# fast.ai

## ▼ Chapter.4 부분 정리 (p.171~184)

### Putting it All Together

이번 쳅터에서는 바로 Fig.4.1의 도식도를 직접 구현해보는 것을 목표로 한다.

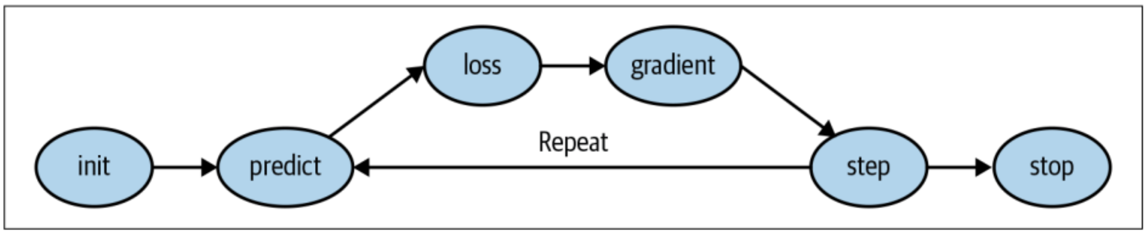
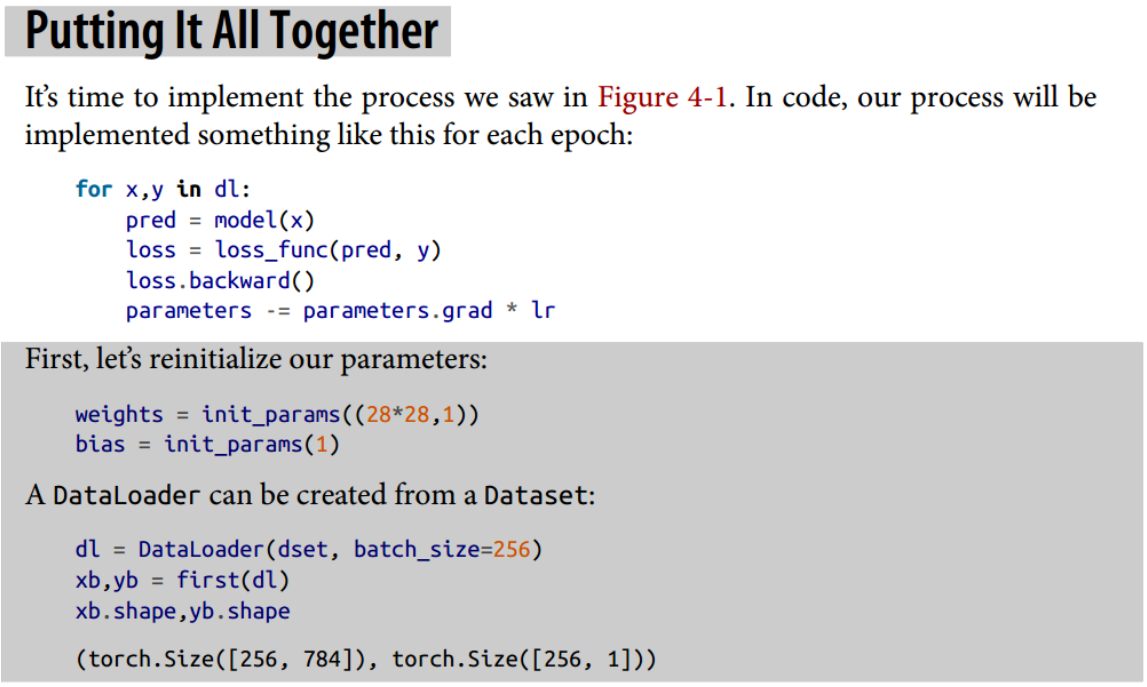


*Figure 4-1. The gradient descent process*

아래는 그 시작으로, weight와 dataloader의 구성 부분입니다.



**Putting It All Together**

It's time to implement the process we saw in Figure 4-1. In code, our process will be implemented something like this for each epoch:

```
for x,y in dl:
    pred = model(x)
    loss = loss_func(pred, y)
    loss.backward()
    parameters -= parameters.grad * lr
```

First, let's reinitialize our parameters:

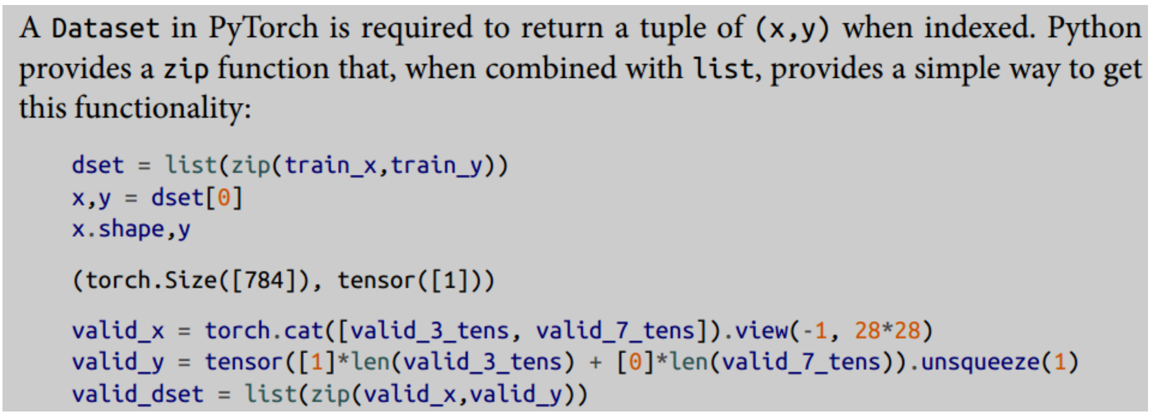
```
weights = init_params((28*28,1))
bias = init_params(1)
```

A `DataLoader` can be created from a `Dataset`:

```
dl = DataLoader(dset, batch_size=256)
xb,yb = first(dl)
xb.shape,yb.shape

(torch.Size([256, 784]), torch.Size([256, 1]))
```

위의 **dset** 부분



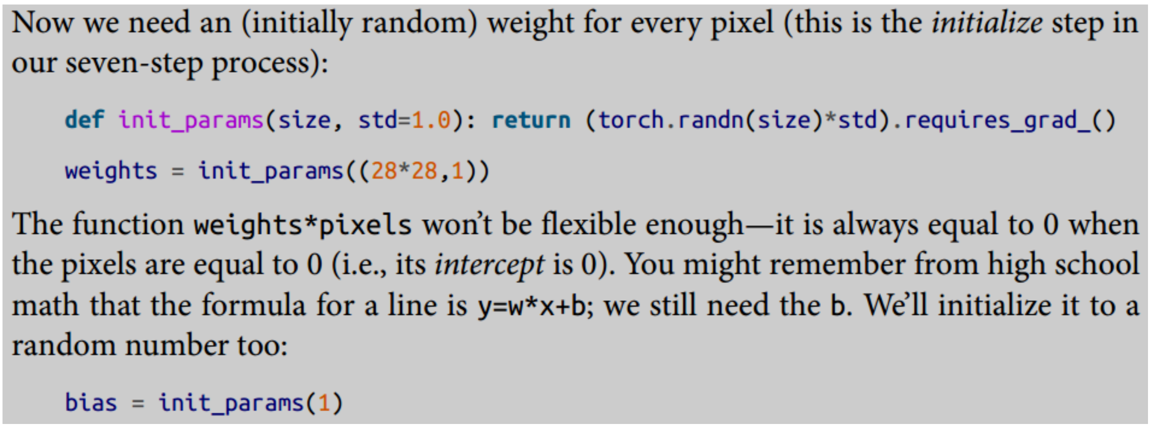
A `Dataset` in PyTorch is required to return a tuple of `(x,y)` when indexed. Python provides a `zip` function that, when combined with `list`, provides a simple way to get this functionality:

```
dset = list(zip(train_x,train_y))
x,y = dset[0]
x.shape,y
```

```
(torch.Size([784]), tensor([1]))
```

```
valid_x = torch.cat([valid_3_tens, valid_7_tens]).view(-1, 28*28)
valid_y = tensor([1]*len(valid_3_tens) + [0]*len(valid_7_tens)).unsqueeze(1)
valid_dset = list(zip(valid_x,valid_y))
```

위의 **init_params 함수** 부분



Now we need an (initially random) weight for every pixel (this is the *initialize* step in our seven-step process):

```
def init_params(size, std=1.0): return (torch.randn(size)*std).requires_grad_()
```

```
weights = init_params((28*28,1))
```

The function `weights*pixels` won't be flexible enough—it is always equal to 0 when the pixels are equal to 0 (i.e., its *intercept* is 0). You might remember from high school math that the formula for a line is `y=w*x+b`; we still need the `b`. We'll initialize it to a random number too:

```
bias = init_params(1)
```

Mini-Batch로 테스트를 하고, loss에 대한 gradient를 구해보자. 그리고 이를 calc_grad로 함수화 시켜보자.

Let's create a mini-batch of size 4 for testing:

```
batch = train_x[:4]
batch.shape
```

```
torch.Size([4, 784])
```

```
preds = linear1(batch)
preds
```

```
tensor([[-11.1002],
        [  5.9263],
        [  9.9627],
        [ -8.1484]], grad_fn=<AddBackward0>)
```

```
loss = mnist_loss(preds, train_y[:4])
loss
```

```
tensor(0.5006, grad_fn=<MeanBackward0>)
```

Now we can calculate the gradients:

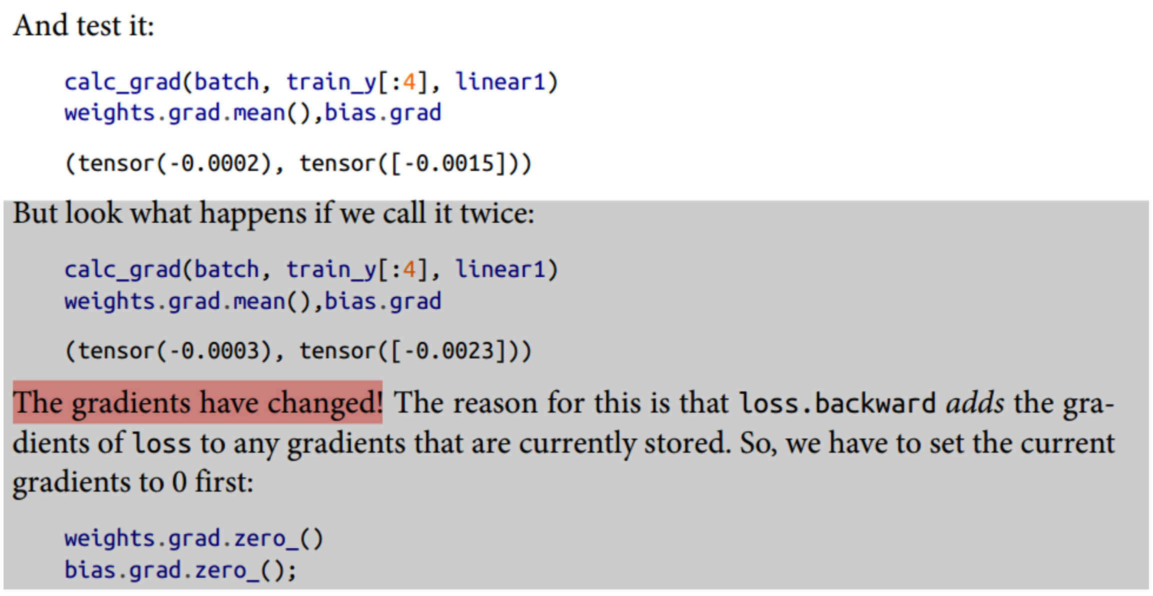
```
loss.backward()
weights.grad.shape,weights.grad.mean(),bias.grad
```

```
(torch.Size([784, 1]), tensor(-0.0001), tensor([-0.0008]))
```

Let's put that all in a function:

```
def calc_grad(xb, yb, model):
    preds = model(xb)
    loss = mnist_loss(preds, yb)
    loss.backward()
```

여기서는 함수 테스트를 하면서, 동시에 backward를 2번 했을 시에 어떤 일이 발생하는지 알려주는 중요한 점을 지목한다.


And test it:

```
calc_grad(batch, train_y[:4], linear1)
weights.grad.mean(),bias.grad
```

```
(tensor(-0.0002), tensor([-0.0015]))
```

But look what happens if we call it twice:

```
calc_grad(batch, train_y[:4], linear1)
weights.grad.mean(),bias.grad
```

```
(tensor(-0.0003), tensor([-0.0023]))
```

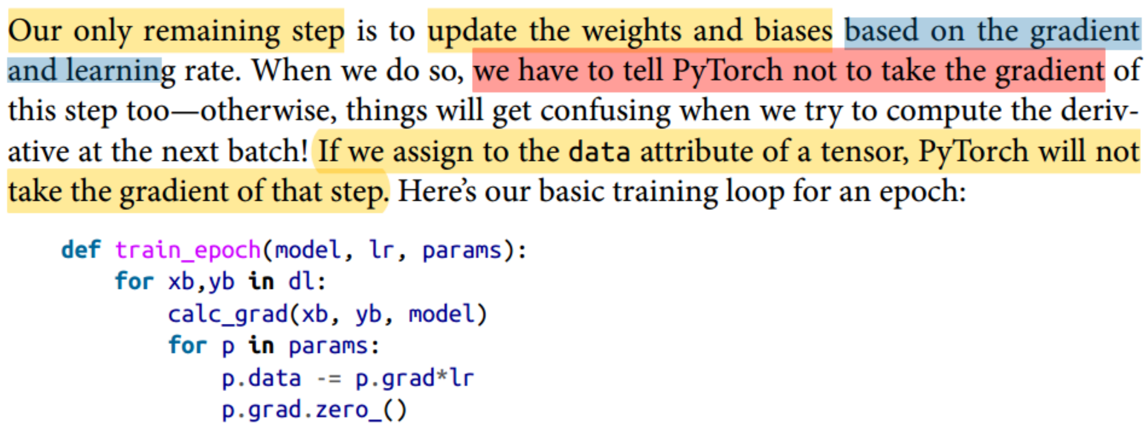
The gradients have changed! The reason for this is that `loss.backward` *adds* the gradients of `loss` to any gradients that are currently stored. So, we have to set the current gradients to 0 first:

```
weights.grad.zero_()
bias.grad.zero_();
```

바로, gradient가 backward를 할 때마다, gradient의 결과값을 계속 저장한다는 점이다.

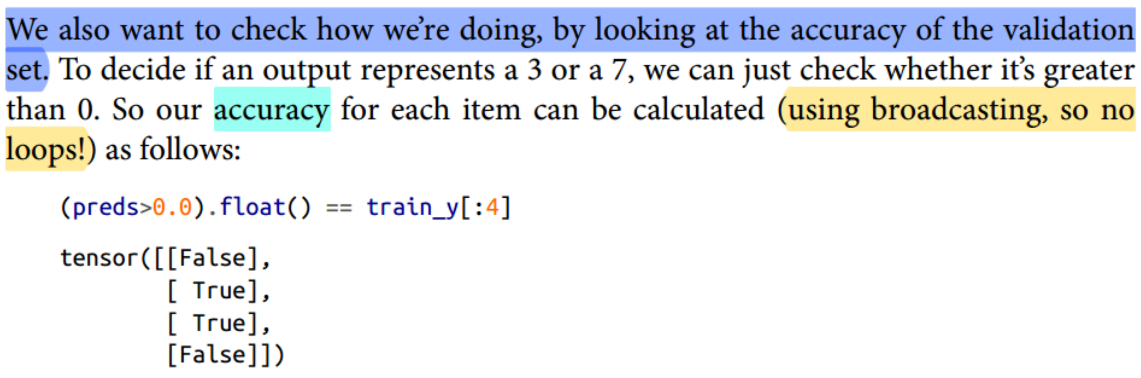
cf. 뒤에 언더바가 하나인 메소드는 기본적으로 어떤 object를 in-place로 변형시킬 때 쓴다.

이제 위의 gradient 값에 learning rate을 곱해줘서 parameter update하는 학습 함수를 만들어보자.


Our only remaining step is to update the weights and biases based on the gradient and learning rate. When we do so, we have to tell PyTorch not to take the gradient of this step too—otherwise, things will get confusing when we try to compute the derivative at the next batch! If we assign to the `data` attribute of a tensor, PyTorch will not take the gradient of that step. Here's our basic training loop for an epoch:

```
def train_epoch(model, lr, params):
    for xb,yb in dl:
        calc_grad(xb, yb, model)
        for p in params:
            p.data -= p.grad*lr
            p.grad.zero_()
```

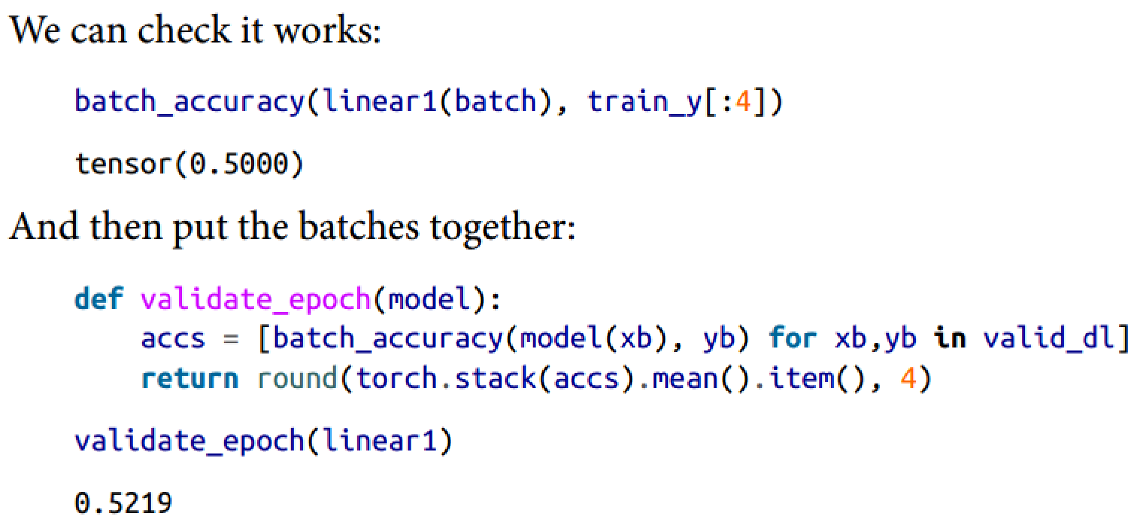
validation accuracy를 다음과 같이 check할 수 있다. 책은 친절하게도 python의 broadcasting을 쓴다는 것도 알려준다.


We also want to check how we're doing, by looking at the accuracy of the validation set. To decide if an output represents a 3 or a 7, we can just check whether it's greater than 0. So our accuracy for each item can be calculated (using broadcasting, so no loops!) as follows:

```
(preds>0.0).float() == train_y[:4]
```

```
tensor([[False],
        [ True],
        [ True],
        [False]])
```

추가적으로 다음과 같이 validate_epoch이라는 함수를 만들어서, model을 줬을 때, 그 model의 성능 평가를 validation dataloader로 평가한다음, acc를 return해주도록 하였다.


We can check it works:

```
batch_accuracy(linear1(batch), train_y[:4])
```

```
tensor(0.5000)
```

And then put the batches together:

```
def validate_epoch(model):
    accs = [batch_accuracy(model(xb), yb) for xb,yb in valid_dl]
    return round(torch.stack(accs).mean().item(), 4)
```

```
validate_epoch(linear1)
```

```
0.5219
```

최종적으로 다음과 같이 학습과 평가를 하는 training loop을 만들어 볼 수 있다.

That's our starting point. Let's train for one epoch and see if the accuracy improves:

```
lr = 1.
params = weights,bias
train_epoch(linear1, lr, params)
validate_epoch(linear1)
```

```
0.6883
```

Then do a few more:

```
for i in range(20):
    train_epoch(linear1, lr, params)
    print(validate_epoch(linear1), end=' ')
```

```
0.8314 0.9017 0.9227 0.9349 0.9438 0.9501 0.9535 0.9564 0.9594 0.9618 0.9613
 > 0.9638 0.9643 0.9652 0.9662 0.9677 0.9687 0.9691 0.9691 0.9696
```

### Creating an Optimizer

이제부터는 우리가 원하는 optimizer를 작성해보자.

일단 pytorch 는 어떤 모델의 parameter를 parameters() 메소드 호출로 불러올 수 있다.

Because this is such a general foundation, PyTorch provides some useful classes to make it easier to implement. The first thing we can do is replace our `linear` function with PyTorch's `nn.Linear` module. A *module* is an object of a class that inherits from the PyTorch `nn.Module` class. Objects of this class behave identically to standard Python functions, in that you can call them using parentheses, and they will return the activations of a model.

`nn.Linear` does the same thing as our `init_params` and `linear` together. It contains both the *weights* and *biases* in a single class. Here's how we replicate our model from the previous section:

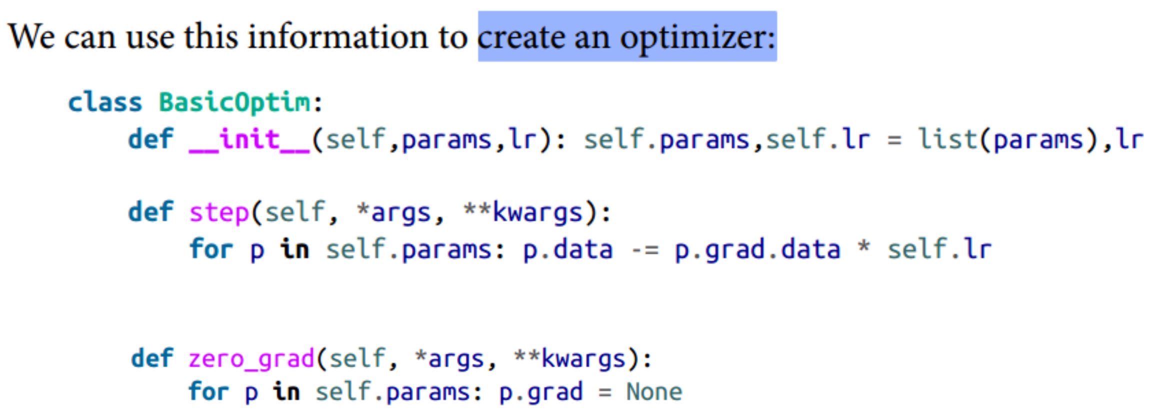
```
linear_model = nn.Linear(28*28,1)
```

Every PyTorch module knows what parameters it has that can be trained; they are available through the `parameters` method:

```
w,b = linear_model.parameters()
w.shape,b.shape
```

```
(torch.Size([1, 784]), torch.Size([1]))
```

그렇다면 parameter()를 이용하여 optimizer를 작성해보자.



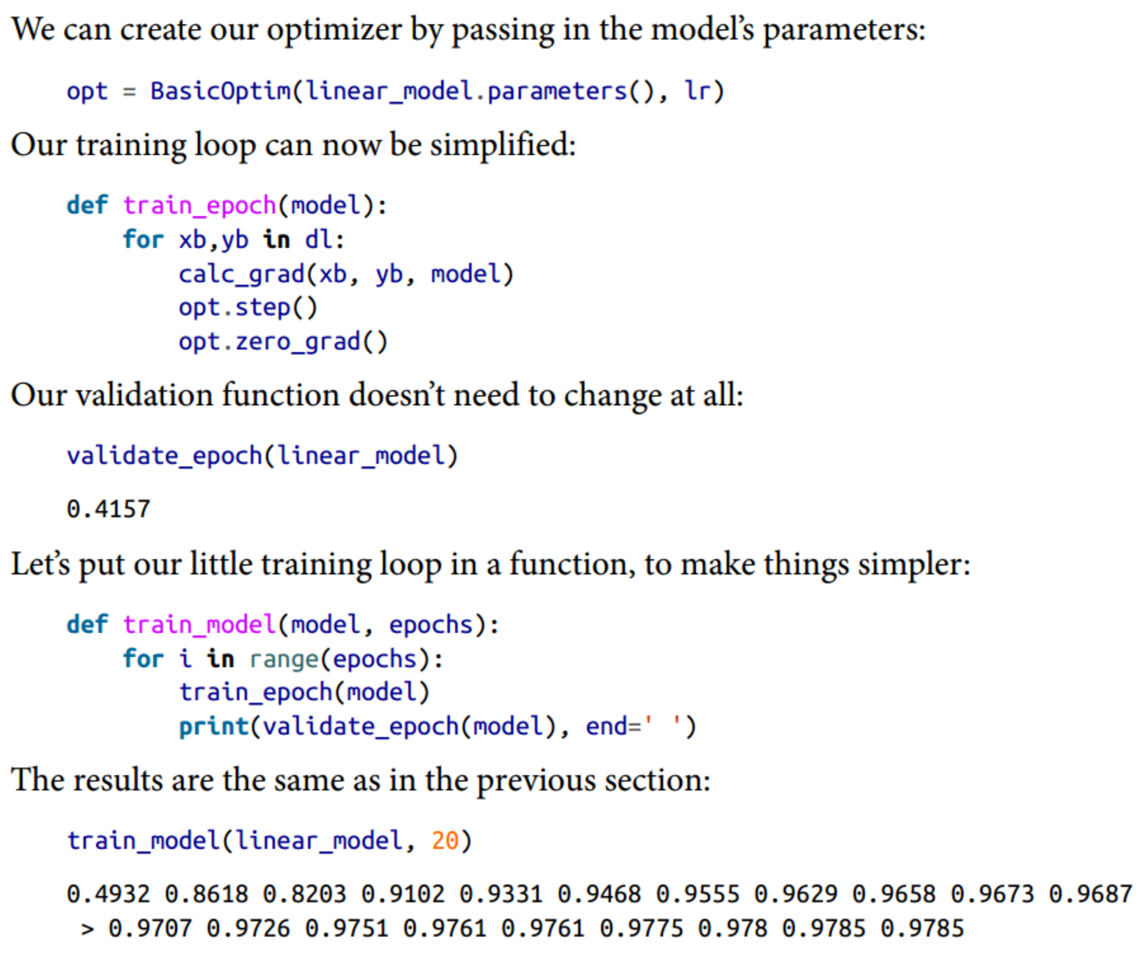
We can use this information to create an optimizer:

```
class BasicOptim:
    def __init__(self,params,lr): self.params,self.lr = list(params),lr

    def step(self, *args, **kwargs):
        for p in self.params: p.data -= p.grad.data * self.lr


    def zero_grad(self, *args, **kwargs):
        for p in self.params: p.grad = None
```

이렇게 클래스로 작성된 optimizer는 gradient의 대상이 되는 parametert를 인자로 받아서, 새로운 인스턴스를 생성해낸다. (lr은 learning rate이다)



We can create our optimizer by passing in the model's parameters:

```
opt = BasicOptim(linear_model.parameters(), lr)
```

Our training loop can now be simplified:

```
def train_epoch(model):
    for xb,yb in dl:
        calc_grad(xb, yb, model)
        opt.step()
        opt.zero_grad()
```

Our validation function doesn't need to change at all:

```
validate_epoch(linear_model)
```

```
0.4157
```

Let's put our little training loop in a function, to make things simpler:

```
def train_model(model, epochs):
    for i in range(epochs):
        train_epoch(model)
        print(validate_epoch(model), end=' ')
```

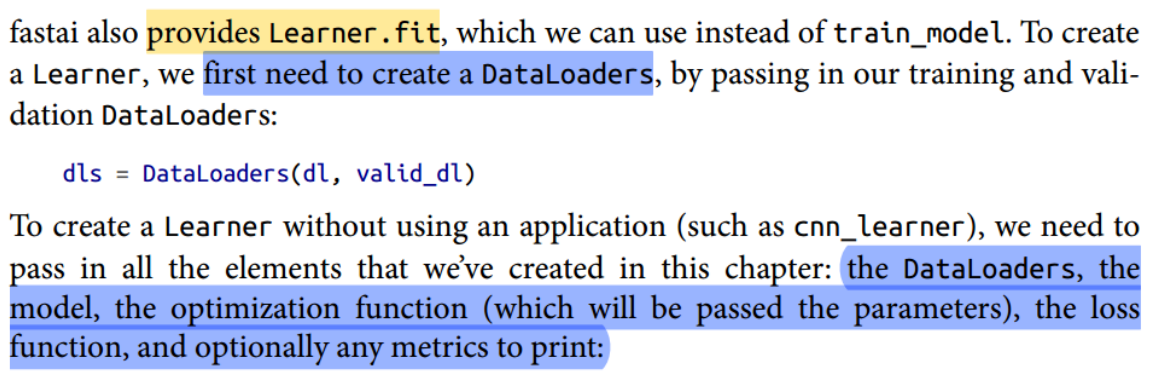
The results are the same as in the previous section:

```
train_model(linear_model, 20)
```

```
0.4932 0.8618 0.8203 0.9102 0.9331 0.9468 0.9555 0.9629 0.9658 0.9673 0.9687
 > 0.9707 0.9726 0.9751 0.9761 0.9761 0.9775 0.978 0.9785 0.9785
```

위는 조금 더 간결하게 train_model이라는 함수로 총 학습 epoch을 인자로 받는 함수를 정의하였다.

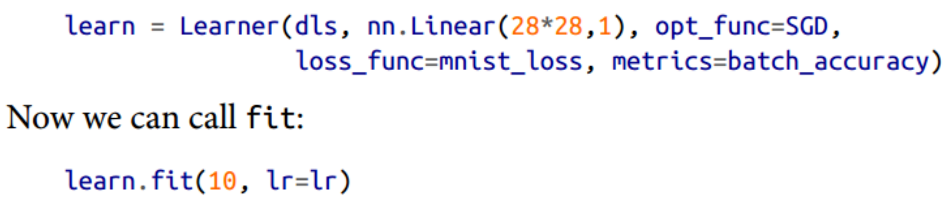
fastai는 .fit 메소드를 통해서 keras와 유사하게 작동할 수 있는 객체를 지원한다.



fastai also provides `Learner.fit`, which we can use instead of `train_model`. To create a `Learner`, we first need to create a `DataLoaders`, by passing in our training and validation `DataLoaders`:

```
dls = DataLoaders(dl, valid_dl)
```

To create a `Learner` without using an application (such as `cnn_learner`), we need to pass in all the elements that we've created in this chapter: the `DataLoaders`, the model, the optimization function (which will be passed the parameters), the loss function, and optionally any metrics to print:



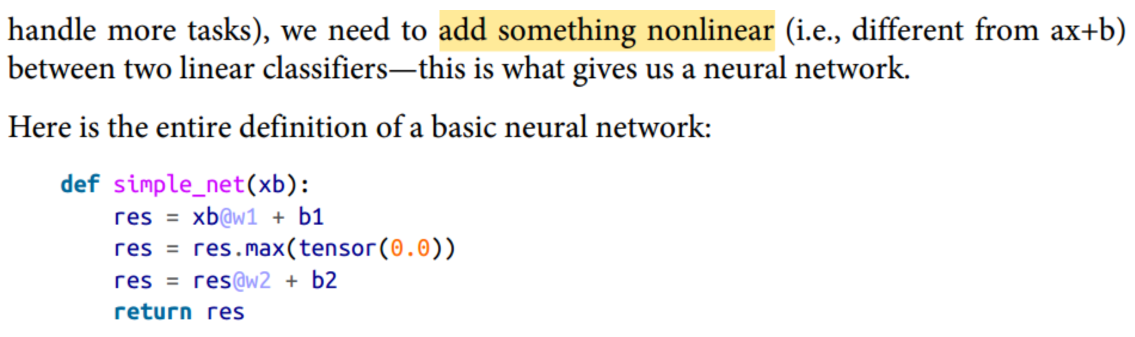
```
learn = Learner(dls, nn.Linear(28*28,1), opt_func=SGD,
                loss_func=mnist_loss, metrics=batch_accuracy)
```

Now we can call `fit`:

```
learn.fit(10, lr=lr)
```

## Adding a Nonlinearity

모델을 구성하는 중요 요소 중의 하나로 layer마다 사이사이에 activation 함수라는 것을 들 수 있다. 아래는 이러한 함수들 중에 널리 쓰이는 ReLU라는 함수를 구현해놓은 부분이다.

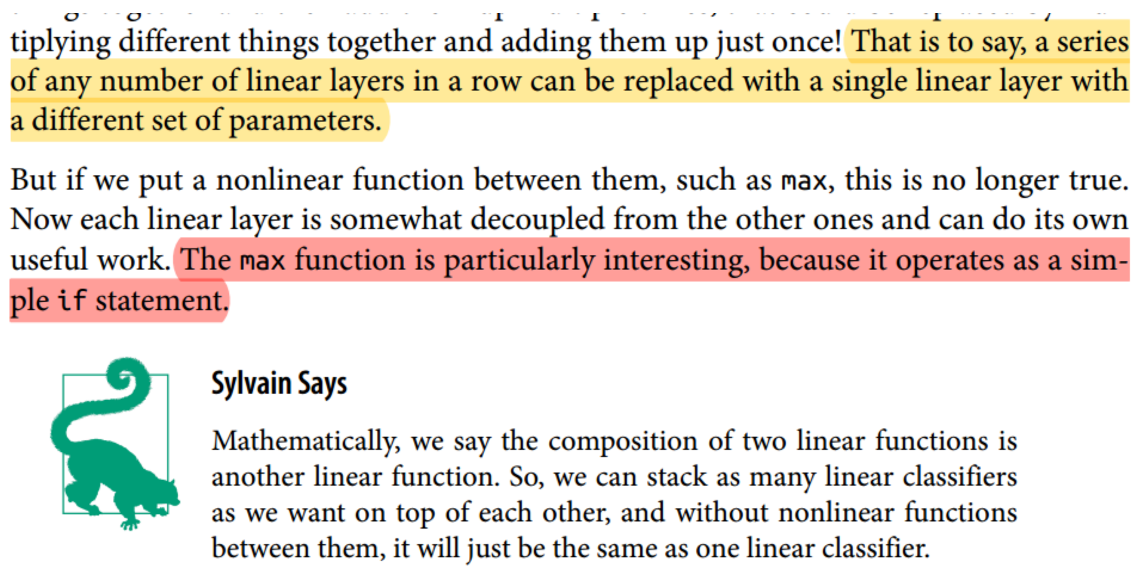


handle more tasks), we need to add something nonlinear (i.e., different from ax+b) between two linear classifiers—this is what gives us a neural network.

Here is the entire definition of a basic neural network:

```
def simple_net(xb):
    res = xb@w1 + b1
    res = res.max(tensor(0.0))
    res = res@w2 + b2
    return res
```

Linear 레이어를 아무리 많이 쌓더라도, 결국은 하나의 linear 레이어로 표현가능하기 때문에, 레이어 사이사이에 비선형 성을 추가한다.



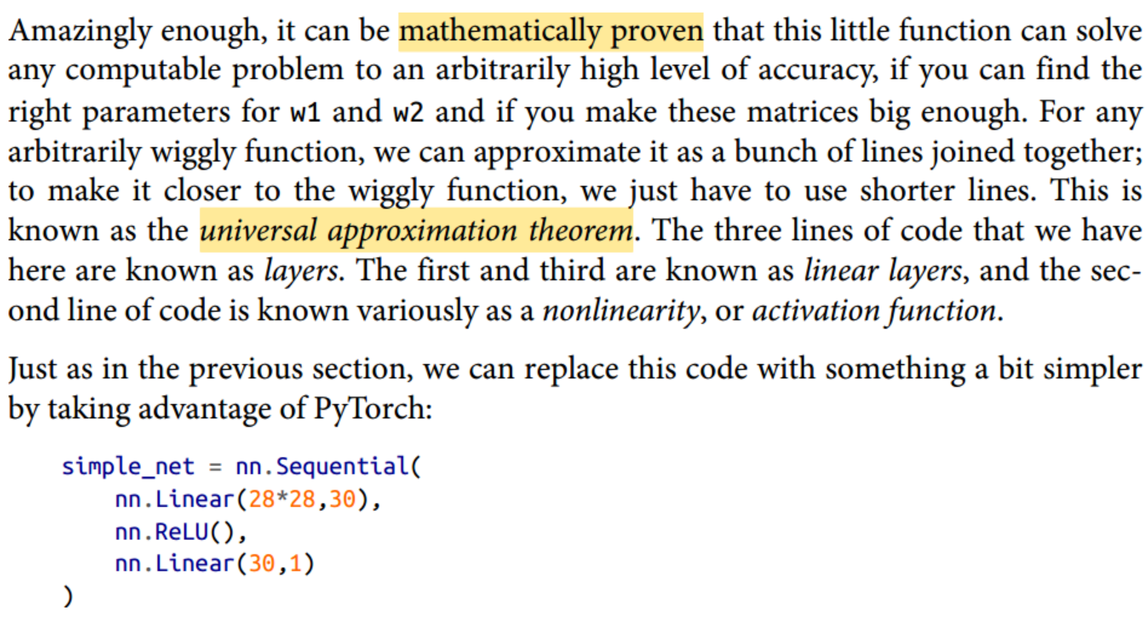
tiplying different things together and adding them up just once! That is to say, a series of any number of linear layers in a row can be replaced with a single linear layer with a different set of parameters.

But if we put a nonlinear function between them, such as `max`, this is no longer true. Now each linear layer is somewhat decoupled from the other ones and can do its own useful work. The `max` function is particularly interesting, because it operates as a simple `if` statement.

**Sylvain Says**

Mathematically, we say the composition of two linear functions is another linear function. So, we can stack as many linear classifiers as we want on top of each other, and without nonlinear functions between them, it will just be the same as one linear classifier.

Universal approximation theorem 의미



Amazingly enough, it can be mathematically proven that this little function can solve any computable problem to an arbitrarily high level of accuracy, if you can find the right parameters for `w1` and `w2` and if you make these matrices big enough. For any arbitrarily wiggly function, we can approximate it as a bunch of lines joined together; to make it closer to the wiggly function, we just have to use shorter lines. This is known as the *universal approximation theorem*. The three lines of code that we have here are known as *layers*. The first and third are known as *linear layers*, and the second line of code is known variously as a *nonlinearity*, or *activation function*.

Just as in the previous section, we can replace this code with something a bit simpler by taking advantage of PyTorch:

```
simple_net = nn.Sequential(
    nn.Linear(28*28,30),
    nn.ReLU(),
    nn.Linear(30,1)
)
```

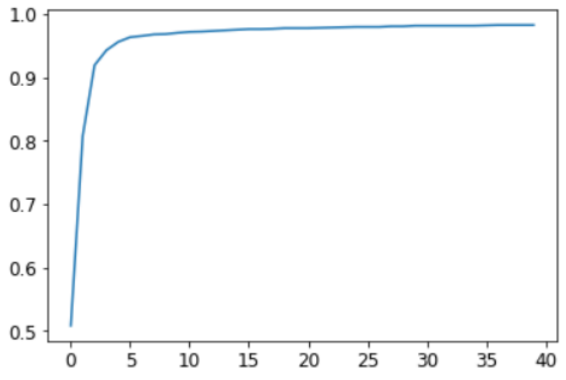
keras는 history를 써서 학습 곡선을 얻을 수 있었다면, 여기서는 recoder라는 메소드를 호출하여 유사한 방식으로 학습 곡선을 얻을 수 있다. 다음은 accuracy를 그린 것.

```
learn = Learner(dls, simple_net, opt_func=SGD,
                loss_func=mnist_loss, metrics=batch_accuracy)
```

```
learn.fit(40, 0.1)
```

We're not showing the 40 lines of output here to save room; the training process is recorded in `learn.recorder`, with the table of output stored in the `values` attribute, so we can plot the accuracy over training:

```
plt.plot(L(learn.recorder.values).itemgot(2));
```



배운 것을 요약하자면 다음과 같다.

And we can view the final accuracy:

```
learn.recorder.values[-1][2]
```

```
0.982826292514801
```

At this point, we have something that is rather magical:

- A function that can solve any problem to any level of accuracy (the neural network) given the correct set of parameters
- A way to find the best set of parameters for any function (stochastic gradient descent)

마지막으로 마무리는 resnet18을 보여준다.

Here is what happens when we train an 18-layer model using the same approach we saw in Chapter 1:

```
dls = ImageDataLoaders.from_folder(path)
learn = cnn_learner(dls, resnet18, pretrained=False,
                    loss_func=F.cross_entropy, metrics=accuracy)
learn.fit_one_cycle(1, 0.1)
```

| epoch | train_loss | valid_loss | accuracy | time |
|---|---|---|---|---|
| 0 | 0.082089 | 0.009578 | 0.997056 | 00:11 |

Nearly 100% accuracy! That's a big difference compared to our simple neural net. But as you'll learn in the remainder of this book, there are just a few little tricks you need to use to get such great results from scratch yourself. You already know the key foundational pieces. (Of course, even when you know all the tricks, you'll nearly always want to work with the prebuilt classes provided by PyTorch and fastai, because they save you from having to think about all the little details yourself.)

## Jargon Recap

Jargon 에 대해서 간단하게 정리를 해준다.

*Activations*
Numbers that are calculated (both by linear and nonlinear layers)

*Parameters*
Numbers that are randomly initialized, and optimized (that is, the numbers that define the model)

제대로 학습이 되는 중에 parameter와 activation이 어떻게 동작하는지, 그 "숫자"에 집중하여 살펴볼 줄 알아야 좋은 implementer가 될 수 있음을 강조한다.

We will often talk in this book about activations and parameters. Remember that they have specific meanings. They are numbers. They are not abstract concepts, but they are actual specific numbers that are in your model. Part of becoming a good deep learning practitioner is getting used to the idea of looking at your activations and parameters, and plotting them and testing whether they are behaving correctly.

Our activations and parameters are all contained in *tensors*. These are simply regularly shaped arrays—for example, a matrix. Matrices have rows and columns; we call these the *axes* or *dimensions*. The number of dimensions of a tensor is its *rank*. There are some special tensors:

- Rank-0: scalar
- Rank-1: vector
- Rank-2: matrix

용어 정리 및 chapter 2의 강조

| Term | Meaning |
|---|---|
| ReLU | Function that returns 0 for negative numbers and doesn't change positive numbers. |
| Mini-batch | A small group of inputs and labels gathered together in two arrays. A gradient descent step is updated on this batch (rather than a whole epoch). |
| Forward pass | Applying the model to some input and computing the predictions. |
| Loss | A value that represents how well (or badly) our model is doing. |
| Gradient | The derivative of the loss with respect to some parameter of the model. |
| Backward pass | Computing the gradients of the loss with respect to all model parameters. |
| Gradient descent | Taking a step in the direction opposite to the gradients to make the model parameters a little bit better. |
| Learning rate | The size of the step we take when applying SGD to update the parameters of the model. |

***Choose Your Own Adventure* Reminder**

Did you choose to skip over Chapters 2 and 3, in your excitement to peek under the hood? Well, here's your reminder to head back to Chapter 2 now, because you'll be needing to know that stuff soon!