Double-click (or enter) to edit

```
!pip install d2l==1.0.3
```

Show hidden output

COSE474-2024F: Deep Learning

5.2. Implementation of Multilayer Perceptrons

```
import torch
from torch import nn
from d2l import torch as d2l
```

5.2.1. Implementation from Scratch

5.2.1.1. Initializing Model Parameters

Recall that Fashion-MNIST contains 10 classes, and that each image consists of a $28 \times 28 = 784$ grid of grayscale pixel values. As before we will disregard the spatial structure among the pixels for now, so we can think of this as a classification dataset with 784 input features and 10 classes. To begin, we will [**implement an MLP with one hidden layer and 256 hidden units.**] Both the number of layers and their width are adjustable (they are considered hyperparameters). Typically, we choose the layer widths to be divisible by larger powers of 2. This is computationally efficient due to the way memory is allocated and addressed in hardware.

Again, we will represent our parameters with several tensors. Note that *for every layer*, we must keep track of one weight matrix and one bias vector. As always, we allocate memory for the gradients of the loss with respect to these parameters.

In the code below we use `nn.Parameter` to automatically register a class attribute as a parameter to be tracked by `autograd` (:numref: `sec_autograd`).

```
class MLPScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, num_hiddens, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W1 = nn.Parameter(torch.randn(num_inputs, num_hiddens) * sigma)
        self.b1 = nn.Parameter(torch.zeros(num_hiddens))
        self.W2 = nn.Parameter(torch.randn(num_hiddens, num_outputs) * sigma)
        self.b2 = nn.Parameter(torch.zeros(num_outputs))
```

5.2.1.2. Model

To make sure we know how everything works, we will implement the ReLU activation ourselves rather than invoking the built-in relu function directly.

```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

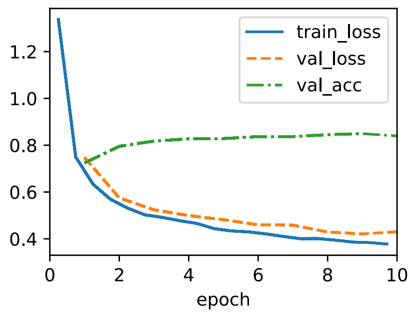
Since we are disregarding spatial structure, we `reshape` each two-dimensional image into a flat vector of length `num_inputs`. Finally, we (**implement our model**) with just a few lines of code. Since we use the framework built-in autograd this is all that it takes.

```
@d2l.add_to_class(MLPScratch)
def forward(self, X):
    X = X.reshape((-1, self.num_inputs))
    H = relu(torch.matmul(X, self.W1) + self.b1)
    return torch.matmul(H, self.W2) + self.b2
```

5.2.1.3. Training

Fortunately, the training loop for MLPs is exactly the same as for softmax regression. We define the model, data, and trainer, then finally invoke the fit method on model and data.

```
model = MLPScratch(num_inputs=784, num_outputs=10, num_hiddens=256, lr=0.1)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



5.2.2. Concise Implementation

As you might expect, by relying on the high-level APIs, we can implement MLPs even more concisely.

5.2.2.1. Model

Compared with our concise implementation of softmax regression implementation (Section 4.5), the only difference is that we add *two* fully connected layers where we previously added only one. The first is the hidden layer, the second is the output layer.

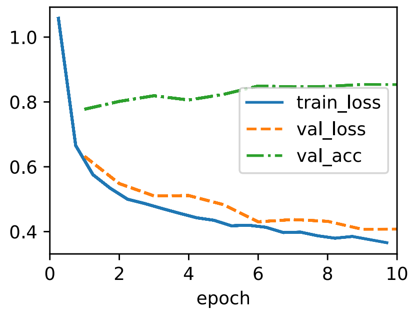
```
class MLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(), nn.LazyLinear(num_hiddens),
                                 nn.ReLU(), nn.LazyLinear(num_outputs))
```

Previously, we defined forward methods for models to transform input using the model parameters. These operations are essentially a pipeline: you take an input and apply a transformation (e.g., matrix multiplication with weights followed by bias addition), then repetitively use the output of the current transformation as input to the next transformation. However, you may have noticed that no forward method is defined here. In fact, MLP inherits the forward method from the Module class (Section 3.2.2) to simply invoke self.net(X) (X is input), which is now defined as a sequence of transformations via the Sequential class. The Sequential class abstracts the forward process enabling us to focus on the transformations. We will further discuss how the Sequential class works in Section 6.1.2.

5.2.2.2. Training

The training loop is exactly the same as when we implemented softmax regression. This modularity enables us to separate matters concerning the model architecture from orthogonal considerations.

```
model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
trainer.fit(model, data)
```



5.2.3. Summary

Now that we have more practice in designing deep networks, the step from a single to multiple layers of deep networks does not pose such a significant challenge any longer. In particular, we can reuse the training algorithm and data loader. Note, though, that implementing MLPs from scratch is nonetheless messy: naming and keeping track of the model parameters makes it difficult to extend models. For instance, imagine wanting to insert another layer between layers 42 and 43. This might now be layer 42b, unless we are willing to perform sequential renaming. Moreover, if we implement the network from scratch, it is much more difficult for the framework to perform meaningful performance optimizations.

Nonetheless, you have now reached the state of the art of the late 1980s when fully connected deep networks were the method of choice for neural network modeling. Our next conceptual step will be to consider images. Before we do so, we need to review a number of statistical basics and details on how to compute models efficiently.

```
Start coding or generate with AI.
```