UNC CHARLOTTE

The WILLIAM STATES LEE COLLEGE of ENGINEERING

Introduction to ML
Lecture 12: Gradient Descent, BackPropagation in Pytorch

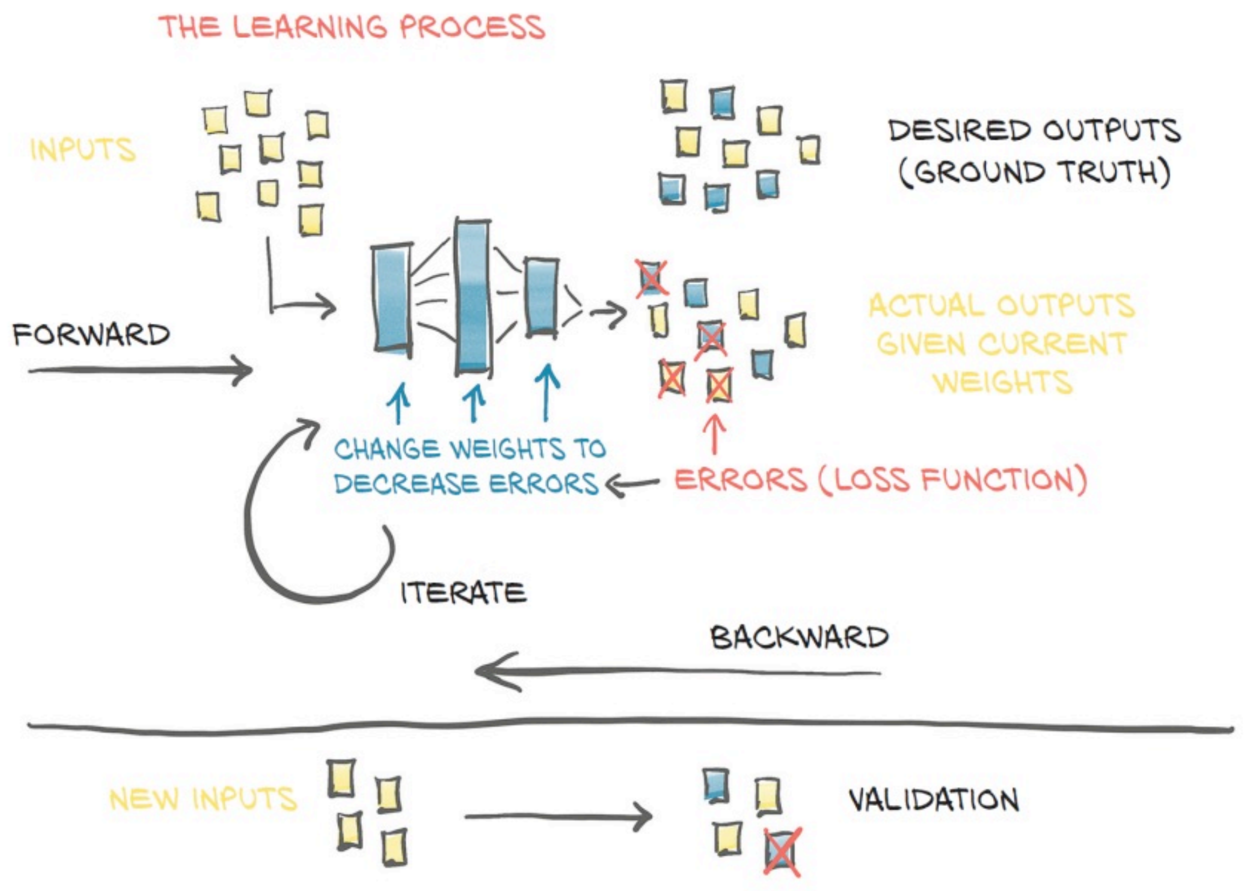
Hamed Tabkhi

Department of Electrical and Computer Engineering,

University of North Carolina Charlotte (UNCC)

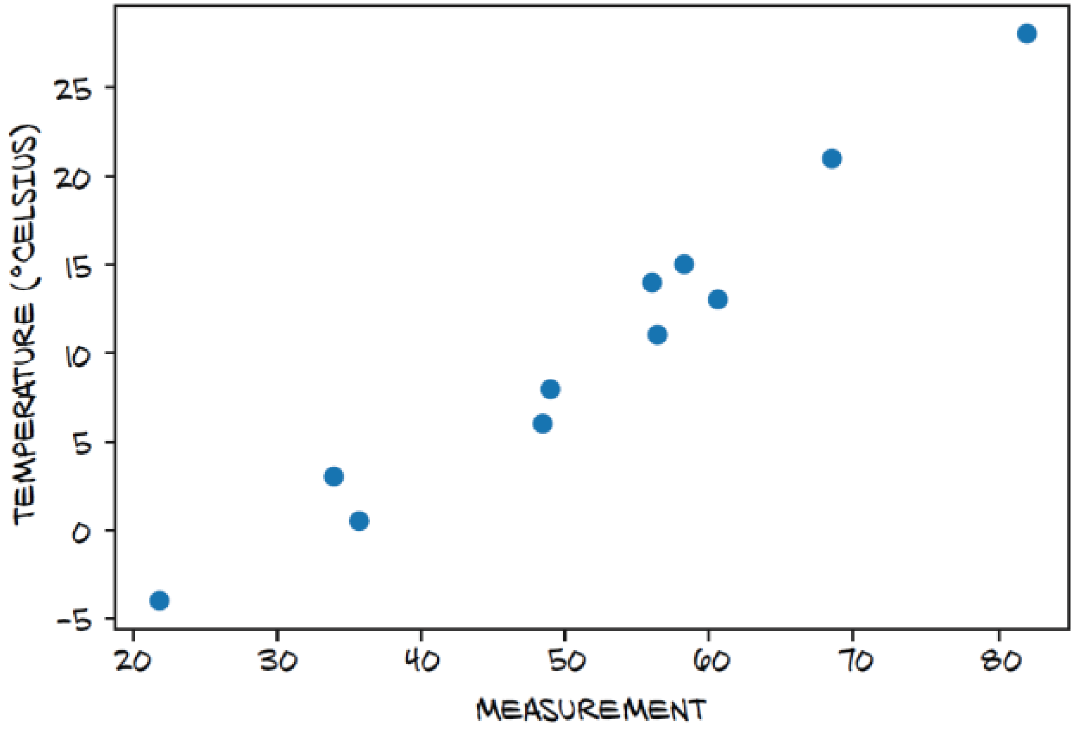
htabkhiv@uncc.edu

General Supervised Learning Framework


THE LEARNING PROCESS
INPUTS
DESIRED OUTPUTS (GROUND TRUTH)
FORWARD
ACTUAL OUTPUTS GIVEN CURRENT WEIGHTS
CHANGE WEIGHTS TO DECREASE ERRORS
ERRORS (LOSS FUNCTION)
ITERATE
BACKWARD
NEW INPUTS
VALIDATION

Example

Goal: Predicting temperature based on some measured values.



```
# In[2]:
t_c = [0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0, 21.0]
t_u = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4]
t_c = torch.tensor(t_c)
t_u = torch.tensor(t_u)
```

Linear Model

```
t_c = w * t_u + b
```

Finding a linear relationship between t_u and t_C

Pytorch code:

```
    def model(t_u, w, b):
    return w * t_u + b
```

Aim: finding a linear relation shop between the input and the desired output.

Loss Calculation

How to calculate the loss:

`|t_p - t_c|` and `(t_p - t_c)^2`.

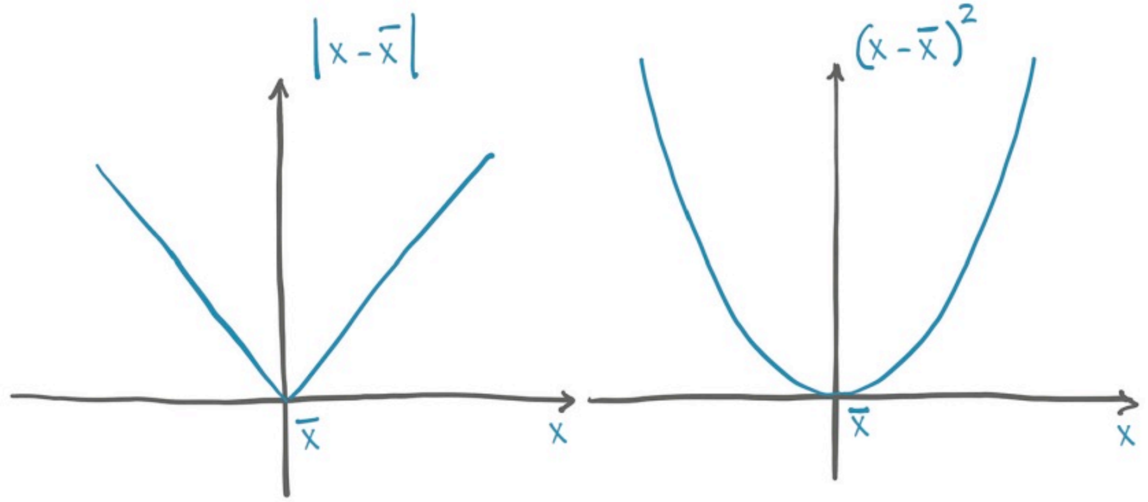
Note: Loss should be a positive number

The square of the differences behaves more nicely around the minimum.

The square difference also penalizes wildly wrong results more than the absolute difference does.

```
def loss_fn(t_p, t_c):
squared_diffs = (t_p - t_c)**2
return squared_diffs.mean()
```

Note: this is the average loss



Loss Calculation

```
# In[5]:
w = torch.ones(())                initial W is 1
b = torch.zeros(())              initial b is 0
t_p = model(t_u, w, b)
t_p
# Out[5]:
tensor([35.7000, 55.9000, 58.2000, 81.9000, 56.3000, 48.9000, 33.9000,
21.8000, 48.4000, 60.4000, 68.4000])
```

- And check the value of the loss:

```
# In[6]:
loss = loss_fn(t_p, t_c)
loss
# Out[6]:
tensor(1763.8846)
```

Autograd: Computing the gradient automatically

- This is when PyTorch tensors come to the rescue, with a PyTorch component called *Autograd*.

```
# In[3]:
def model(t_u, w, b):
return w * t_u + b

# In[4]:
def loss_fn(t_p, t_c):
squared_diffs = (t_p - t_c)**2
return squared_diffs.mean()

# In[5]:
params = torch.tensor([1.0, 0.0],
requires_grad=True)
```

Autograd

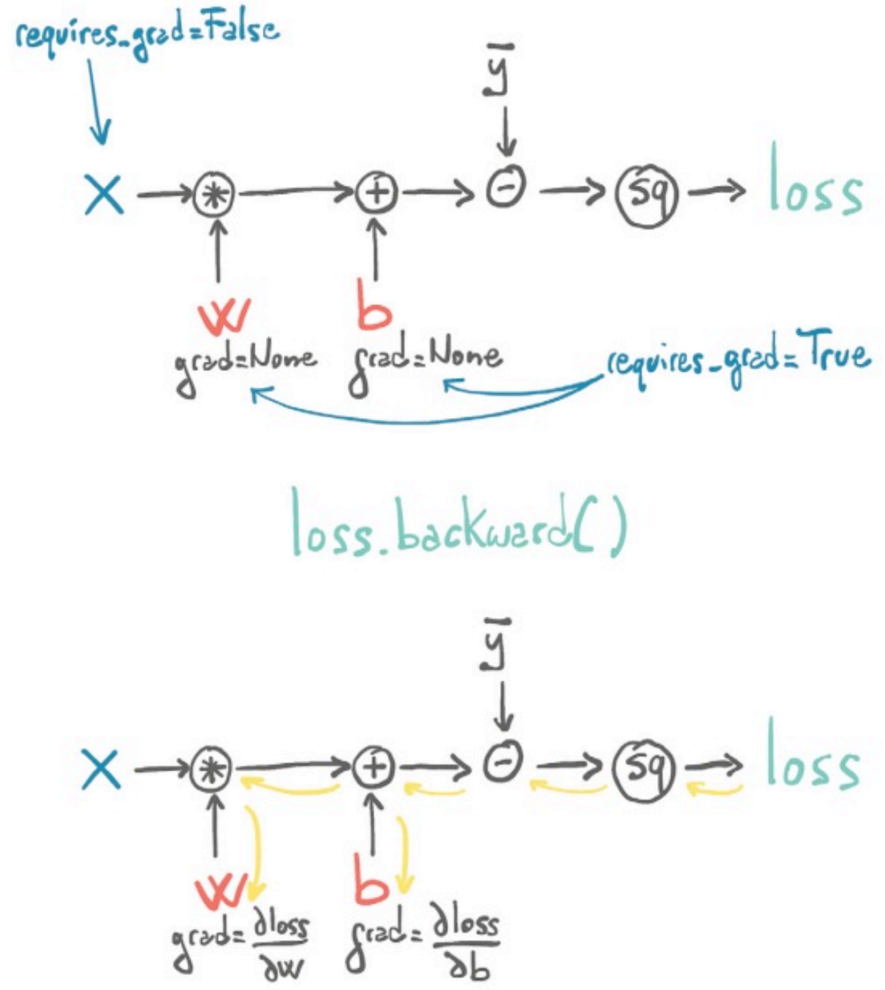
- `requires_grad=True is` telling PyTorch to track the entire family tree of tensors resulting from operations on all `params` involved in the model.
- All we have to do is to start with `params` tensor with `requires_grad` set to `True`, then call the model and compute the loss, and then call `backward` on the `loss` tensor:

```
# In[7]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
loss = loss_fn(model(t_u, *params), t_c)
loss.backward()
params.grad
# Out[7]:
tensor([4517.2969, 82.6000])
```

At this point, the `grad` attribute of `params` contains the derivatives of the loss with respect to each element of params.

The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

Autograd and Forward and Backward Graphs

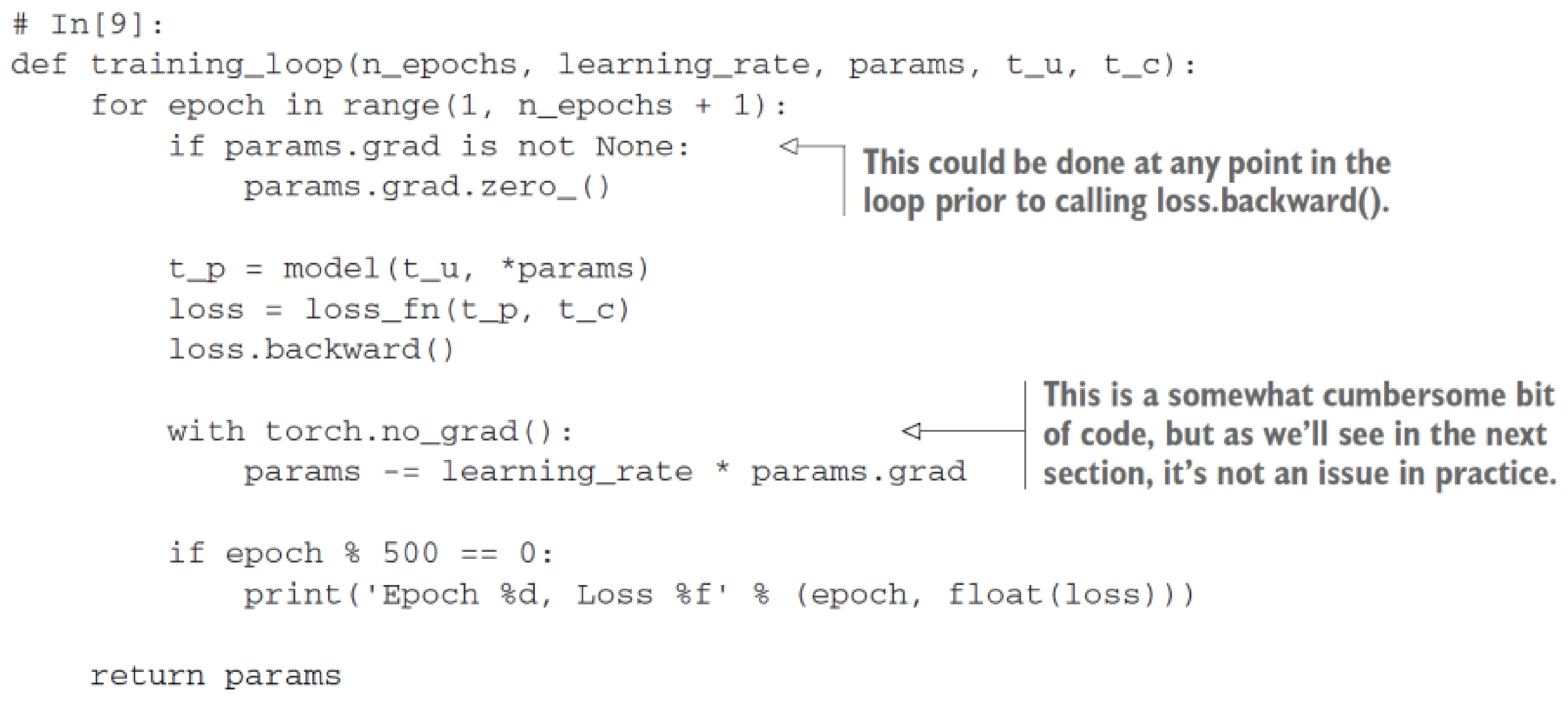


- **Forward:** when we compute our `loss` while the parameters `w` and `b` require gradients, in addition to performing the actual computation, PyTorch creates the autograd graph with the operations (in black circles) as nodes.

- **Backward:** when we call `loss.backward()`, PyTorch traverses this graph in the reverse direction to compute the gradients, as shown by the arrows

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING
UNC CHARLOTTE

Putting Everything together with Autograd



```
# In[9]:
def training_loop(n_epochs, learning_rate, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        if params.grad is not None:
            params.grad.zero_()

        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)
        loss.backward()

        with torch.no_grad():
            params -= learning_rate * params.grad

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params
```

This could be done at any point in the loop prior to calling loss.backward().

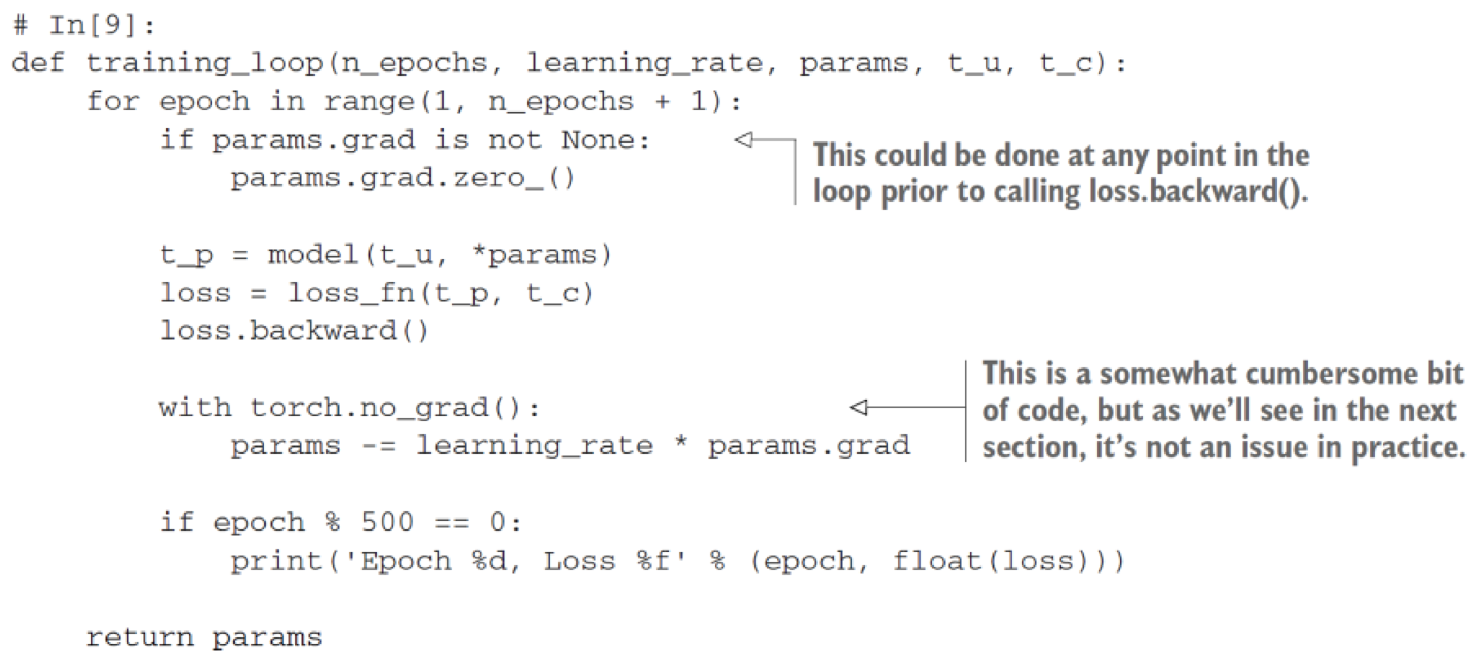
This is a somewhat cumbersome bit of code, but as we'll see in the next section, it's not an issue in practice.

Calling `backward` will lead derivatives to *accumulate* at leaf nodes. We need to *zero the gradient explicitly* per each iteration of training.

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING
UNC CHARLOTTE

Putting Everything together with Autograd

- We are encapsulating the parameters update in a `no_grad` context using the Python `with` statement. This means within the `with` block, the PyTorch autograd mechanism will not be applied.

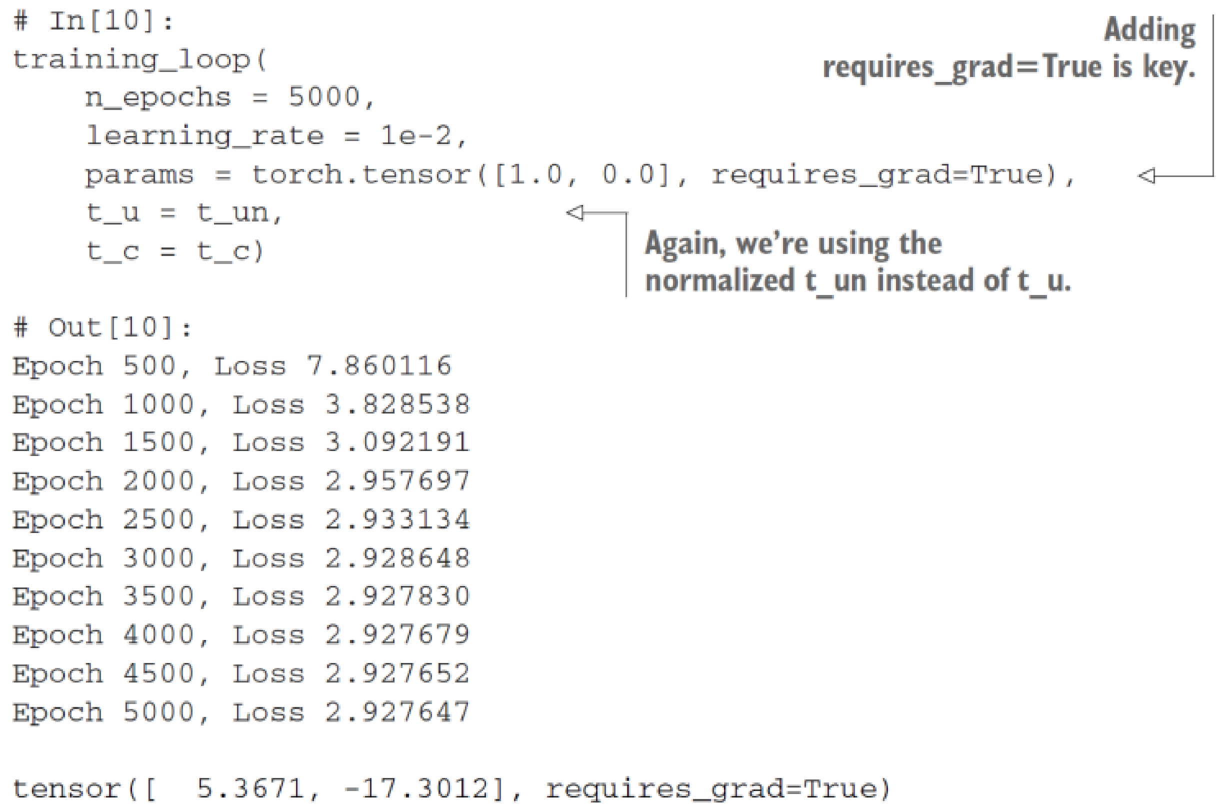


```
# In[9]:
def training_loop(n_epochs, learning_rate, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        if params.grad is not None:
            params.grad.zero_()

        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)
        loss.backward()

        with torch.no_grad():
            params -= learning_rate * params.grad

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params
```

This could be done at any point in the loop prior to calling loss.backward().

This is a somewhat cumbersome bit of code, but as we'll see in the next section, it's not an issue in practice.

Putting Everything together with Autograd



```
# In[10]:
training_loop(
    n_epochs = 5000,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0], requires_grad=True),
    t_u = t_un,
    t_c = t_c)
```

Adding requires_grad=True is key.

Again, we're using the normalized t_un instead of t_u.

```
# Out[10]:
Epoch 500, Loss 7.860116
Epoch 1000, Loss 3.828538
Epoch 1500, Loss 3.092191
Epoch 2000, Loss 2.957697
Epoch 2500, Loss 2.933134
Epoch 3000, Loss 2.928648
Epoch 3500, Loss 2.927830
Epoch 4000, Loss 2.927679
Epoch 4500, Loss 2.927652
Epoch 5000, Loss 2.927647

tensor([  5.3671, -17.3012], requires_grad=True)
```

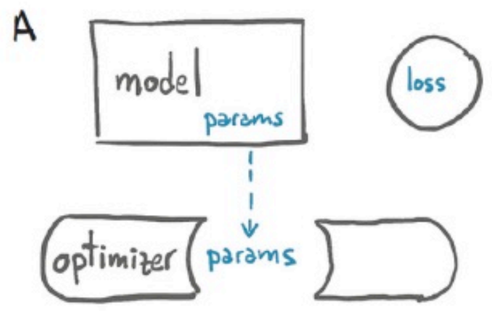
NOTE: We get the same result as before

Gradient Decent Optimizer

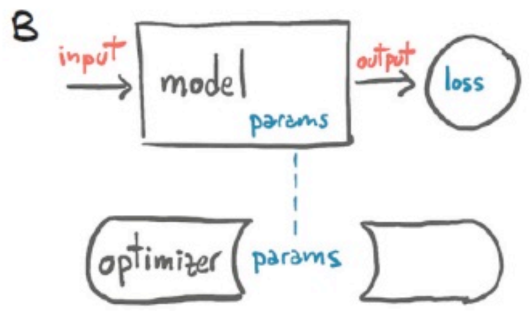
- In the example code, we used ***vanilla* gradient descent** for optimization, which worked fine for our simple case.
- Here, **vanilla** means pure / without any adulteration.
- Its main feature is that we take small steps in the direction of the minima by taking **gradient** of the cost function.
- This is the simplest form of **gradient descent** technique. For Complex Models with many parameters, more complex gradient decent optimizers can be used.

Gradient Decent Optimizer

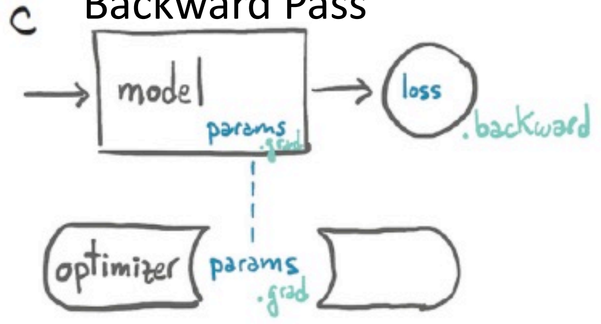
Access to Parameters



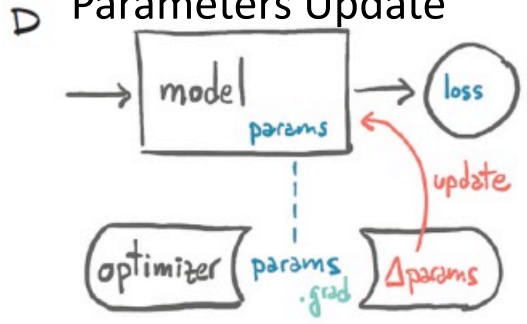
Forward Pass



Backward Pass



Parameters Update



- Every optimizer constructor takes a list of parameters (aka PyTorch tensors, typically with `requires_grad` set to `True`) as the first input.

- All parameters passed to the optimizer are retained inside the optimizer object so the optimizer can update their values and access their `grad` attribute.

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING
UNC CHARLOTTE

Gradient Decent Optimizer

- The `torch` module has an `optim` submodule where we can find classes implementing different optimization algorithms. Here’s an abridged list (code/p1ch5/3_optimizers.ipynb):

```
# In[5]:
import torch.optim as optim
dir(optim)
# Out[5]:
['ASGD',
'Adadelta',
'Adagrad',
'Adam',
'Adamax',
'LBFGS',
'Optimizer’,
'RMSprop',
'Rprop',
'SGD',
'SparseAdam',
...
]
```

- Each optimizer exposes two methods: `zero_grad` and `step`.
- **`zero_grad`** zeroes the `grad` attribute of all the parameters passed to the optimizer upon construction.
- **`step`** updates the value of those parameters according to the optimization strategy implemented by the specific optimizer.

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING
UNC CHARLOTTE

Stochastic Gradient Descent (SGD)

- The term *stochastic* comes from the fact that the gradient is typically obtained by averaging over a random subset of all input samples, called a *minibatch.*
- Actually, the optimizer itself is exactly a vanilla gradient descent (as long as the `momentum` argument is set to `0.0`, which is the default).
- The algorithm is literally the same in the two cases.
- vanilla is evaluated on all the samples

```
# In[7]:
t_p = model(t_u, *params)
loss = loss_fn(t_p, t_c)
loss.backward()
optimizer.step()
params
# Out[7]:
tensor([ 9.5483e-01, -8.2600e-04],
requires_grad=True)
```

- The value of `params` is updated upon calling `step`.
  - The optimizer looks into `params.grad` and updates `params`, subtracting `learning_rate` times `grad` from it, exactly as in our former handwrittedn code.

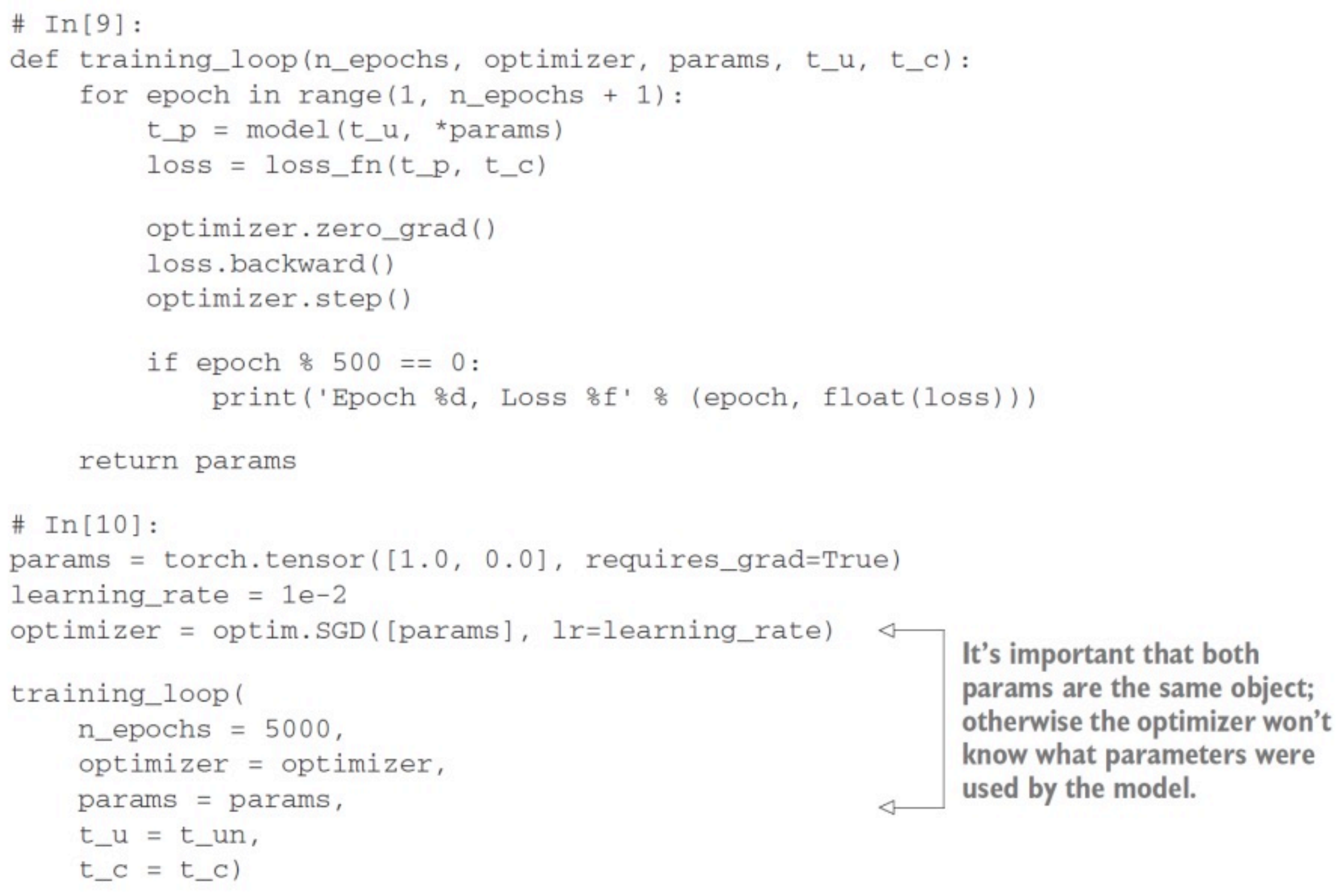
Putting Everything together with Optimizer

```
# In[8]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)

t_p = model(t_un, *params)
loss = loss_fn(t_p, t_c)

optimizer.zero_grad()
loss.backward()
optimizer.step()

params

# Out[8]:
tensor([1.7761, 0.1064], requires_grad=True)
```

As before, the exact placement of this call is somewhat arbitrary. It could be earlier in the loop as well.

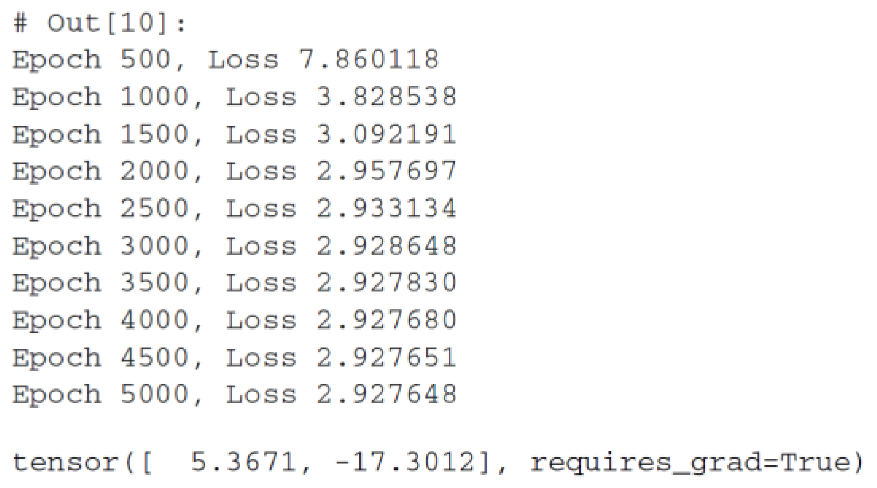
- All we have to do is provide a list of params to it (that list can be extremely long, as is needed for very deep neural network models), and we can forget about the details.

Putting Everything together with Optimizer



```
# In[9]:
def training_loop(n_epochs, optimizer, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params

# In[10]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)

training_loop(
    n_epochs = 5000,
    optimizer = optimizer,
    params = params,
    t_u = t_un,
    t_c = t_c)
```

It's important that both params are the same object; otherwise the optimizer won't know what parameters were used by the model.

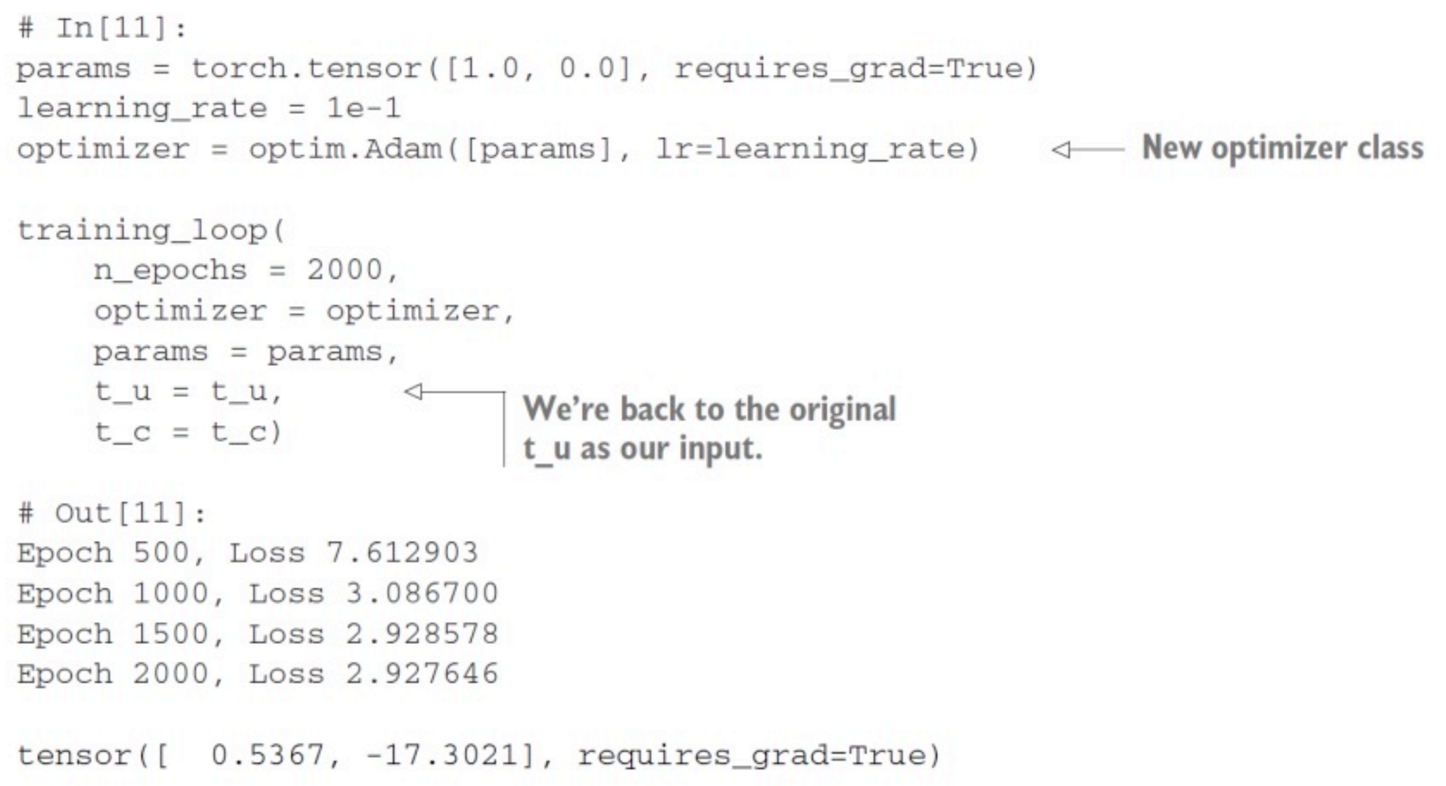


```
# Out[10]:
Epoch 500, Loss 7.860118
Epoch 1000, Loss 3.828538
Epoch 1500, Loss 3.092191
Epoch 2000, Loss 2.957697
Epoch 2500, Loss 2.933134
Epoch 3000, Loss 2.928648
Epoch 3500, Loss 2.927830
Epoch 4000, Loss 2.927680
Epoch 4500, Loss 2.927651
Epoch 5000, Loss 2.927648

tensor([  5.3671, -17.3012], requires_grad=True)
```

We get the same result as before

Adam Optimizer

- Adam Optimizer is a more sophisticated optimizer in which the learning rate is set adaptively.
- In addition, it is a lot less sensitive to the scaling of the parameter.



```
# In[11]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-1
optimizer = optim.Adam([params], lr=learning_rate)

training_loop(
    n_epochs = 2000,
    optimizer = optimizer,
    params = params,
    t_u = t_u,
    t_c = t_c)

# Out[11]:
Epoch 500, Loss 7.612903
Epoch 1000, Loss 3.086700
Epoch 1500, Loss 2.928578
Epoch 2000, Loss 2.927646

tensor([  0.5367, -17.3021], requires_grad=True)
```

New optimizer class

We're back to the original t_u as our input.

We can go back to using the original (non-normalized) input `t_u`, and even increase the learning rate to `1e-1`, and Adam won't even blink.