T05460301-01

Deep Learning (Fall 2023)

Ikbeom Jang

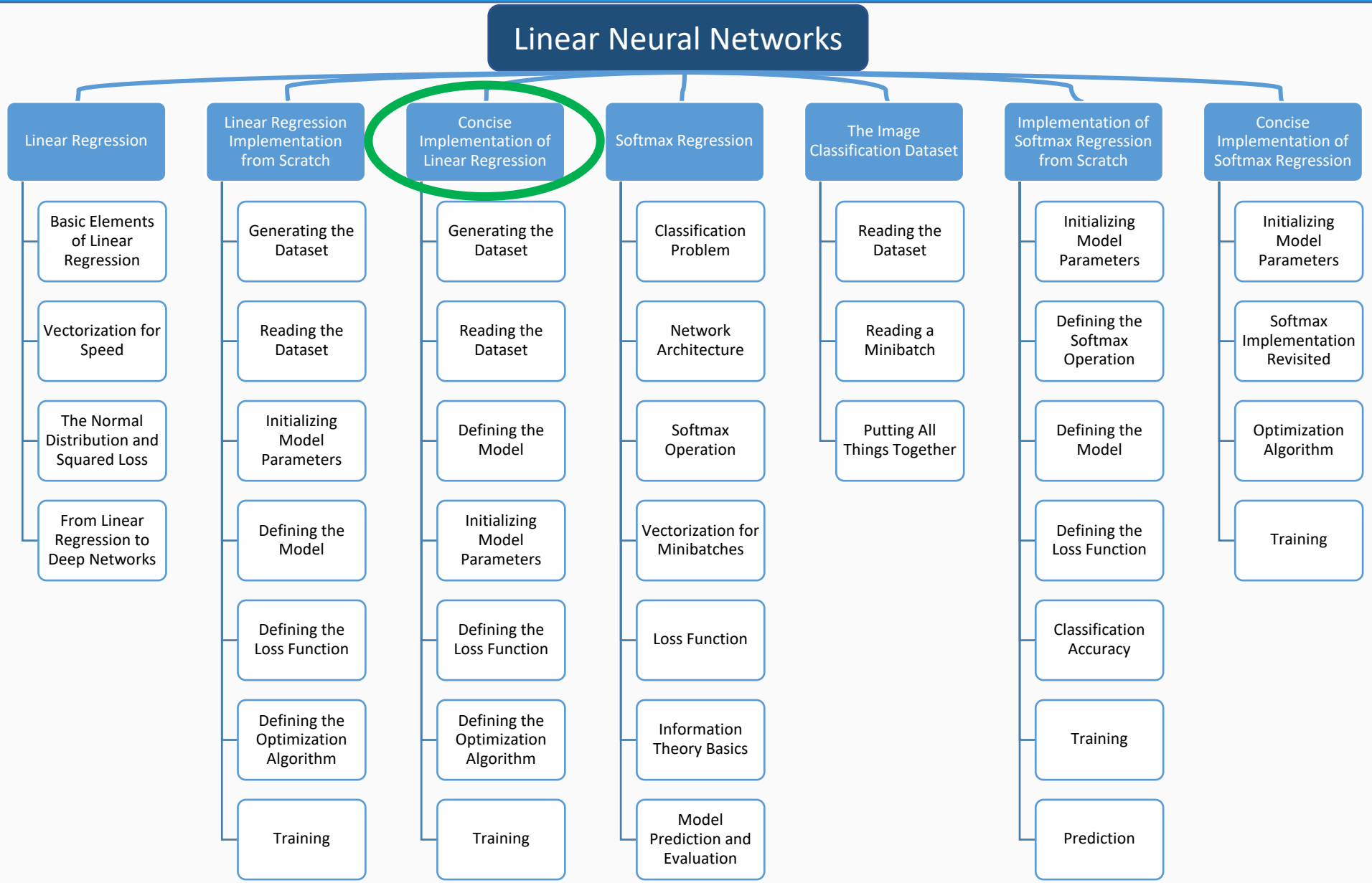
ijang@hufs.ac.kr

CES HUFS

Linear Neural Networks

- Concise Implementation of Linear Regression

Contents


Linear Neural Networks
Linear Regression
Basic Elements of Linear Regression
Vectorization for Speed
The Normal Distribution and Squared Loss
From Linear Regression to Deep Networks
Linear Regression Implementation from Scratch
Generating the Dataset
Reading the Dataset
Initializing Model Parameters
Defining the Model
Defining the Loss Function
Defining the Optimization Algorithm
Training
Concise Implementation of Linear Regression
Generating the Dataset
Reading the Dataset
Defining the Model
Initializing Model Parameters
Defining the Loss Function
Defining the Optimization Algorithm
Training
Softmax Regression
Classification Problem
Network Architecture
Softmax Operation
Vectorization for Minibatches
Loss Function
Information Theory Basics
Model Prediction and Evaluation
The Image Classification Dataset
Reading the Dataset
Reading a Minibatch
Putting All Things Together
Implementation of Softmax Regression from Scratch
Initializing Model Parameters
Defining the Softmax Operation
Defining the Model
Defining the Loss Function
Classification Accuracy
Training
Prediction
Concise Implementation of Softmax Regression
Initializing Model Parameters
Softmax Implementation Revisited
Optimization Algorithm
Training

Concise Implementation of Linear Regression

- We will learn how to implement the linear regression model concisely by using high-level APIs of DL frameworks.

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

Generating the Dataset

- We will generate the same dataset as in Section 3.2.

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, np, npx
npx.set_np()

true_w = np.array([2, -3.4])
true_b = 4.2
features, labels = d2l.synthetic_data(true_w, true_b, 1000)
```

$$\mathbf{X}$$

Reading the Dataset

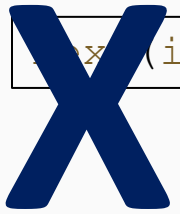
- Use the existing API in a framework to read data.
  - We pass in $features$ and $labels$ as arguments and specify batch_size when instantiating a data iterator object.
  - The boolean value is_train indicates whether or not to shuffle the data on each epoch.

```
def load_array(data_arrays, batch_size, is_train=True):  #@save
    """Construct a Gluon data iterator."""
    dataset = gluon.data.ArrayDataset(*data_arrays)
    return gluon.data.DataLoader(dataset, batch_size, shuffle=is_train)

batch_size = 10
data_iter = load_array((features, labels), batch_size)
```

- Now we can use $data\_iter$ in much the same way as we called the $data\_iter$ function in Section 3.2.
  - We use $iter$ to construct a Python iterator.
  - We use $next$ to obtain the first item from the iterator.

```
next(iter(data_iter))
```



Defining the Model

- We use a framework's predefined layers to focus on the layers used to construct the model rather than focusing on the implementation.

- The layer in a single-layer neural network (Fig. 3.1.2) is said to be $fully-connected$ because each of its inputs is connected to each of its outputs by means of a matrix-vector multiplication.
  - In PyTorch, the fully-connected layer is defined in the $Linear$ and $LazyLinear$ classes.
  - $LazyLinear$ allows users to specify merely the output dimension.
  - $Linear$ additionally asks for how many inputs go into this layer.
  - We will use such "lazy" layers whenever we can for simplicity.

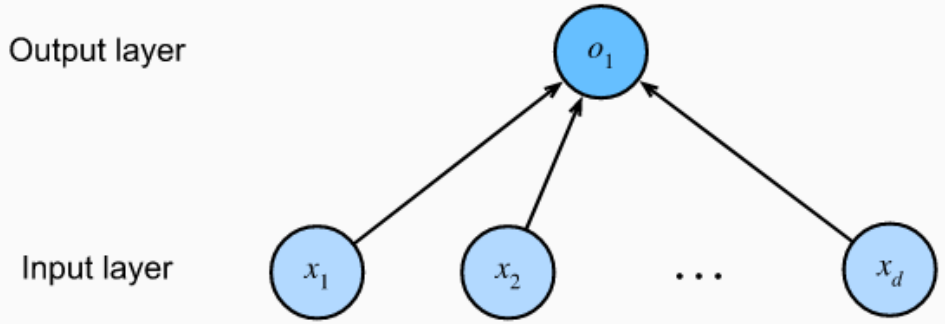


Fig. 3.1.2 Linear regression is a single-layer neural network.

Defining the Model

```
class LinearRegression(d2l.Module): #@save
    """The linear regression model implemented with high-level APIs."""
    def __init__(self, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.LazyLinear(1)
        self.net.weight.data.normal_(0, 0.01)
        self.net.bias.data.fill_(0)
```

- In the *forward* method we just invoke the built-in __call__ method of the predefined layers to compute the outputs.

```
@d2l.add_to_class(LinearRegression) #@save
def forward(self, X):
    return self.net(X)
```

Defining the Loss Function

- The MSELoss class computes the mean squared error (without the 1/2 factor in (3.1.5)).
- By default, MSELoss returns the average loss over examples.
- It is faster (and easier to use) than implementing our own.
  - nn.MSELoss()

```
@d2l.add_to_class(LinearRegression) #@save
def loss(self, y_hat, y):
    fn = nn.MSELoss()
    return fn(y_hat, y)
```

Defining the Optimization Algorithm

- Minibatch SGD is a standard tool for optimizing neural networks and thus PyTorch supports it alongside a number of variations on this algorithm in the *optim* module. When instantiating an SGD instance, we will specify:
  - Parameters to optimize over (obtainable from our model via $self.parameters()$)
  - Learning rate (*self.lr*) required by our optimization algorithm

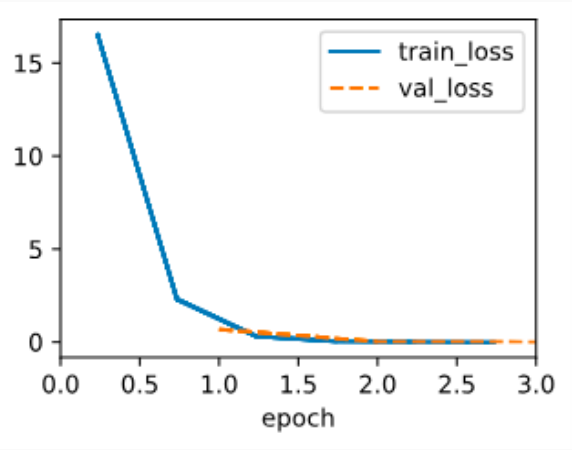
```
@d2l.add_to_class(LinearRegression) #@save
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), self.lr)
```

Training

- Notice that expressing our model through high-level APIs of a DL framework requires fewer lines of code.
- We did not have to
  - allocate parameters individually
  - define our loss function
  - implement minibatch SGD
- Once we start working with much more complex models, the advantages of the high-level API will grow considerably.

- The training loop is the same as the one we implemented from scratch.
- We just call the *fit* method, which relies on the implementation of the *fit_epoch* method, to train our model.

```
model = LinearRegression(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)
```



Training

- We compare the model parameters learned by training and the actual parameters that generated our dataset.
- Our estimated parameters are close to their true counterparts.

```
@d2l.add_to_class(LinearRegression) #@save
def get_w_b(self):
        return (self.net.weight.data, self.net.bias.data)
w, b = model.get_w_b()

print(f'error in estimating w: {data.w - w.reshape(data.w.shape)}')
print(f'error in estimating b: {data.b - b}')
```

Summary

- Using PyTorch, we can implement models much more concisely.
- In PyTorch, the $data$ module provides tools for data processing.
- The nn module defines a large number of neural network layers and many common loss functions.
- We can initialize the parameters by replacing their values with methods ending with _.