T05460301-01

Deep Learning (Fall 2023)

Ikbeom Jang

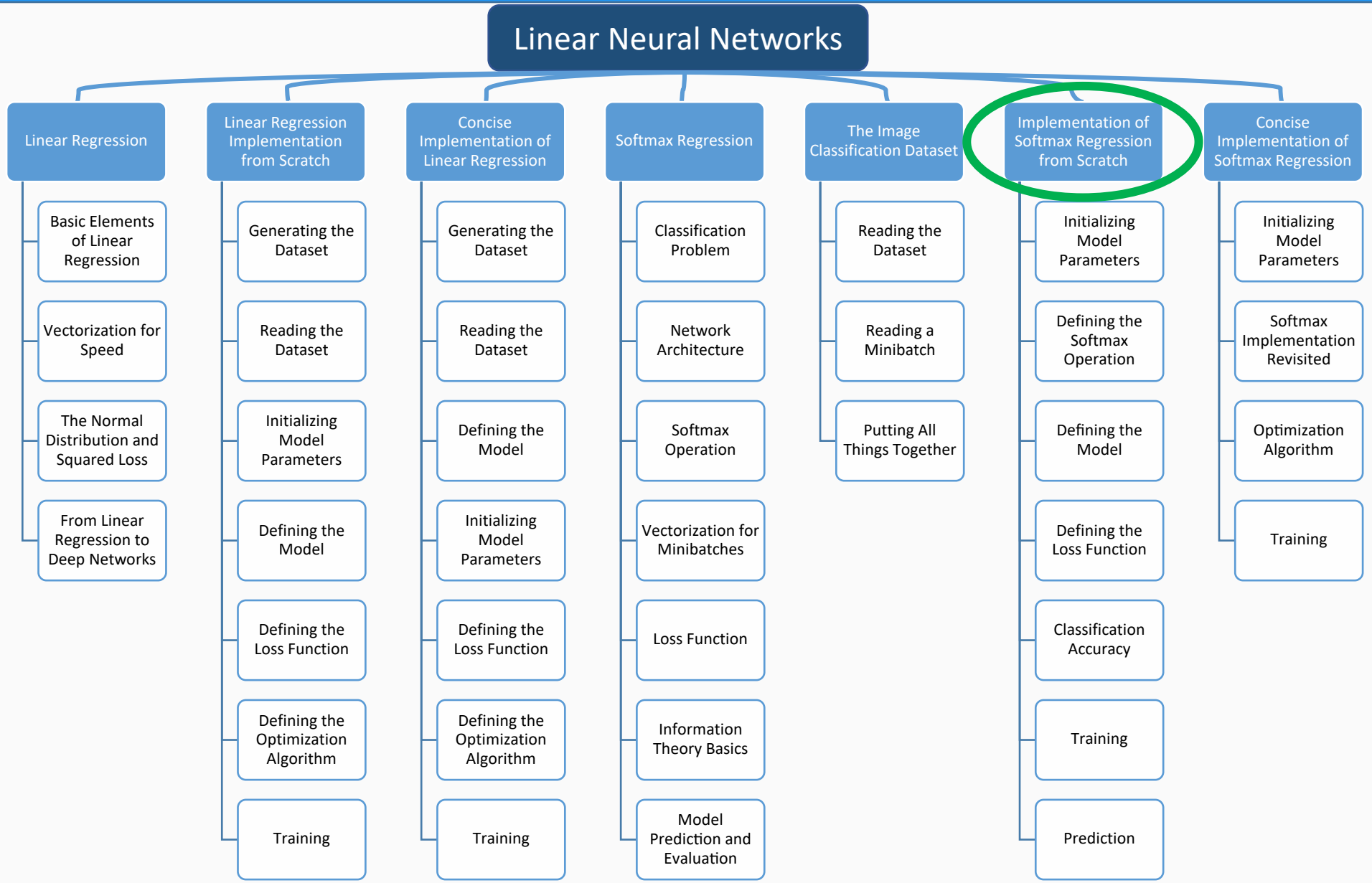
ijang@hufs.ac.kr

CES HUFS

Linear Neural Networks

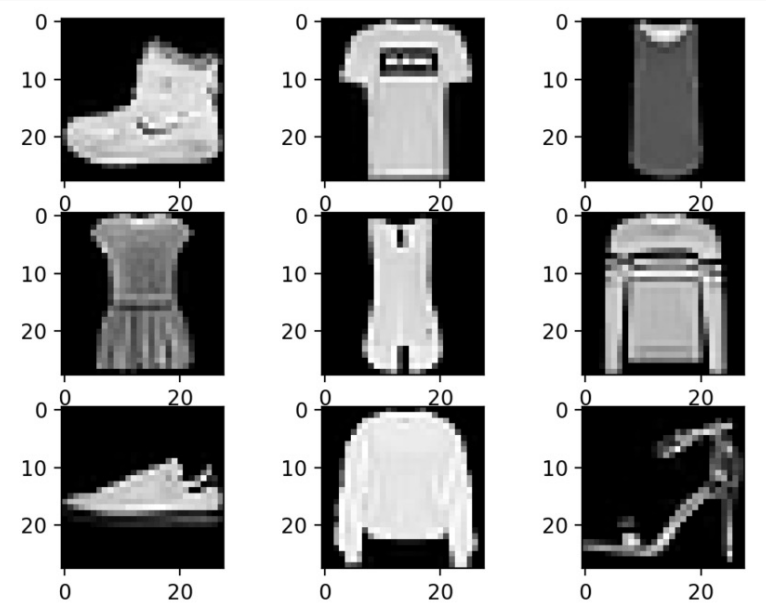
- Softmax Regression Implementation from Scratch
- Concise Implementation of Softmax Regression

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

Softmax Regression Implementation from Scratch

- We focus on defining the softmax-specific aspects of the model.
- We reuse the other components from our linear regression section, including the training loop.
- We will use the Fashion-MNIST dataset.



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

The Softmax

- Let's start with the mapping from scalars to probabilities.
- Recall the operation of the sum operator along specific dimensions in a tensor
  - Given a matrix X we can sum over all elements (by default) or only over elements in the same axis.
  - The axis variable lets us compute row and column sums:

```
X = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
X.sum(0, keepdims=True), X.sum(1, keepdims=True)
```

```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

The Softmax

- Computing the softmax requires three steps:
  - (i) exponentiation of each term
  - (ii) a sum over each row to compute the normalization constant for each example
  - (iii) division of each row by its normalization constant, ensuring that the result sums to 1

$$\mathrm{softmax}(\mathbf{X})_{ij} = \frac{\exp(\mathbf{X}_{ij})}{\sum_k \exp(\mathbf{X}_{ik})}$$

- The (logarithm of the) denominator is called the (log) partition function.
  - It was introduced in statistical physics to sum over all possible states in a thermodynamic ensemble.

```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdims=True)
    return X_exp / partition # The broadcasting mechanism is applied here
```

The Softmax

- For any input X, we turn each element into a non-negative number.
- Each row sums up to 1, as is required for a probability.
- Caution: the code above is not robust against very large or very small arguments.
  - While it is sufficient to illustrate what is happening, you should not use this code verbatim for any serious purpose.
  - Deep learning frameworks have such protections built in and we will be using the built-in softmax going forward.

```
X = torch.rand((2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

```
(tensor([[0.2511, 0.1417, 0.1158, 0.2529, 0.2385],
         [0.2004, 0.1419, 0.1957, 0.2504, 0.2117]]),
 tensor([1., 1.]))
```

The Model

- We now have everything that we need to implement the softmax regression model.
- As in our linear regression example, each instance will be represented by a fixed-length vector.
- Since the raw data here consists of 28×28 pixel images, we flatten each image, treating them as vectors of length 784.
  - In later chapters, we will introduce convolutional neural networks, which exploit the spatial structure in a more satisfying way.

- In softmax regression, the number of outputs from our network should be equal to the number of classes.
- Since our dataset has 10 classes, our network has an output dimension of 10.
- Consequently, our weights constitute a 784×10 matrix plus a 1×10 row vector for the biases.
- As with linear regression, we initialize the weights W with Gaussian noise. The biases are initialized as zeros.

The Model

```
class SoftmaxRegressionScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W = torch.normal(0, sigma, size=(num_inputs, num_outputs), requires_grad=True)
        self.b = torch.zeros(num_outputs, requires_grad=True)

    def parameters(self):
        return [self.W, self.b]
```

- The code below defines how the network maps each input to an output.
- Note that we flatten each 28×28 pixel image in the batch into a vector using reshape before passing the data through our model.

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def forward(self, X):
    X = X.reshape((-1, self.W.shape[0]))
    return softmax(torch.matmul(X, self.W) + self.b)
```

The Cross-Entropy Loss

- The cross-entropy loss is the most common loss function in all of deep learning.
  - At the moment, applications of deep learning easily cast as classification problems far outnumber those better treated as regression problems.

- Recall that cross-entropy takes the negative log-likelihood of the predicted probability assigned to the true label.

- For efficiency we avoid Python for-loops and use indexing instead.
  - In particular, the one-hot encoding in *y* allows us to select the matching terms in *y_hat*.

- To see this in action we create sample data *y_hat* with 2 examples of predicted probabilities over 3 classes and their corresponding labels *y*.
  - The correct labels are 0 and 2 respectively (i.e., the first and third class).
  - Using *y* as the indices of the probabilities in *y_hat*, we can pick out terms efficiently.

```
y = torch.tensor([0, 2])  # true labels (not one-hot encoded). It's [[1, 0, 0], [0,0,1]] if one-hot encoded.
y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])  # predicted labels
y_hat[[0, 1], y]  # for the 0th and 1st samples, choose the class in y_hat based on the true labels y
```

```
tensor([0.1000, 0.5000])
```

The Cross-Entropy Loss

- Recall the formula of cross-entropy loss:

$$l(y, \hat{y}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j$$

- Now implement the cross-entropy loss function by averaging over the logarithms of the selected probabilities.
- Focus on "y_hat[list(range(len(y_hat))), y]" – it's tricky!!
  - What it does is: for all samples, choose the class in y_hat based on the true labels y (using indexing)
  - Choosing is equivalent to multiplying (in the equation) if you recall y are one-hot vectors like [0,0,1]

```
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

cross_entropy(y_hat, y)
```

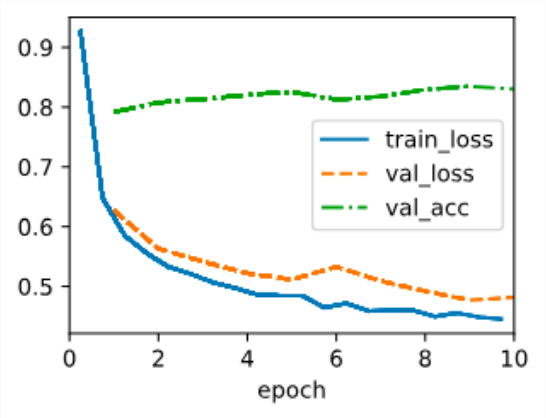
```
tensor(1.4979)
```

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

Training

- We reuse the fit method defined in Section 3.4 to train the model with 10 epochs.
- Note that the number of epochs (max_epochs), the minibatch size (batch_size), and learning rate (lr) are adjustable hyperparameters.
  - That means that while these values are not learned during our primary training loop, they still influence the performance of our model, both in terms of training and generalization performance.
- In practice you will want to choose these values based on the validation split of the data and then, ultimately, to evaluate your final model on the test split.
  - We will regard the test data of Fashion-MNIST as the validation set, thus reporting validation loss and validation accuracy on this split.

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



Prediction

- Our model is ready to classify some images.

```
X, y = next(iter(data.val_dataloader()))
preds = model(X).argmax(axis=1)
preds.shape
```

```
torch.Size([256])
```

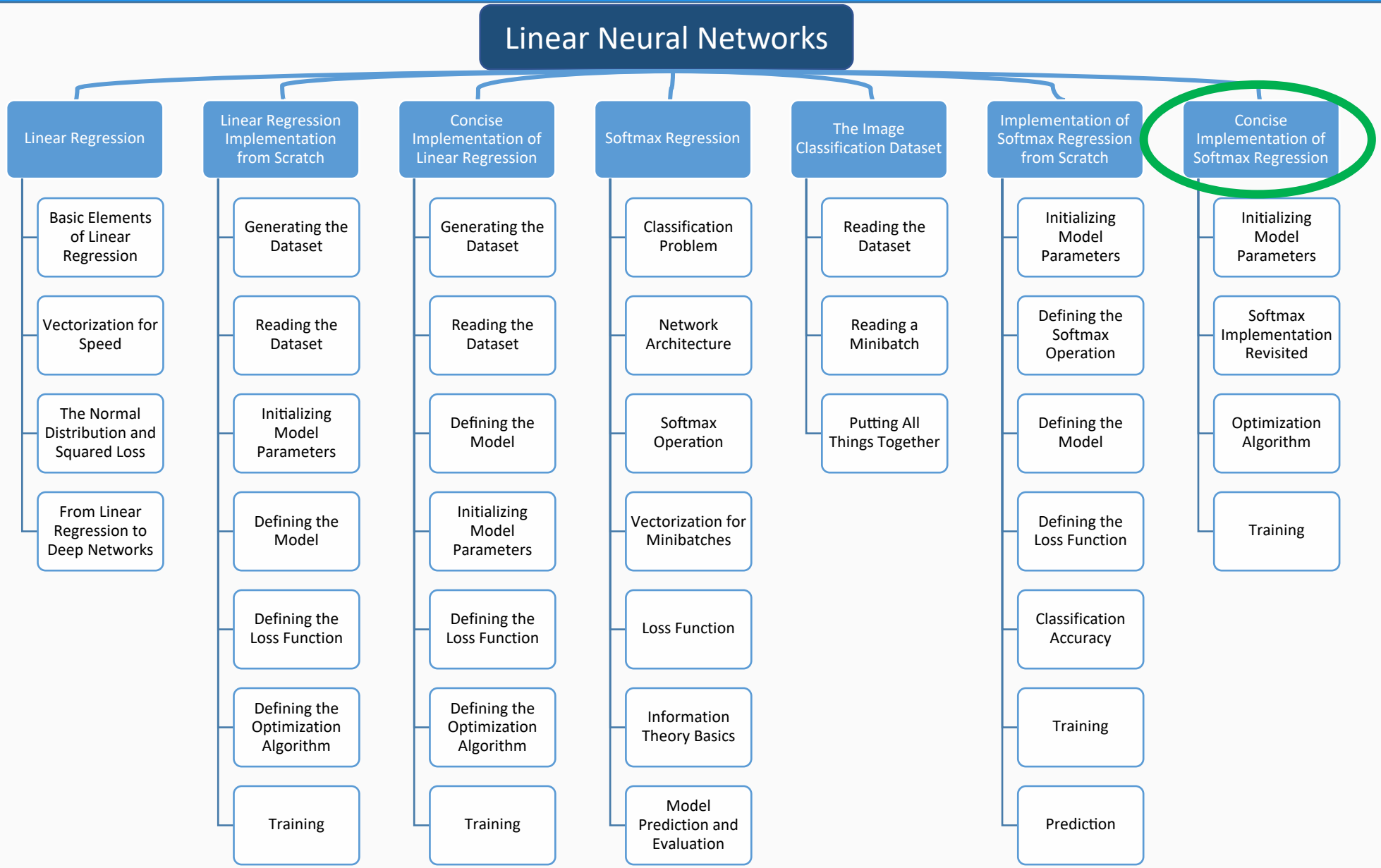
- We are usually more interested in the images we label incorrectly.
- We visualize them by comparing their actual labels (1st line of text output) with the model predictions (2nd line of text output).

```
wrong = preds.type(y.dtype) != y
X, y, preds = X[wrong], y[wrong], preds[wrong]
labels = [a+'\n'+b for a, b in zip(data.text_labels(y), data.text_labels(preds))]
data.visualize([X, y], labels=labels)
```

Summary

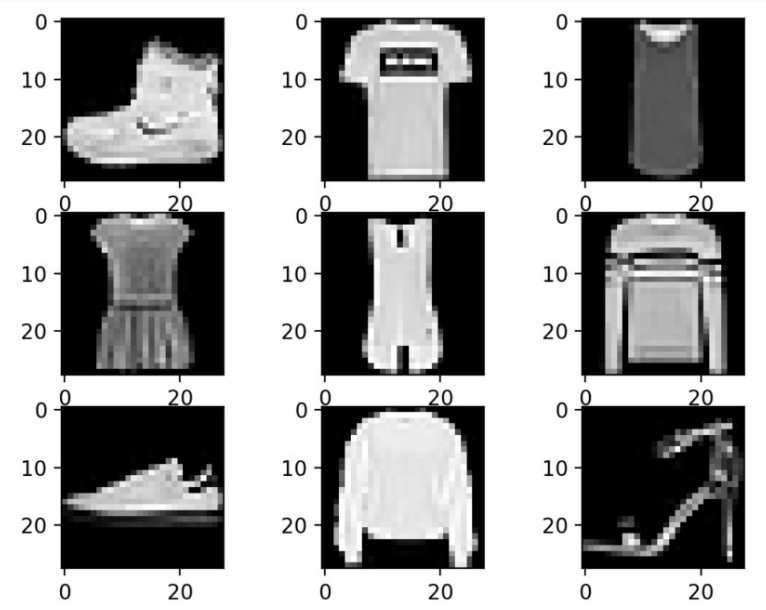
- By now we are starting to get some experience with solving linear regression and classification problems.
- With it, we have reached what would be the state of the art of 1960–1970s of statistical modeling.
- Next, we will learn how to leverage DL frameworks to implement this model much more efficiently.

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 Softmax Regression

- Just as high-level DL frameworks made it easier to implement linear regression, they are similarly convenient here.



```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

Defining the Model

- We construct our fully connected layer using the built-in layer.
  - The built-in __call__ method then invokes forward whenever we need to apply the network to some input.

- We use a Flatten layer to convert the fourth-order tensor X to second order by keeping the dimensionality along the first axis unchanged.

```
class SoftmaxRegression(d2l.Classifier): #@save
    """The softmax regression model."""
    def __init__(self, num_outputs, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(), nn.LazyLinear(num_outputs))

    def forward(self, X):
        return self.net(X)
```

Softmax

- In the previous section, we calculated our model's output and applied the cross-entropy loss.
- While this is perfectly reasonable mathematically, it is risky computationally, because of numerical underflow and overflow in the exponentiation.
  - If some of the o_k are very large, then $exp(o_k)$ might be larger than the largest number we can have for certain data types. This is called overflow.
  - Likewise, if every argument is a very large negative number, we will get underflow.
  - For instance, single precision floating point numbers approximately cover the range of 10^{-38} to 10^{38}.
  - As such, if the largest term in $\boldsymbol{o}$ lies outside the interval [−90,90], the result will not be stable.
- A way round this problem is to subtract $\bar{o} \stackrel{\text{def}}{=} \max_k o_k$ from all entries (more explanation in textbook):

$$\hat{y}_j = \frac{\exp o_j}{\sum_k \exp o_k} = \frac{\exp(o_j - \bar{o}) \exp \bar{o}}{\sum_k \exp(o_k - \bar{o}) \exp \bar{o}} = \frac{\exp(o_j - \bar{o})}{\sum_k \exp(o_k - \bar{o})}$$

Softmax

- Fortunately, we are saved by the fact that even though we are computing exponential functions, we ultimately intend to take their log (when calculating the cross-entropy loss).
- By combining softmax and cross-entropy, we can escape the numerical stability issues altogether. We have:

$$\log \hat{y}_j = \log \frac{\exp(o_j - \bar{o})}{\sum_k \exp(o_k - \bar{o})} = o_j - \bar{o} - \log \sum_k \exp(o_k - \bar{o}).$$

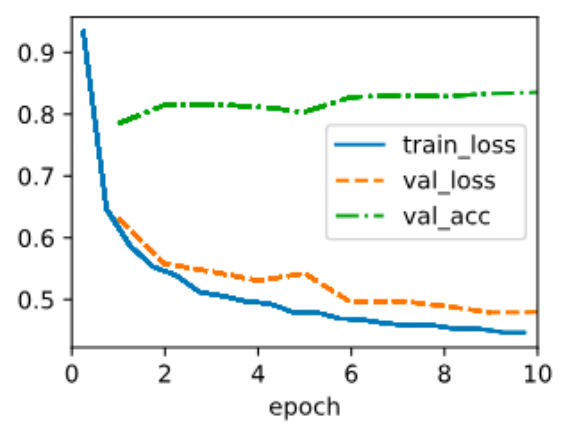
- This avoids both overflow and underflow.
- We will want to keep the conventional softmax function handy in case we ever want to evaluate the output probabilities by our model.
- But instead of passing softmax probabilities into our new loss function, we just pass the logits and compute the softmax and its log all at once inside the cross-entropy loss function.

```
@d2l.add_to_class(d2l.Classifier) #@save
def loss(self, Y_hat, Y, averaged=True):
    Y_hat = Y_hat.reshape((-1, Y_hat.shape[-1]))
    Y = Y.reshape((-1,))
    return F.cross_entropy(Y_hat, Y, reduction='mean' if averaged else 'none')
```

Training

- Next, we train our model.
- We use Fashion-MNIST images, flattened to 784-dimensional feature vectors.
- As before, this algorithm converges to a solution that is reasonably accurate

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



Summary

- High-level APIs are very convenient at hiding from their user potentially dangerous aspects, such as numerical stability.
- Moreover, they allow users to design models concisely with very few lines of code.
- This is both a blessing and a curse.
  - The benefit is that it makes things highly accessible, even to engineers who never took a single class of statistics in their life.
  - But hiding the sharp edges also comes with a price: a disincentive to add new and different components on your own, since there is little muscle memory for doing it.
  - Moreover, it makes it more difficult to fix things whenever the protective padding of a framework fails to cover all the corner cases entirely.
- As such, it is urged to review both the bare bones and the elegant versions of many of the implementations that follow.
- I hope you can build on these when you invent something new that no framework can give you.