Linear Regression Implementation

In [1]:
```
%matplotlib inline
import torch
from d2l import torch as d2l
```

Defining the Model

Initialized the weights by drawing random numbers from
a normal distribution with mean 0 and a standard deviation of 0.01.

In [2]:
```
class LinearRegressionScratch(d2l.Module):  #@save
    def __init__(self, num_inputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.w = torch.normal(0, sigma, (num_inputs, 1), requires_grad=True)
        self.b = torch.zeros(1, requires_grad=True)
```

The following forward function is registered as a method in the
LinearRegressionScratch class via add_to_class.

In [3]:
```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def forward(self, X):
    """The linear regression model."""
    return torch.matmul(X, self.w) + self.b
```

Defining the squared loss Function

In [4]:
```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

Defining the Optimization Algorithm

In [5]:
```
class SGD(d2l.HyperParameters):  #@save
    def __init__(self, params, lr):
        """Minibatch stochastic gradient descent."""
        self.save_hyperparameters()

    def step(self):
        for param in self.params:
            param -= self.lr * param.grad

    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()
```

In [ ]:
```
Defined the configure_optimizers method, which returns an instance of the SG
```

In [6]:
```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

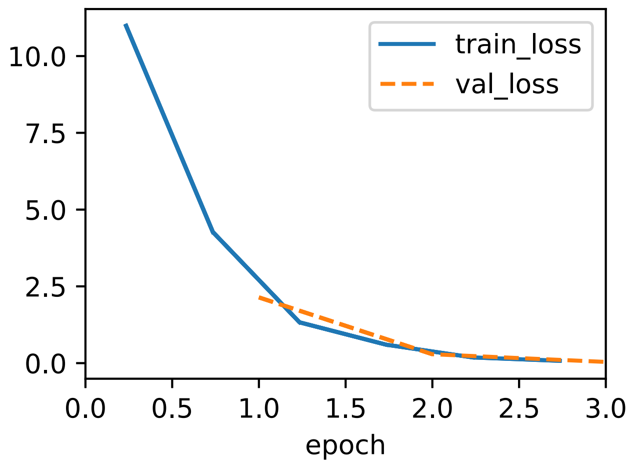
In [ ]:
```
Training Phase
```

In [7]:
```
@d2l.add_to_class(d2l.Trainer)  #@save
def prepare_batch(self, batch):
    return batch

@d2l.add_to_class(d2l.Trainer)  #@save
def fit_epoch(self):
    self.model.train()
    for batch in self.train_dataloader:
        loss = self.model.training_step(self.prepare_batch(batch))
        self.optim.zero_grad()
        with torch.no_grad():
            loss.backward()
            if self.gradient_clip_val > 0:  # To be discussed later
                self.clip_gradients(self.gradient_clip_val, self.model)
            self.optim.step()
        self.train_batch_idx += 1
    if self.val_dataloader is None:
        return
    self.model.eval()
    for batch in self.val_dataloader:
        with torch.no_grad():
            self.model.validation_step(self.prepare_batch(batch))
        self.val_batch_idx += 1
```

Here we use the SyntheticRegressionData class and pass in some ground-truth parameters.
Then, we train our model with the learning rate lr=0.03 and set max_epochs=3.

In [8]:
```
model = LinearRegressionScratch(2, lr=0.03)
data = d2l.SyntheticRegressionData(w=torch.tensor([2, -3.4]), b=4.2)
trainer = d2l.Trainer(max_epochs=3)
trainer.fit(model, data)
```



In [9]:
```
print(f'error in estimating w: {data.w - model.w.reshape(data.w.shape)}')
print(f'error in estimating b: {data.b - model.b}')
```

```
error in estimating w: tensor([ 0.1309, -0.1180], grad_fn=<SubBackward0>)
error in estimating b: tensor([0.2138], grad_fn=<RsubBackward1>)
```

By putting a fully functional neural network model and training loop into place, we made a big advancement toward
developing deep learning systems.
We created a data loader, a model, a loss function, an optimization method, and
a visualisation and monitoring tool as part of this approach. We achieved this by creating a Python object that
consists of all necessary elements for model training.
Even though it is not yet professional-grade code, it is fully functional and could already assist you in solving simple issues rapidly.