UNC CHARLOTTE

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING

Introduction to ML
Lecture 17: Training and Validation in Pytorch

Hamed Tabkhi

Department of Electrical and Computer Engineering,

University of North Carolina Charlotte (UNCC)

htabkhiv@uncc.edu

Overview

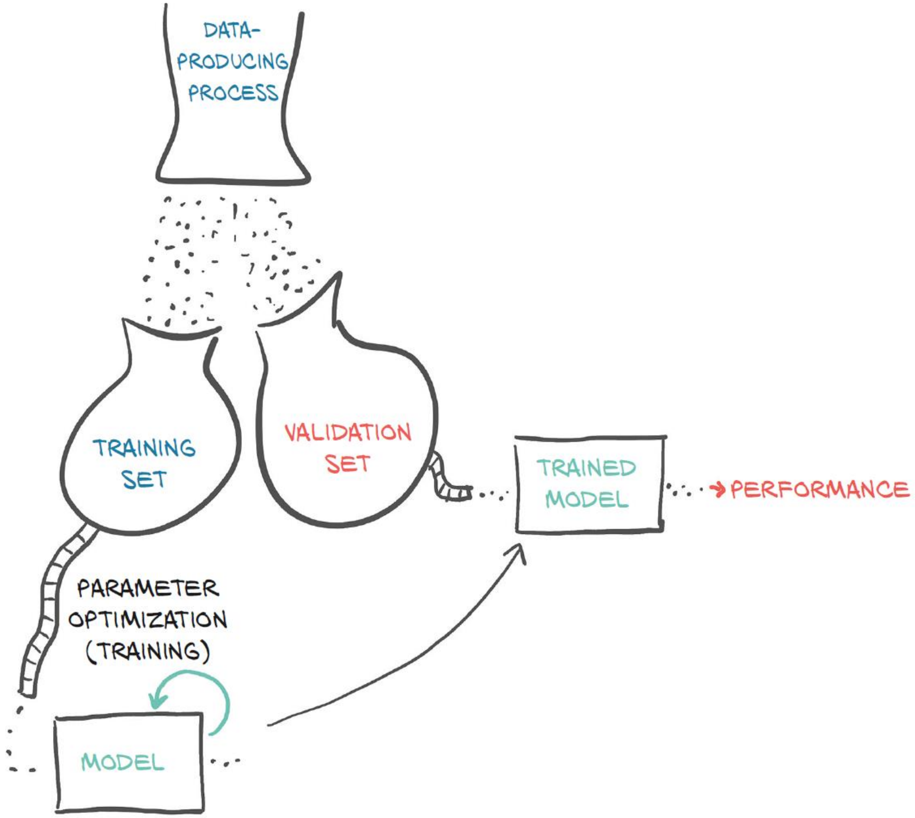
- We just saw a simple example of backpropagation:
  - we computed the gradient of a composition of functions—the linear model and the square loss—with respect to their innermost parameters (`w` and `b`) by propagating derivatives backward using the *chain rule*.
  - The basic requirement here is that all functions we're dealing with can be differentiated analytically.
- A **function** is **differentiable** at a point when there's a defined derivative at that point - The short is that a function is differentiable if you can compute its derivative.
- Even if we have a complicated model with millions of parameters, as long as our model is differentiable, computing the gradient of the loss with respect to the parameters to writing the analytical expression for the derivatives and evaluating them *once*.

Overview

- This is when PyTorch tensors come to the rescue, with a PyTorch component called *autograd*.
- PyTorch tensors can remember where they come from, in terms of the operations and parent tensors that originated them, and they can automatically provide the chain of derivatives of such operations with respect to their inputs.
- This means we won’t need to derive our model by hand.
- Given a forward expression, no matter how nested, PyTorch will automatically provide the gradient of that expression with respect to its input parameters.

Evaluating the Training Loss

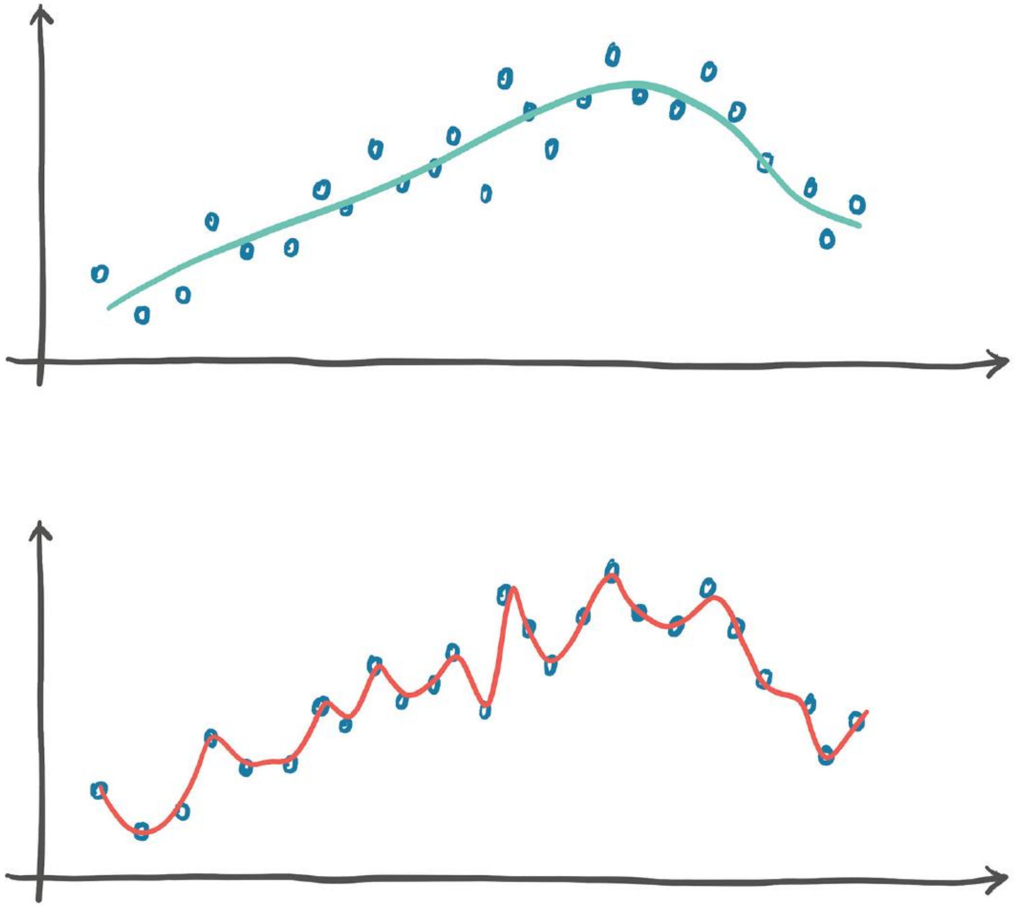
- During the training, we make sure that the loss is minimal *across training* data points, but we'll have no guarantee that the model behaves well *away from* or *in between* the data points.
- Evaluating the loss at those independent data points would yield higher-than-expected loss.
- We must take a few data points out of our dataset (the *validation set*) and only fit our model on the remaining data points (the *training set*).
- Then, while we're fitting the model, we can evaluate the loss once on the training set and once on the validation set.



Training Rules

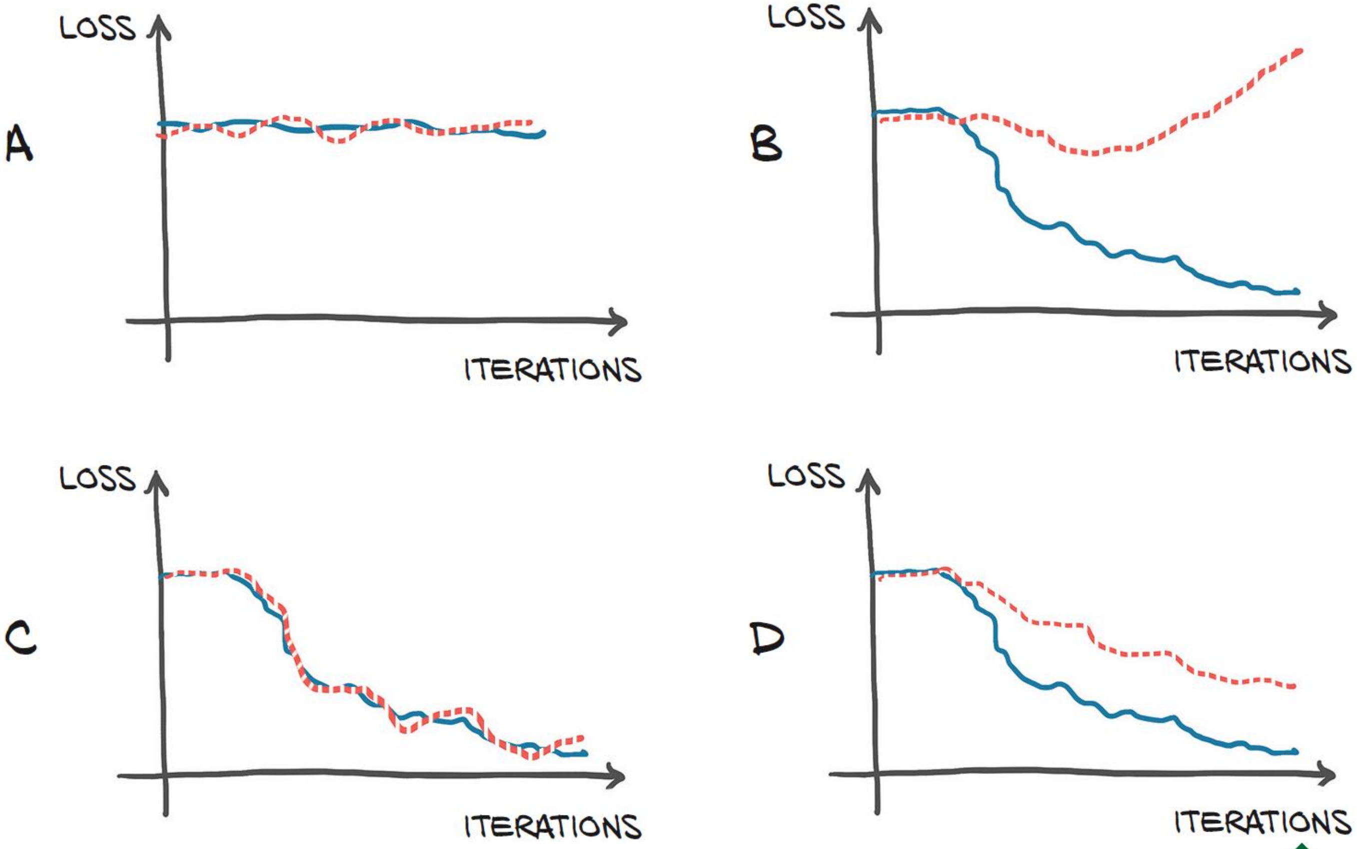
- Rule 1: if the training loss is not decreasing, chances are the model is too simple for the data. The other possibility is that our data just doesn’t contain meaningful information that lets it explain the output
- Rule 2: if the training loss and the validation loss diverge, we’re overfitting.
    - If the loss evaluated in the validation set doesn’t decrease along with the training set, it means our model is improving its fit of the samples it is seeing during training, but it is not *generalizing* to samples outside this precise set.

Overfitting



- Over fitting: evaluating the loss at those independent data points would yield
- Both a piecewise polynomial or a really large neural could generate a model meandering its way through the data points.
  - There is nothing to keep the model in check for inputs away from the training data points.
- To avoid overfitting, While training, we need to keep the model in check for inputs away from the training data points.
- We've got some nice trade-offs here. On the one hand, we need the model to have enough capacity for it to fit the training set. On the other, we need the model to avoid overfitting.

Generalization



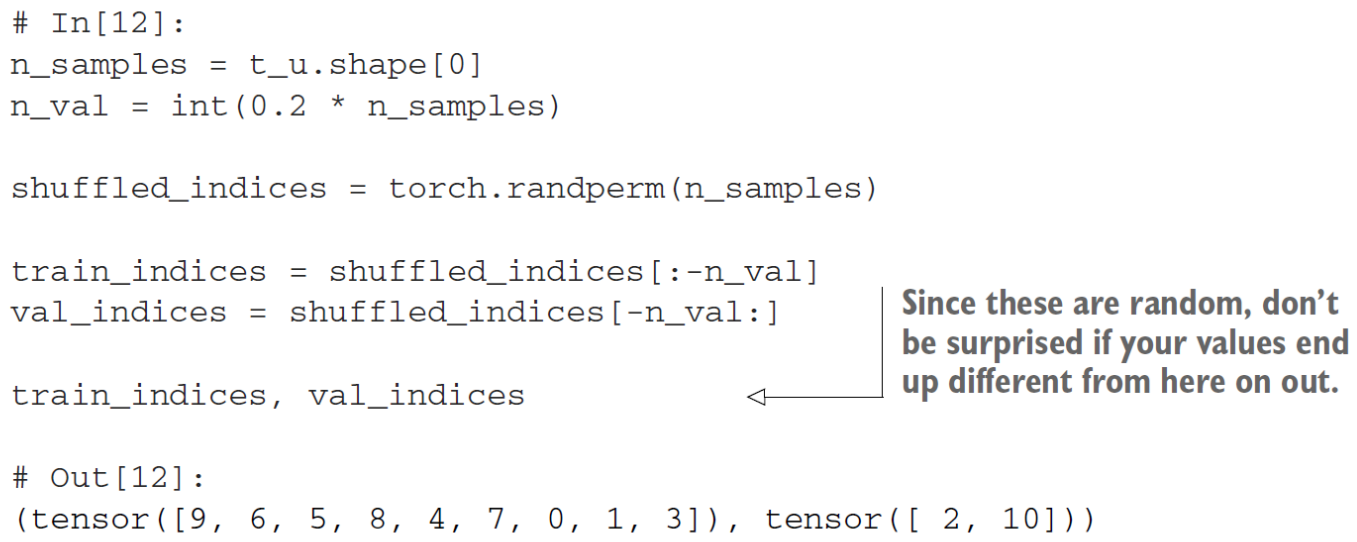
(A) Training and validation losses do not decrease; the model is not learning due to no information in the data or insufficient capacity of the model.

(B) Training loss decreases while validation loss increases: overfitting.

(C) Training and validation losses decrease exactly in tandem. Performance may be improved further as the model is not at the limit of overfitting.

(D) Training and validation losses have different absolute values but similar trends: overfitting is under control.

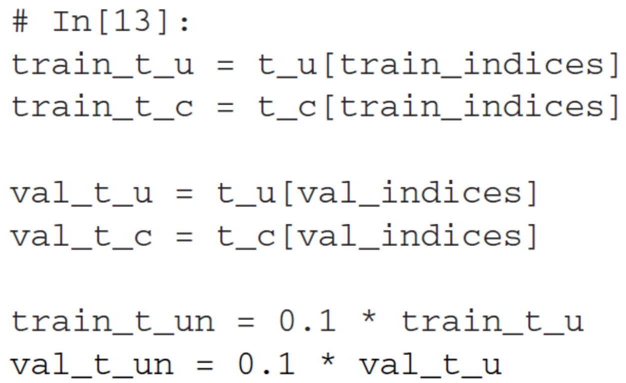
Splitting A dataset



```
# In[12]:
n_samples = t_u.shape[0]
n_val = int(0.2 * n_samples)

shuffled_indices = torch.randperm(n_samples)

train_indices = shuffled_indices[:-n_val]
val_indices = shuffled_indices[-n_val:]

train_indices, val_indices

# Out[12]:
(tensor([9, 6, 5, 8, 4, 7, 0, 1, 3]), tensor([ 2, 10]))
```

Since these are random, don't be surprised if your values end up different from here on out.

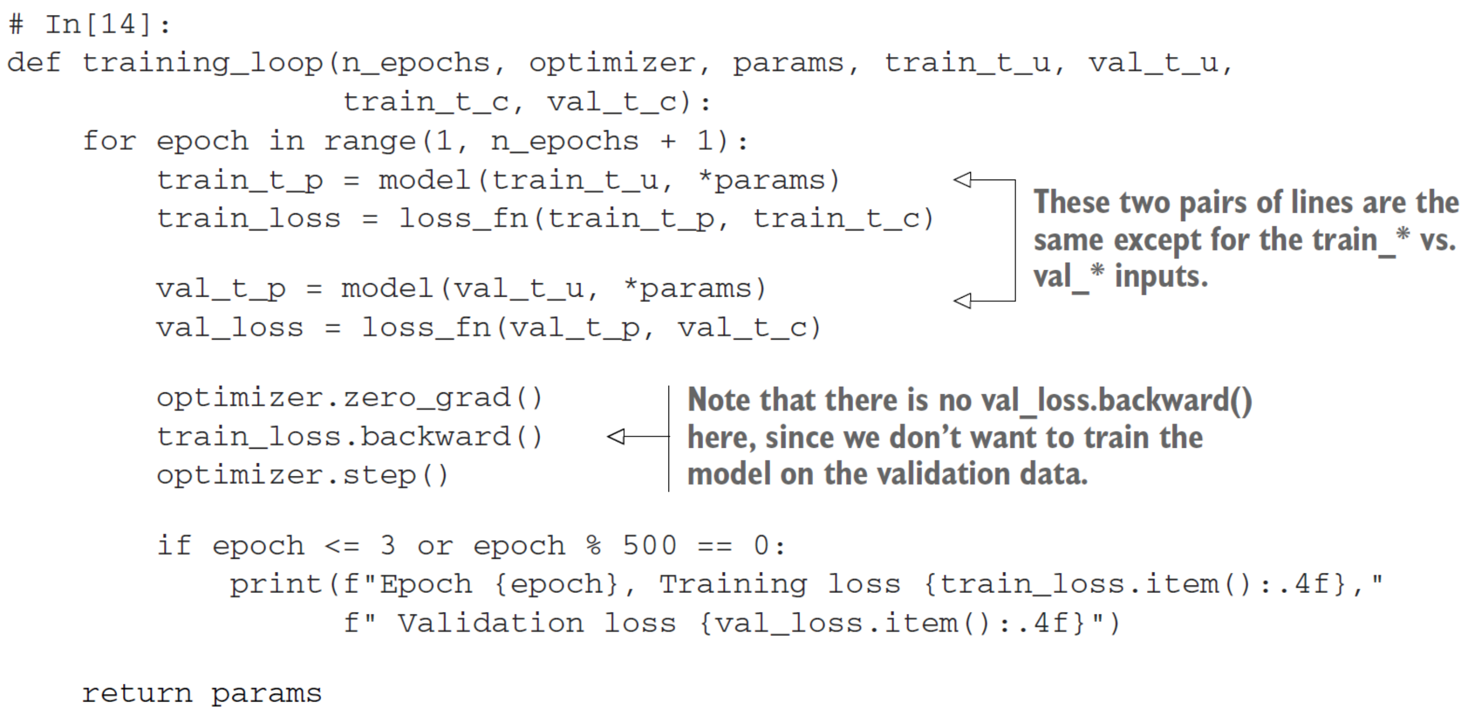
`randperm:` shuffling the elements of a tensor amounts to finding a permutation of its indices.



```
# In[13]:
train_t_u = t_u[train_indices]
train_t_c = t_c[train_indices]

val_t_u = t_u[val_indices]
val_t_c = t_c[val_indices]

train_t_un = 0.1 * train_t_u
val_t_un = 0.1 * val_t_u
```

We just got index tensors that we can use to build training and validation sets starting from the data tensors

Modified Training Loop



```
# In[14]:
def training_loop(n_epochs, optimizer, params, train_t_u, val_t_u,
                  train_t_c, val_t_c):
    for epoch in range(1, n_epochs + 1):
        train_t_p = model(train_t_u, *params)
        train_loss = loss_fn(train_t_p, train_t_c)

        val_t_p = model(val_t_u, *params)
        val_loss = loss_fn(val_t_p, val_t_c)

        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()

        if epoch <= 3 or epoch % 500 == 0:
            print(f"Epoch {epoch}, Training loss {train_loss.item():.4f},"
                  f" Validation loss {val_loss.item():.4f}")

    return params
```

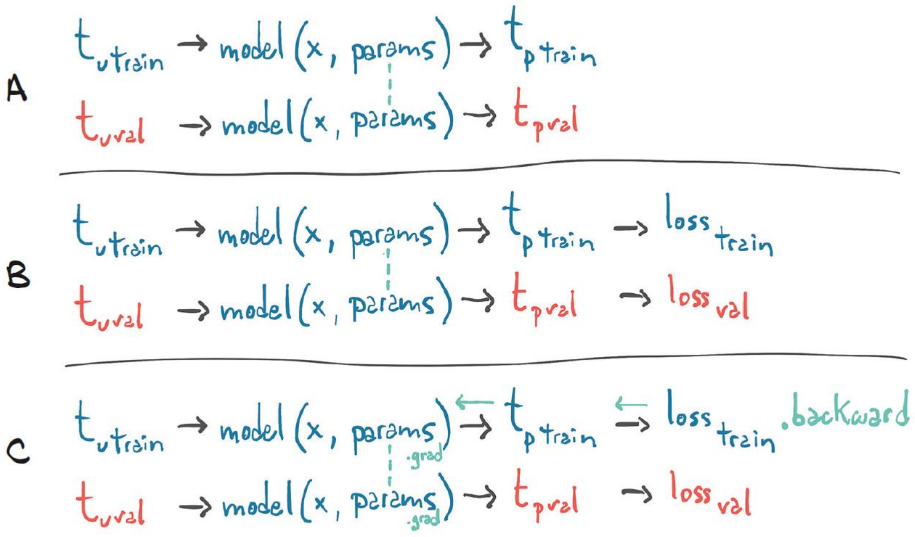
These two pairs of lines are the same except for the train_* vs. val_* inputs.

Note that there is no val_loss.backward() here, since we don't want to train the model on the validation data.

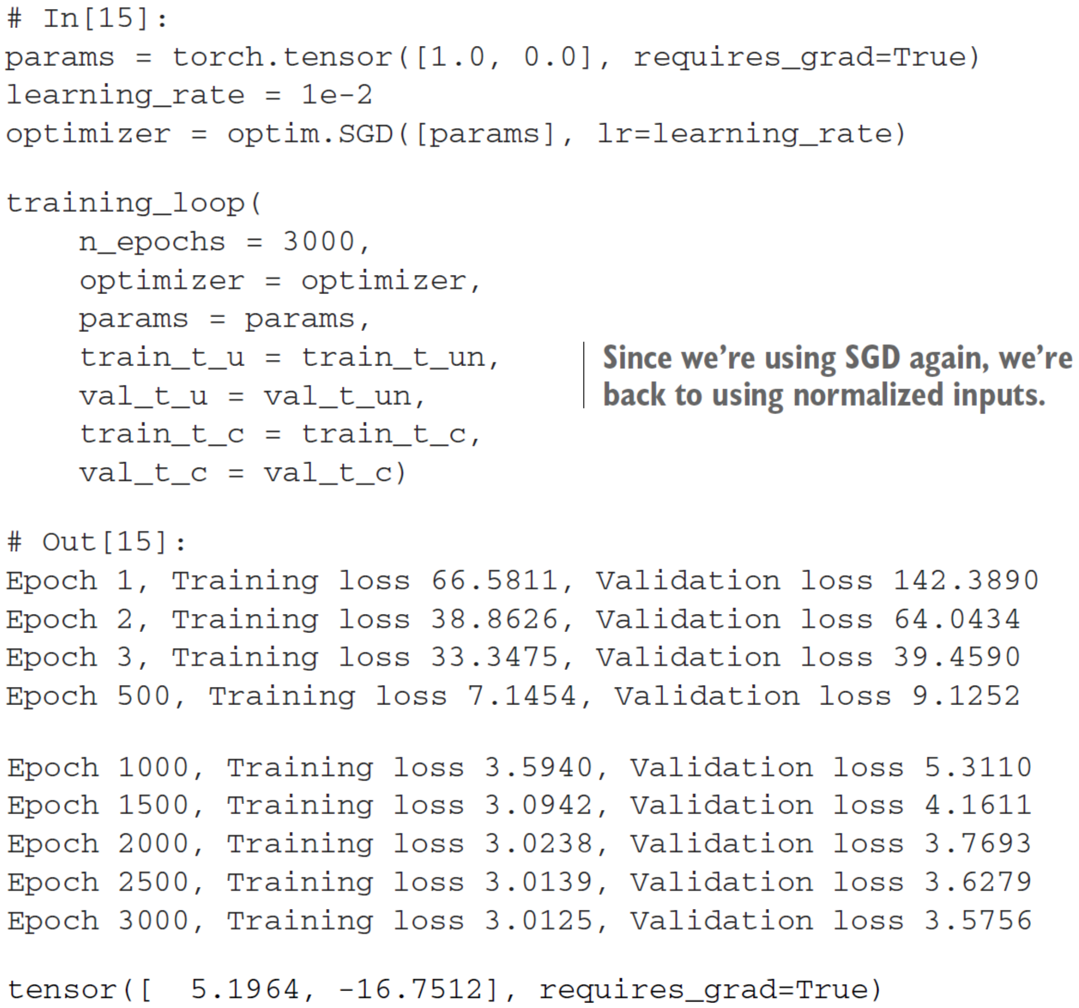
- The first line in the training loop evaluates `model` on `train_t_u` to produce `train_t_p`. Then `train_loss` is evaluated from `train_t_p`. This creates a computation graph that links `train_t_u` to `train_t_p` to `train_loss`.
- When `model` is evaluated again on `val_t_u`, it produces `val_t_p` and `val_loss`. In this case, a separate computation graph will be created that links `val_t_u` to `val_t_p` to `val_loss`.
- Separate tensors have been run through the same functions, `model` and `loss_fn`, generating separate computation graphs.

Modified Training Loop

- The only tensors these two graphs have in common are the parameters.
- When we call `backward` on `train_loss`, we run `backward` on the first graph.
- In other words, we accumulate the derivatives of `train_loss` with respect to the parameters based on the computation generated from `train_t_u`.
- If we (incorrectly) called `backward` on `val_loss` as well, we would accumulate the derivatives of `val_loss` with respect to the parameters *on the same leaf nodes*.


A
t utrain → model(x, params) → t ptrain
t uval → model(x, params) → t pval
B
t utrain → model(x, params) → t ptrain → loss train
t uval → model(x, params) → t pval → loss val
C
t utrain → model(x, params) ← → t ptrain ← → loss train .backward
.grad
t uval → model(x, params) → t pval → loss val
.grad

Modified Training Results

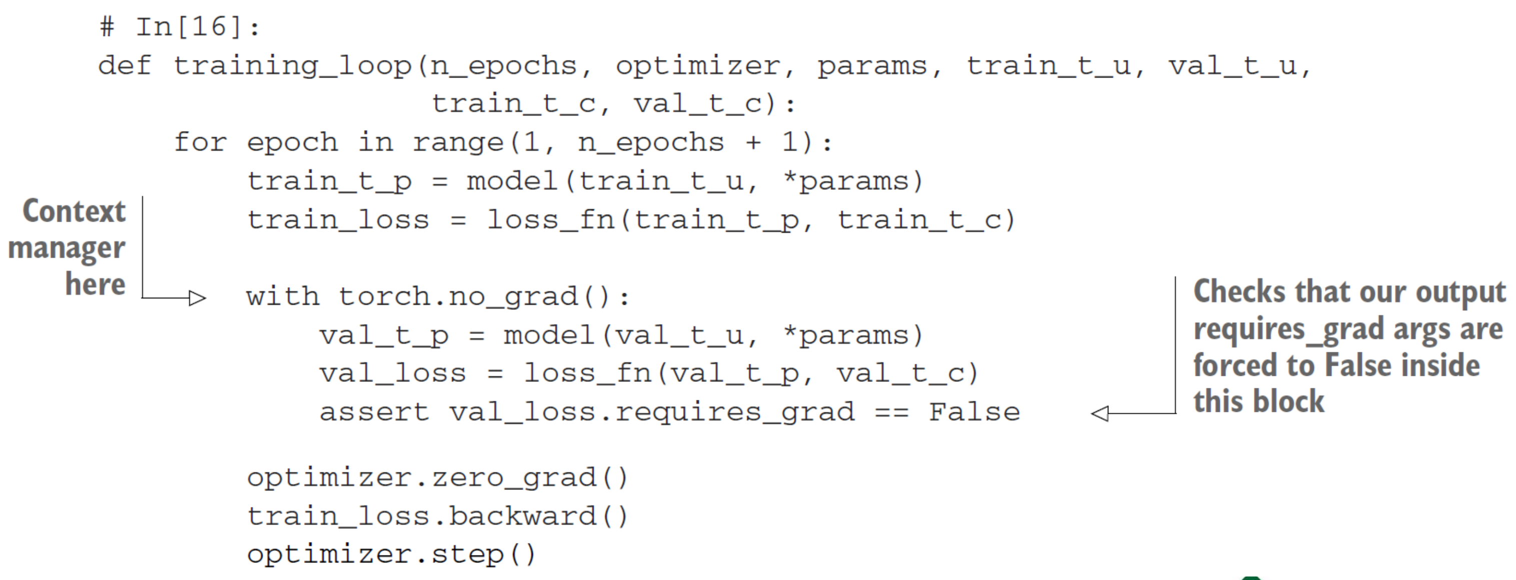


```
# In[15]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)

training_loop(
    n_epochs = 3000,
    optimizer = optimizer,
    params = params,
    train_t_u = train_t_un,
    val_t_u = val_t_un,
    train_t_c = train_t_c,
    val_t_c = val_t_c)

# Out[15]:
Epoch 1, Training loss 66.5811, Validation loss 142.3890
Epoch 2, Training loss 38.8626, Validation loss 64.0434
Epoch 3, Training loss 33.3475, Validation loss 39.4590
Epoch 500, Training loss 7.1454, Validation loss 9.1252

Epoch 1000, Training loss 3.5940, Validation loss 5.3110
Epoch 1500, Training loss 3.0942, Validation loss 4.1611
Epoch 2000, Training loss 3.0238, Validation loss 3.7693
Epoch 2500, Training loss 3.0139, Validation loss 3.6279
Epoch 3000, Training loss 3.0125, Validation loss 3.5756

tensor([  5.1964, -16.7512], requires_grad=True)
```

Since we're using SGD again, we're back to using normalized inputs.

Switching Off Autograd

- Since we're not ever calling `backward` on `val_loss`, why are we building the graph in the first place? We could in fact just call `model` and `loss_fn` as plain functions, without tracking the computation.
- In order to address this, PyTorch allows us to switch off autograd when we don't need it, using the `torch.no_grad` context manager.

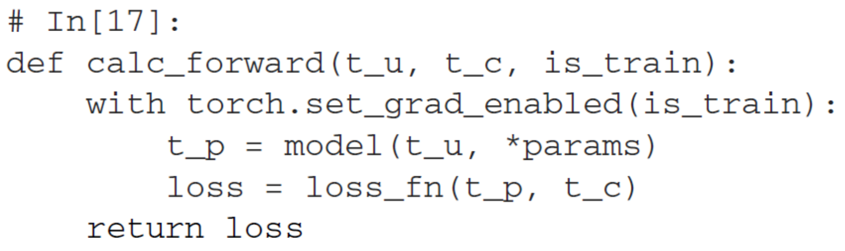


```
# In[16]:
def training_loop(n_epochs, optimizer, params, train_t_u, val_t_u,
                  train_t_c, val_t_c):
    for epoch in range(1, n_epochs + 1):
        train_t_p = model(train_t_u, *params)
        train_loss = loss_fn(train_t_p, train_t_c)

        with torch.no_grad():
            val_t_p = model(val_t_u, *params)
            val_loss = loss_fn(val_t_p, val_t_c)
            assert val_loss.requires_grad == False

        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()
```

Context manager here

Checks that our output requires_grad args are forced to False inside this block

Switching Off Autograd



```
# In[17]:
def calc_forward(t_u, t_c, is_train):
    with torch.set_grad_enabled(is_train):
        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)
    return loss
```

- For instance, define a `calc_forward` function that takes data as input and runs `model` and `loss_fn` with or without autograd according to a Boolean `is_train` argument.
- Using the related `set_grad_enabled` context, we can also condition the code to run
- With `autograd` enabled or disabled, according to a Boolean expression—typically indicating whether we are running in training or inference mode.