Deep Learning course - LAB 4

A tour of the optimizers in PyTorch

Recap from previous Lab

- We experimented with building a Multilayer Perceptron (MLP) trained on the MNIST dataset using *vanilla* Stochastic Gradient Descent (SGD) and constructing a training loop that lets us track loss and accuracy as training goes on

Agenda for today

- Today we will be taking a quick tour of the `torch.optim` library, having a look at some optimizers which are more advanced than vanilla SGD
- in addition to that, we will be exploring how to toggle the hyperparameters (chiefly, the learning rate) of the optimizer as the training is operated

In [ ]:
```python
import torch
from scripts.architectures import MLP # I have pasted the code for the MLP with regularization in this script, no nee
from scripts.train_utils import AverageMeter, accuracy
from scripts import mnist
```

Exploring optimizers in PyTorch

PT optimizers can be found in the `torch.optim` library.

We'll take a look at some of those, namely:

- SGD with momentum
- RMSProp
- Adam

If you're a fan of optimizers, you can yourself have a look at the plethora of optimizers in the `optim` library on the official docs.

SGD w/ momentum

Adding a momentum term helps SGD optimize faster in some situations where the optimum is situated in *valleys* which are way steeper along sime directions w.r.t. others.

Image from Deep Learning book (Goodfellow et al.) - chapter 8.3.2

The gradient is updated via the following quantity M:

$$\mathbf{M} \leftarrow \gamma \cdot \mathbf{M} + \eta \cdot \nabla \mathcal{L}$$

$$\mathbf{\Theta} \leftarrow \mathbf{\Theta} - \mathbf{M}$$

where $\nabla \mathcal{L}$ is the gradient and γ is the momentum term (usually picked $\to 1$).

Actually, SGD with momentum is part of vanilla SGD in PT. Indeed, one of its arguments is `momentum`.

In [ ]:
```python
lr = .1
wd = 5e-4
momentum = .9

num_epochs = 5

loss_fn = torch.nn.CrossEntropyLoss()

model = MLP()
optimizer = torch.optim.SGD(model.parameters(), lr=lr, weight_decay=wd, momentum=momentum)
```

Let us also recover the training and testing routines we defined during the last lab (without the trajectory):

In [ ]:
```python
def train_epoch(model, dataloader, loss_fn, optimizer, loss_meter, performance_meter, performance, device):
    for X, y in dataloader:
        X = X.to(device)
        y = y.to(device)
        # 1. reset the gradients previously accumulated by the optimizer
        #    this will avoid re-using gradients from previous loops
        optimizer.zero_grad()
        # 2. get the predictions from the current state of the model
        #    this is the forward pass
        y_hat = model(X)
        # 3. calculate the loss on the current mini-batch
        loss = loss_fn(y_hat, y)
        # 4. execute the backward pass given the current loss
        loss.backward()
        # 5. update the value of the params
        optimizer.step()
        # 6. calculate the accuracy for this mini-batch
        acc = performance(y_hat, y)
        # 7. update the loss and accuracy AverageMeter
        loss_meter.update(val=loss.item(), n=X.shape[0])
        performance_meter.update(val=acc, n=X.shape[0])


def train_model(model, dataloader, loss_fn, optimizer, num_epochs, checkpoint_loc=None, checkpoint_name="checkpoint.p

    # create the folder for the checkpoints (if it's not None)
    if checkpoint_loc is not None:
        os.makedirs(checkpoint_loc, exist_ok=True)

    model.train()

    # epoch loop
    for epoch in range(num_epochs):
        loss_meter = AverageMeter()
        performance_meter = AverageMeter()

        train_epoch(model, dataloader, loss_fn, optimizer, loss_meter, performance_meter, performance, device)

        print(f"Epoch {epoch+1} completed. Loss - total: {loss_meter.sum} - average: {loss_meter.avg}; Performance: {

        # produce checkpoint dictionary -- but only if the name and folder of the checkpoint are not None
        if checkpoint_name is not None and checkpoint_loc is not None:
            checkpoint_dict = {
                "parameters": model.state_dict(),
                "optimizer": optimizer.state_dict(),
                "epoch": epoch
            }
            torch.save(checkpoint_dict, os.path.join(checkpoint_loc, checkpoint_name))

    return loss_meter.sum, performance_meter.avg

def test_model(model, dataloader, performance=accuracy, loss_fn=None, device="cuda"):
    # create an AverageMeter for the loss if passed
    if loss_fn is not None:
        loss_meter = AverageMeter()

    performance_meter = AverageMeter()

    model.eval()
    with torch.no_grad():
        for X, y in dataloader:
            X = X.to(device)
            y = y.to(device)
            y_hat = model(X)
            loss = loss_fn(y_hat, y) if loss_fn is not None else None
            acc = performance(y_hat, y)
            if loss_fn is not None:
                loss_meter.update(loss.item(), X.shape[0])
            performance_meter.update(acc, X.shape[0])
    # get final performances
    fin_loss = loss_meter.sum if loss_fn is not None else None
    fin_perf = performance_meter.avg
    print(f"TESTING - loss {fin_loss if fin_loss is not None else '--'} - performance {fin_perf}")
    return fin_loss, fin_perf
```

and recover the data

In [ ]:
```
trainloader, testloader = mnist.get_data()
```

Let's train the network

In [ ]:
```
train_model(model, trainloader, loss_fn, optimizer, num_epochs)
```

by adding the momentum term, we already saw a small increase in training accuracy. Let's test

In [ ]:
```
test_model(model, testloader)
```

RMSProp and the LR sensitivity "dilemma"

With RMSProp we want to tackle a problem with SGD/SGD+momentum, which is related to the fact that, with SGD, there seems to be a deal of *sensitivity* towards some specific *directions* (read, parameters, since each parameter of the model represent a dimension in the optimization space).

RMSProp tries to tackle this issue by introducing an *adaptive rule* for updating the learning rate parameter-wise in each step. In particular:

- it keeps track of the *history* of the squared gradient (second moment estimation) via an exponentially decaying running average: $\mathbf{V} \leftarrow \alpha \mathbf{V} + (1-\alpha)\nabla\mathcal{L}^2$
  - (the *decay* is controlled by a hyperparameter $\alpha \in (0, 1)$, usually 0.9)
- The parameter update is

- directly proportional to the learning rate
- directly proportional to the gradient for this step
- inversely proportional to the gradient average
    - i.e., the direct effect of the gradient is *mitigated* by dividing it with the accumulated average gradient

The formula for the update is:

$$\Theta \leftarrow \Theta + \frac{\eta}{\epsilon + \sqrt{\mathbf{V}}} \odot \nabla\mathcal{L}$$

where:

- ϵ is a small constant for numerical stability
- $\mathbf{V}$ is the squared gradient running averate (which depends upon ρ)
- $\nabla\mathbf{L}$ is the gradient for the current step
- $\odot$ indicates the Hadamard matrix product (el.-by-el. product)

In [ ]:
```
model = MLP() # always remember to reinstantiate the net between tries
rmsprop = torch.optim.RMSprop(model.parameters()) # let's use the default hyperparams (lr=.01, eps=1e-8)
```

In [ ]:
```
train_model(model, trainloader, loss_fn, rmsprop, num_epochs)
test_model(model, testloader)
```

ADAM

Adam is an extension of RMSProp where we try to implement momentum-like mechanics as well.

Instead of adding one single momentum term, we add two of them:

$$\mathbf{M} \leftarrow (\beta_1\mathbf{M} + (1-\beta_1)\nabla\mathcal{L}) \ / \ (1-\beta_1^t)$$

$$\mathbf{V} \leftarrow (\beta_2\mathbf{V} + (1-\beta_2)\nabla\mathcal{L}^2) \ / \ (1-\beta_2^t)$$

where t is the training iteration.

The two terms are running averages (with a so called *bias correction* at the denominator) of the gradient and its square respectively.

These terms are then incorporated into the parameters update formula:

$$\mathbf{\Theta} \leftarrow \mathbf{\Theta} + \text{lr}\frac{\mathbf{M}}{\sqrt{\mathbf{V}}+\epsilon} \cdot \nabla\mathcal{L}$$

Notice the similarities between Adam and RMSProp.

In [ ]:
```
model = MLP()
adam = torch.optim.Adam(model.parameters()) # we keep the default hyperparameters
```

In [ ]:
```
train_model(model, trainloader, loss_fn, adam, num_epochs)
test_model(model, testloader)
```

The literature is loaded with SGD variants for optimization: Adagrad, AdaMax, Nadam, AdamW... You can use one of them of your own choice in your exercises, (provided you can explain the concept behind it during the exam).

Additional reading

If you're interested in SGD variants, you may check out this blog post which, in my opinion, does a good job in summarising and presenting recent work in the field.

Modifying the optimizer hyperparameters

One thing we might be interested in doing is to modify the hyperparameters of our optimizer mid-training.

The parameters of the optimizer are contained:

- in its `state_dict`
- under the `param_groups`

we will see how to work with the latter.

In [ ]:
```
print(type(optimizer.param_groups))
print(len(optimizer.param_groups))
```

The `param_groups` represent groups of parameters for which given conditions apply.

Here we have only one group, corresponding to the params of our MLP network.

Let us see how this group is composed:

In [ ]:
```
print(type(optimizer.param_groups[0]))
print(optimizer.param_groups[0].keys())
```

To the surprise of no-one, the parameters of the MLP are stored under the `params` key.

The other keys represent the *conditions* that apply to these parameter group.

Toggling one of these hyperparameters can be done in that way.

In [ ]:
```
optimizer.param_groups[0]["momentum"] = 0.8 # -> from now on, the momentum will be decreased a little bit
```

even if it's better to be general: if we're willing to do a global update for that optimizer, we better do it for all groups.

In [ ]:
```
for pg in optimizer.param_groups:
    pg["momentum"] = .8
```

Let us suppose we wish to use a different momentum or learning rate for each layer

In [ ]:
```
optimizer_diff = torch.optim.SGD(
    [
        {"params": model.layers[:6].parameters()},
        {"params": model.layers[6:].parameters()}
    ],
    lr=.1, weight_decay=5e-4, momentum=.9
)
```

We have split the params of our MLP in two groups (the first 6 layers and the remaining ones). Let's check that we have >1 `param_group`

In [ ]:
```
len(optimizer_diff.param_groups)
```

Now, suppose we might want to have a different weight decay in the second group: we only need to toggle it.

In [ ]:
```
optimizer_diff.param_groups[1]["weight_decay"] = 1e-3
```

In [ ]:
```
_ = [[print(hyp, "\t", val) for hyp, val in pg.items() if hyp!="params"] for pg in optimizer_diff.param_groups]
```

We also could've done this like so:

In [ ]:
```
optimizer_diff = torch.optim.SGD(
    [
        {"params": model.layers[:6].parameters(), "weight_decay": 5e-4},
        {"params": model.layers[6:].parameters(), "weight_decay": 1e-3}
    ],
     momentum=.9, lr=.1
)

_ = [[print(hyp, "\t", val) for hyp, val in pg.items() if hyp!="params"] for pg in optimizer_diff.param_groups]
```

The Learning Rate dilemma in Deep Learning

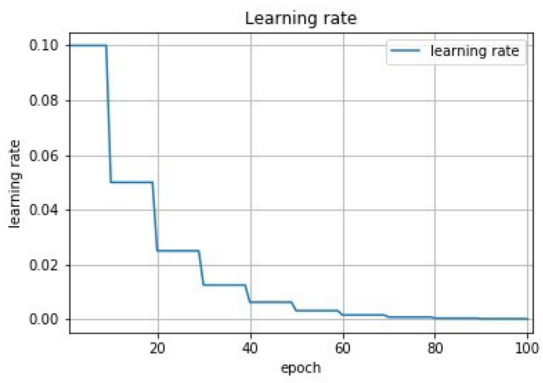
static nature of the learning rate (LR):

- if the LR is too high, we'll notice a sharp increase in accuracy with a relatively quick plateu corresponding to non-optimal solutions.
  - this is because we'll likely miss local optima because our step in the parameter space is too large
- if the LR is too low, training will be excruciatingly low and we'll likely get stuck in very bad local optima, being unable to get out of them because the step in the parameter space is too low to get out of these *valleys*

An *ideal* solution would be to keep a *high enough* LR until we find a *good enough* portion of the parameter space, then decrease progressively the LR in order to carefully explore these areas for good optima.

Mid-training learning rate toggling is called in a variety of terms: **learning rate decay**, **learning rate annealing**, **learning rate scheduling**...

The simplest idea to implement this is a **stepwise** learning rate annealing:



picture from towardsdatascience.com.

In our MLP trained with SGD + momentum, we wish to train for 15 epochs and decrease the lr by a factor of 1/10 **before** epoch 7 and 12.

Let us recover our training loop and update it accordingly:

In [ ]:
```
def train_model(model, dataloader, loss_fn, optimizer, num_epochs, checkpoint_loc=None, checkpoint_name="checkpoint.p

    # create the folder for the checkpoints (if it's not None)
    if checkpoint_loc is not None:
        os.makedirs(checkpoint_loc, exist_ok=True)

    model.train()

    # epoch loop
    for epoch in range(num_epochs):
        if epoch in (6, 11):
            for pg in optimizer.param_groups:
                pg["lr"] *= .1

        loss_meter = AverageMeter()
        performance_meter = AverageMeter()

        train_epoch(model, dataloader, loss_fn, optimizer, loss_meter, performance_meter, performance)

        print(f"Epoch {epoch+1} completed. Loss - total: {loss_meter.sum} - average: {loss_meter.avg}; Performance: {

        # produce checkpoint dictionary -- but only if the name and folder of the checkpoint are not None
        if checkpoint_name is not None and checkpoint_loc is not None:
            checkpoint_dict = {
                "parameters": model.state_dict(),
                "optimizer": optimizer.state_dict(),
                "epoch": epoch
            }
            torch.save(checkpoint_dict, os.path.join(checkpoint_loc, checkpoint_name))

    return loss_meter.sum, performance_meter.avg
```

PyTorch has a tool additional to the optimizer, the `lr_scheduler`.

The closest thing to the one above is the **StepLR**, which decays the lr by `gamma` each `step_size` epochs.

In [ ]:
```
model = MLP()
```

In [ ]:
```
optimizer = torch.optim.SGD(model.parameters(), lr=.1, weight_decay=5e-4, momentum=.9)

scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[6, 11], gamma=.1)
```

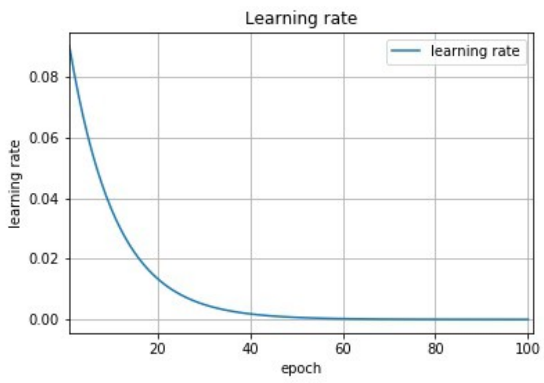
In [ ]:
```
def train_model(model, dataloader, loss_fn, optimizer, num_epochs, checkpoint_loc=None, checkpoint_name="checkpoint.p
    # added lr_scheduler

    # create the folder for the checkpoints (if it's not None)
    if checkpoint_loc is not None:
        os.makedirs(checkpoint_loc, exist_ok=True)

    model.train()

    # epoch loop
    for epoch in range(num_epochs):

        loss_meter = AverageMeter()
        performance_meter = AverageMeter()

        # added print for LR
        print(f"Epoch {epoch+1} --- learning rate {optimizer.param_groups[0]['lr']:.5f}")

        train_epoch(model, dataloader, loss_fn, optimizer, loss_meter, performance_meter, performance)

        print(f"Epoch {epoch+1} completed. Loss - total: {loss_meter.sum} - average: {loss_meter.avg}; Performance: {

        # produce checkpoint dictionary -- but only if the name and folder of the checkpoint are not None
        if checkpoint_name is not None and checkpoint_loc is not None:
            checkpoint_dict = {
                "parameters": model.state_dict(),
                "optimizer": optimizer.state_dict(),
                "epoch": epoch
            }
            torch.save(checkpoint_dict, os.path.join(checkpoint_loc, checkpoint_name))

        if lr_scheduler is not None:
            lr_scheduler.step()

    return loss_meter.sum, performance_meter.avg
```

In [ ]:
```
train_model(model, trainloader, loss_fn, optimizer, 15, lr_scheduler=scheduler)
```

In [ ]:
```
test_model(model, testloader)
```

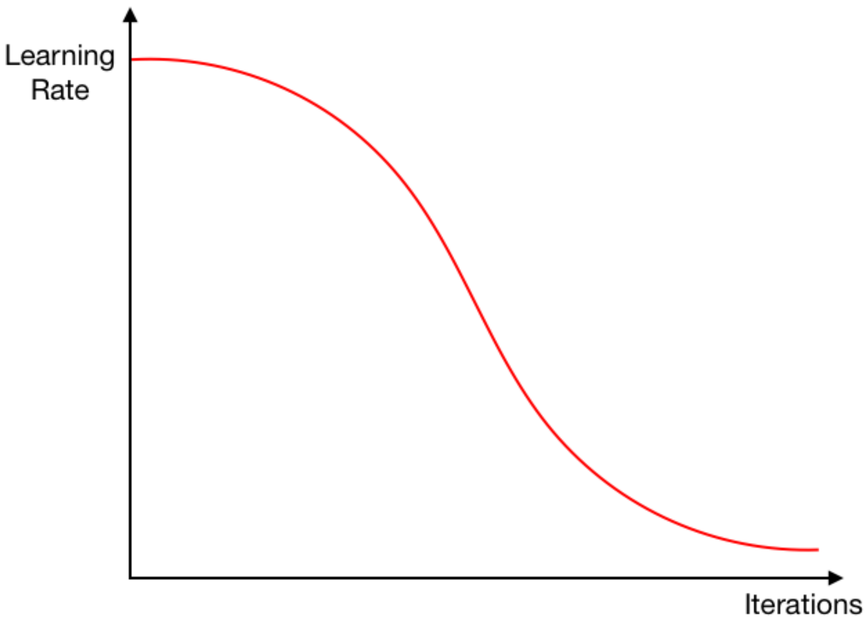
Other techniques

1. Exponential Annealing



picture from towardsdatascience.com.

1. Cosine Annealing



picture from towardsdatascience.com.

1. Triangular Annealing

picture from Universal Language Model Fine-tuning for Text Classification.

Warm-up

Warm-up is a techinque which is centered on the idea that, before we start the actual training, the network has to be *warmed-up* with some iterations of training at an ever-increasing LR, till we hit the target LR η.

A simple implementation of this (which resembles the ascending phase of the triangular schedule above) could be to:

- warm up for U iterations
- increase the LR by a fraction $\frac{\eta}{U}$.

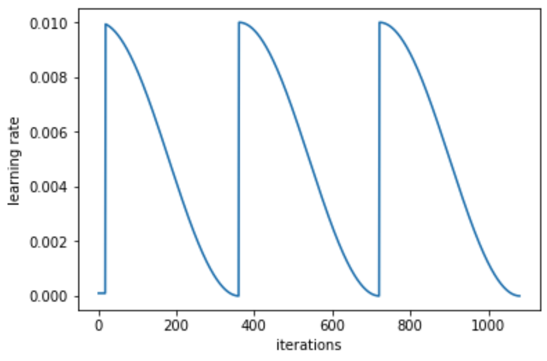
So, at iteration $u \in \{1, \dots, U\}$, the LR is $u\frac{\eta}{U}$.

Hence, the triangular annealing above could be thought of as a composition of

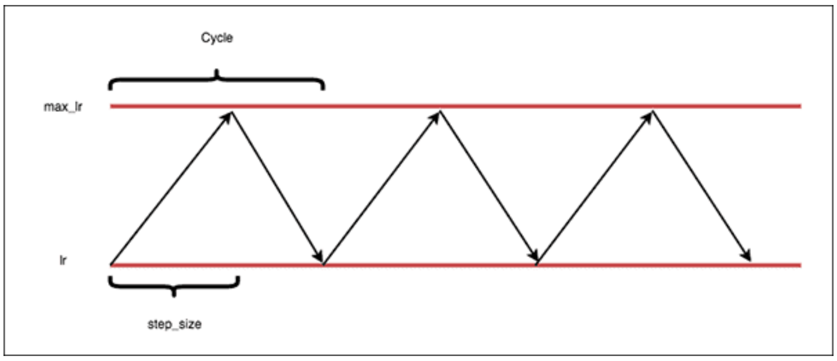
1. Linear warm-up, and
2. Linear annealing

LR schedule cycling

The aforementioned schedules can be cycled multiple times during the same training, giving rise to shapes like the following:



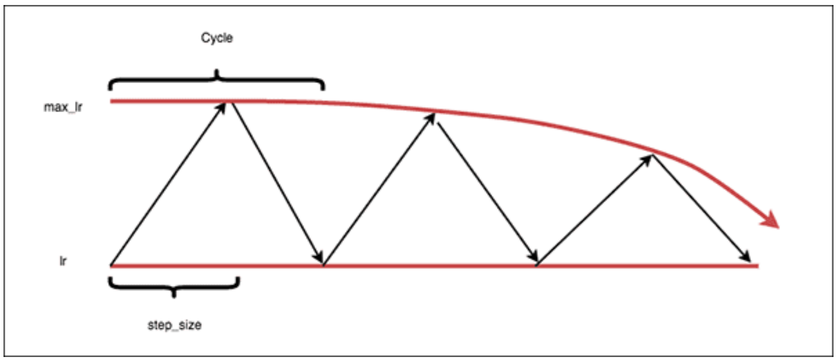
picture from towardsdatascience.com



picture from pyimagesearch.com

This, coupled with E_{opt} early stopping, might actually give the optimizer multiple end points from which to choose our best model. Each time the LR gets "bumped up", we get a "fresh restart" from a possibly more favorable initialization, in the hope of getting closer and closer to a good local optimum.

On to something a bit more complex:



picture from pyimagesearch.com

the maximum LR gets decayed as well in a "logarithmic" way. We can have a similar figure with the cosine annealing as well.

Further watch (a bit older, from 2018): 2

Further read, an argument proposing an alternative to LR annealing: Don't Decay the Learning Rate, Increase the Batch Size. The batch size is not solely related to the available memory for training: a batch size too high will often result in the optimizer getting stuck in bad areas -- indeed, often networks trained with full-batch gradient descent act badly. The **batch size is then another regularizer** and finding its optimal value is dataset- and architecture-dependent. According to the paper above, increasing the batch size as the network approaches a good solution might be a good idea -- I won't take it for granted though, as a lot of research, being based upon experimental proofs, often presents results specific to a given dataset or architecture but depicting them as "general"; moreover, *clickbait titles* are more and more present even for research papers that should instead be based upon humble claims and careful enucleation of limitations.

Defining a custom optimizer

To define a custom optimizer, we essentially need to build a class inheriting from `torch.optim.Optimizer` and having the following methods:

- `__init__`
- `step`

Let's see an example below:

In [ ]:
```python
class SGD_Alternate(torch.optim.Optimizer): #torch module as a class of torch.optim.Optimizer
    '''
    An alternative of torch.optim.SGD with momentum and no weight decay
    '''
    def __init__(self, parameters, lr:float, momentum:float):
        # we skip check on the args values
        # initialize superclass with
            # parameters
            # hyperparameters passed as dictionary at the second position
        super().__init__(
            parameters,
            {
                "lr": lr,
                "momentum": momentum
            }
        )
    # In the training I call optimizer.steo which updates the parameters
    @torch.no_grad() # -> decorator specifying that all the computations here are done without keeping the gradient
    def step(self):
        for pg in self.param_groups: # note: param_groups is already here bc it's defined in the father class
            for param in pg["params"]:
                if param.grad is not None: # param update operated only on those params having gradient
                    # If there is not gradient:
                    # 1) I forgot to call backward
                    # 2) you fix the param not to be trained with .grad = False like for m_T / sigma_T in slides (onl

                    # now we wish to recover the momentum buffer (which in the equation of SGD with momentum is indic
                    # each parameter within the optimizer group indexes a dictionary where the values are a group of
                    # which are crucial for the optimizer to work
                    state = self.state[param]
                    # we use .get so it does not throw an exception in case of nonexistence of the key in the dict
                    M = state.get("momentum_buffer")
                    # momentum buffer: iteratively evaluate m <- gamma*m + eta*grad(L)

                    if M is not None:
                        # if the buffer exists -> update it
                        # M = self.momentum * M + self.lr * param.grad
                        # ↓ more advanced notion than ↑
                        M.mul_(pg["momentum"]).add_(param.grad * pg["lr"])
                        # REMEMBER: _ means an IN-PLACE operation
                    else:
                        # if the buffer does not exist, we must calculate the momentum and then assign it to the dict
                        # we don't use pg["momentum"] as we don't have previous values for it.
                        M = pg["lr"] * param.grad # the first time m is 0 so : m <- 0 + eta*grad(L)
                        state["momentum_buffer"] = M
                    # now we can update the params by subtracting the buffer M
                    param.sub_(M) # Substitution IN-PLACE of m
```

Let us verify that it works by training on a few epochs

In [ ]:
```python
net = MLP()
optim_alt = SGD_Alternate(net.parameters(), lr=.1, momentum=.9)
train_model(net, trainloader, loss_fn, optim_alt, 3)
```

References for this chapter

Writing Your Own Optimizers in PyTorch, by Daniel McNeela

Official PT source for SGD

References

1 LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. nature, 521(7553), 436-444.

2 State-of-the-art Learning Rate Schedules. Apache MXNet. YouTube.

3 Zhang, C., Bengio, S., Hardt, M., Recht, B., & Vinyals, O. (2016). Understanding deep learning requires rethinking generalization.