1 from torch.utils.data import DataLoader
2
3 train_loader = DataLoader(dataset=train_data, batch_size=16, shuffle=True)
```

torch101_loader.py hosted with ❤ by GitHub

[view raw](#)

Building a data loader for our training data

To retrieve a sample mini-batch, one can simply run the command below — it will return a list containing two tensors, one for the features, another one for the labels.

```
next(iter(train_loader))
```

How does this change our training loop? Let's check it out!

```
1 losses = []
2 train_step = make_train_step(model, loss_fn, optimizer)
3
4 for epoch in range(n_epochs):
5     for x_batch, y_batch in train_loader:
6         # the dataset "lives" in the CPU, so do our mini-batches
7         # therefore, we need to send those mini-batches to the
8         # device where the model "lives"
9         x_batch = x_batch.to(device)
10        y_batch = y_batch.to(device)
11
12        loss = train_step(x_batch, y_batch)
13        losses.append(loss)
14
15    print(model.state_dict())
```

torch101_minibatch.py hosted with ❤ by GitHub

[view raw](#)

Using mini-batch gradient descent!

Two things are different now: not only we have an *inner loop* to load each and every *mini-batch* from our **DataLoader** but, more importantly, we are now **sending only one mini-batch to the device**.

*For bigger datasets, loading data sample by sample (into a CPU tensor) using Dataset's **__getitem__** and then sending all samples that belong to the same mini-batch at once*

[Get started](#)[Open in app](#)

Moreover, if you have **many GPUs** to train your model on, it is best to keep your dataset “agnostic” and assign the batches to different GPUs during training.

So far, we’ve focused on the **training data** only. We built a *dataset* and a *data loader* for it. We could do the same for the **validation** data, using the **split** we performed at the beginning of this post... or we could use `random_split` instead.

Random Split

PyTorch’s `random_split()` method is an easy and familiar way of performing a **training-validation split**. Just keep in mind that, in our example, we need to apply it to the **whole dataset** (*not the training dataset* we built in two sections ago).

Then, for each subset of data, we build a corresponding `DataLoader`, so our code looks like this:

```
1 from torch.utils.data.dataset import random_split
2
3 x_tensor = torch.from_numpy(x).float()
4 y_tensor = torch.from_numpy(y).float()
5
6 dataset = TensorDataset(x_tensor, y_tensor)
7
8 train_dataset, val_dataset = random_split(dataset, [80, 20])
9
10 train_loader = DataLoader(dataset=train_dataset, batch_size=16)
11 val_loader = DataLoader(dataset=val_dataset, batch_size=20)
```

torch101_split.py hosted with ❤ by GitHub

[view raw](#)

Splitting the dataset into training and validation sets, the PyTorch way!

Now we have a **data loader** for our **validation set**, so, it makes sense to use it for the...

Evaluation

This is the **last** part of our journey — we need to change the training loop to include the **evaluation of our model**, that is, computing the **validation loss**. The first step is to include another inner loop to handle the *mini-batches* that come from the *validation loader*, sending them to the same *device* as our model. Next, we make **predictions** using our model (line 23) and compute the corresponding **loss** (line 24).

That’s pretty much it, but there are **two small, yet important**, things to consider:

- `torch.no_grad()`: even though it won’t make a difference in our simple model, it is a **good practice to wrap the validation** inner loop with this **context manager to disable any gradient calculation** that you may inadvertently trigger — **gradients belong in training**, not in validation steps;
- `eval()`: the only thing it does is **setting the model to evaluation mode** (just like its `train()` counterpart did), so the model can adjust its behavior regarding some operations, like **Dropout**.

Now, our training loop should look like this:

Get started

Open in app



```

4
5  for epoch in range(n_epochs):
6      for x_batch, y_batch in train_loader:
7          x_batch = x_batch.to(device)
8          y_batch = y_batch.to(device)
9
10         loss = train_step(x_batch, y_batch)
11         losses.append(loss)
12
13     with torch.no_grad():
14         for x_val, y_val in val_loader:
15             x_val = x_val.to(device)
16             y_val = y_val.to(device)
17
18             model.eval()
19
20             yhat = model(x_val)
21             val_loss = loss_fn(y_val, yhat)
22             val_losses.append(val_loss.item())
23
24  print(model.state_dict())

```

torch101_validation.py hosted with ❤ by GitHub

[view raw](#)

Computing validation loss

Is there **anything else** we can improve or change? Sure, there is **always something else** to add to your model — using a **learning rate scheduler**, for instance. But this post is already *waaaaay too long*, so I will stop right here.

“Where is the full working code with all bells and whistles?”, you ask? You can find it **here**.

Final Thoughts

Although this post was *much longer* than I anticipated when I started writing it, I wouldn't make it any different — I believe it has **most of the necessary steps** one needs go to trough in order to **learn**, in a **structured** and **incremental** way, how to **develop Deep Learning models using PyTorch**.

Hopefully, after finishing working through all code in this post, you'll be able to better appreciate and more easily work your way through PyTorch's official **tutorials**.

*Update (May 18th, 2021): Today I've finished my book: **Deep Learning with PyTorch Step-by-Step: A Beginner's Guide**.*

TWO LAST CHAPTERS RELEASED!
"Transform and Roll Out"
 &
"Down the Yellow Brick Rabbit Hole"

Read a FREE Sample at
leanpub.com/pytorch


[Get started](#)[Open in app](#)

If you have any thoughts, comments or questions, please leave a comment below or contact me on [Twitter](#).

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. [Take a look.](#)

 Get this newsletter

[Machine Learning](#)[Deep Learning](#)[Pytorch](#)[Towards Data Science](#)[Tutorial](#)[About](#) [Write](#) [Help](#) [Legal](#)

Get the Medium app

