


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

 Show hidden output

## COSE474-2024F: Deep Learning

### ✓ 3.2. Object-Oriented Design for Implementation

In our introduction to linear regression, we walked through various components including the data, the model, the loss function, and the optimization algorithm. Indeed, linear regression is one of the simplest machine learning models. Training it, however, uses many of the same components that other models in this book require. Therefore, before diving into the implementation details it is worth designing some of the APIs that we use throughout. Treating components in deep learning as objects, we can start by defining classes for these objects and their interactions. This object-oriented design for implementation will greatly streamline the presentation and you might even want to use it in your projects.

Inspired by open-source libraries such as PyTorch Lightning, at a high level we wish to have three classes: (i) Module contains models, losses, and optimization methods; (ii) DataModule provides data loaders for training and validation; (iii) both classes are combined using the Trainer class, which allows us to train models on a variety of hardware platforms. Most code in this book adapts Module and DataModule. We will touch upon the Trainer class only when we discuss GPUs, CPUs, parallel training, and optimization algorithms.

```
import time
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

#### ✓ 3.2.1. Utilities

We need a few utilities to simplify object-oriented programming in Jupyter notebooks. One of the challenges is that class definitions tend to be fairly long blocks of code. Notebook readability demands short code fragments, interspersed with explanations, a requirement incompatible with the style of programming common for Python libraries. The first utility function allows us to register functions as methods in a class after the class has been created. In fact, we can do so even after we have created instances of the class! It allows us to split the implementation of a class into multiple code blocks.

```
def add_to_class(Class):
    """Register functions as methods in created class."""
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper
```

Let's have a quick look at how to use it. We plan to implement a class A with a method do. Instead of having code for both A and do in the same code block, we can first declare the class A and create an instance a.

```
class A:
    def __init__(self):
        self.b = 1

a = A()
```

Next we define the method do as we normally would, but not in class A's scope. Instead, we decorate this method by add\_to\_class with class A as its argument. In doing so, the method is able to access the member variables of A just as we would expect had it been included as part of A's definition. Let's see what happens when we invoke it for the instance a.

```
@add_to_class(A)
def do(self):
    print('Class attribute "b" is', self.b)
```

```
a.do()
```

```
↪ Class attribute "b" is 1
```

The second one is a utility class that saves all arguments in a class's **init** method as class attributes. This allows us to extend constructor call signatures implicitly without additional code.

```
class HyperParameters:
    """The base class of hyperparameters."""
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

To use it, we define our class that inherits from HyperParameters and calls save\_hyperparameters in the **init** method.

```
# Call the fully implemented HyperParameters class saved in d2l
class B(d2l.HyperParameters):
    def __init__(self, a, b, c):
        self.save_hyperparameters(ignore=['c'])
        print('self.a =', self.a, 'self.b =', self.b)
        print('There is no self.c =', not hasattr(self, 'c'))
```

```
b = B(a=1, b=2, c=3)
```

```
↪ self.a = 1 self.b = 2
   There is no self.c = True
```

The final utility allows us to plot experiment progress interactively while it is going on. In deference to the much more powerful (and complex) [TensorBoard](#) we name it ProgressBoard. The implementation is deferred to :numref:sec\_utils. For now, let's simply see it in action.

The draw method plots a point (x, y) in the figure, with label specified in the legend. The optional every\_n smooths the line by only showing 1/n points in the figure. Their values are averaged from the n neighbor points in the original figure.

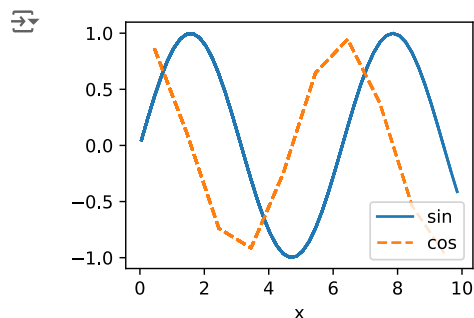
```
class ProgressBoard(d2l.HyperParameters): #@save
    """The board that plots data points in animation."""
    def __init__(self, xlabel=None, ylabel=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 ls=['-', '--', '-.', ':'], colors=['C0', 'C1', 'C2'],
                 fig=None, axes=None, figsize=(3.5, 2.5), display=True):
        self.save_hyperparameters()

    def draw(self, x, y, label, every_n=1):
        raise NotImplemented
```

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

In the following example, we draw sin and cos with a different smoothness. If you run this code block, you will see the lines grow in animation

```
board = d2l.ProgressBoard('x')
for x in np.arange(0, 10, 0.1):
    board.draw(x, np.sin(x), 'sin', every_n=2)
    board.draw(x, np.cos(x), 'cos', every_n=10)
```



### ✓ 3.2.2. Models

The `Module` class is the base class of all models we will implement. At the very least we need three methods. The first, `__init__`, stores the learnable parameters, the `training_step` method accepts a data batch to return the loss value, and finally, `configure_optimizers` returns the optimization method, or a list of them, that is used to update the learnable parameters. Optionally we can define `validation_step` to report the evaluation measures. Sometimes we put the code for computing the output into a separate `forward` method to make it more reusable.

```
class Module(nn.Module, d2l.HyperParameters): #@save
    """The base class of models."""
    def __init__(self, plot_train_per_epoch=2, plot_valid_per_epoch=1):
        super().__init__()
        self.save_hyperparameters()
        self.board = ProgressBoard()

    def loss(self, y_hat, y):
        raise NotImplementedError

    def forward(self, X):
        assert hasattr(self, 'net'), 'Neural network is defined'
        return self.net(X)

    def plot(self, key, value, train):
        """Plot a point in animation."""
        assert hasattr(self, 'trainer'), 'Trainer is not initied'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / \
                self.trainer.num_train_batches
            n = self.trainer.num_train_batches / \
                self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / \
                self.plot_valid_per_epoch
        self.board.draw(x, value.to(d2l.cpu()).detach().numpy(),
                        ('train_' if train else 'val_') + key,
                        every_n=int(n))

    def training_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=True)
        return l

    def validation_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=False)

    def configure_optimizers(self):
        raise NotImplementedError
```

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

You may notice that `Module` is a subclass of `nn.Module`, the base class of neural networks in PyTorch. It provides convenient features for handling neural networks. For example, if we define a forward method, such as `forward(self, X)`, then for an instance `a` we can invoke this method by `a(X)`. This works since it calls the forward method in the built-in `call` method. You can find more details and examples about `nn`.

### ✓ 3.2.3. Data

The `DataModule` class is the base class for data. Quite frequently the `init` method is used to prepare the data. This includes downloading and preprocessing if needed. The `train_dataloader` returns the data loader for the training dataset. A data loader is a (Python) generator that yields a data batch each time it is used. This batch is then fed into the `training_step` method of `Module` to compute the loss. There is an optional `val_dataloader` to return the validation dataset loader. It behaves in the same manner, except that it yields data batches for the `validation_step` method in `Module`.

```
class DataModule(d2l.HyperParameters):
    """The base class of data."""
    def __init__(self, root='../data', num_workers=4):
        self.save_hyperparameters()

    def get_dataloader(self, train):
        raise NotImplementedError
```

```
def train_dataloader(self):
    return self.get_dataloader(train=True)

def val_dataloader(self):
    return self.get_dataloader(train=False)
```

### ✓ 3.2.4. Training

The Trainer class trains the learnable parameters in the Module class with data specified in DataModule. The key method is fit, which accepts two arguments: model, an instance of Module, and data, an instance of DataModule. It then iterates over the entire dataset max\_epochs times to train the model. As before, we will defer the implementation of this method to later chapters.

```
class Trainer(d2l.HyperParameters):
    """The base class for training models with data."""
    def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
        self.save_hyperparameters()
        assert num_gpus == 0, 'No GPU support yet'

    def prepare_data(self, data):
        self.train_dataloader = data.train_dataloader()
        self.val_dataloader = data.val_dataloader()
        self.num_train_batches = len(self.train_dataloader)
        self.num_val_batches = (len(self.val_dataloader)
                                if self.val_dataloader is not None else 0)

    def prepare_model(self, model):
        model.trainer = self
        model.board.xlim = [0, self.max_epochs]
        self.model = model

    def fit(self, model, data):
        self.prepare_data(data)
        self.prepare_model(model)
        self.optim = model.configure_optimizers()
        self.epoch = 0
        self.train_batch_idx = 0
        self.val_batch_idx = 0
        for self.epoch in range(self.max_epochs):
            self.fit_epoch()

    def fit_epoch(self):
        raise NotImplementedError
```

### ✓ 3.2.5. Summary

To highlight the object-oriented design for our future deep learning implementation, the above classes simply show how their objects store data and interact with each other. We will keep enriching implementations of these classes, such as via @add\_to\_class, in the rest of the book. Moreover, these fully implemented classes are saved in the D2L library, a lightweight toolkit that makes structured modeling for deep learning easy. In particular, it facilitates reusing many components between projects without changing much at all. For instance, we can replace just the optimizer, just the model, just the dataset, etc.; this degree of modularity pays dividends throughout the book in terms of conciseness and simplicity (this is why we added it) and it can do the same for your own projects.

Start coding or [generate](#) with AI.

