T05460301-01

Deep Learning (Fall 2023)

Ikbeom Jang

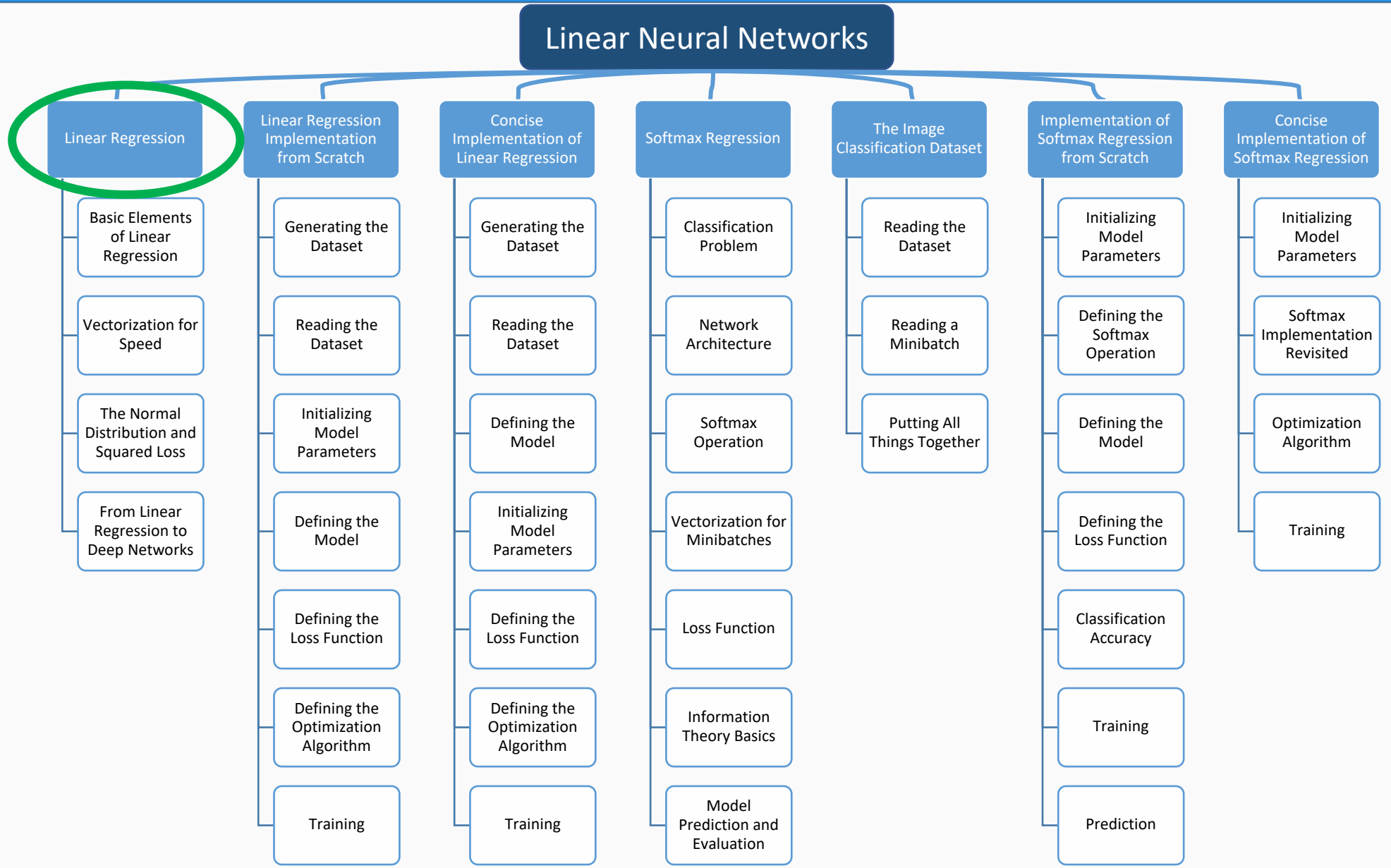
ijang@hufs.ac.kr

CES HUFS

Linear Neural Networks

- Linear Regression
- Linear Regression Implementation from Scratch
- 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

Linear Regression

- *Regression* refers to a set of methods for modeling the relationship between one or more independent variables and a dependent variable.
  - The purpose of regression is most often to *characterize* the relationship between the inputs and outputs.
  - Machine learning, on the other hand, is most often concerned with *prediction*.

https://elsaghirscience.weebly.com/predicting.html

https://www.sfehrlich.com/blog/stans-world-its-prediction-time-or-it

Linear Regression

- Regression problems pop up whenever we want to predict a numerical value.
  - Predicting prices (of homes, stocks, etc.)
  - Predicting length of stay (for patients in the hospital)
  - Demand forecasting (for retail sales)

- Not every prediction problem is a classic regression problem.

- In classification problems, the goal is to predict membership among a set of categories.

https://towardsdatascience.com/machine-learning-classifiers-a5cc4e1b0623

Basic Elements of Linear Regression

- Linear regression flows from a few simple assumptions:
  - The relationship between the independent variables $\boldsymbol{x}$ and the dependent variable y is linear, i.e., that y can be expressed as a weighted sum of the elements in $\boldsymbol{x}$, given some noise on the observations.
  - Assume that any noise is well-behaved (following a Gaussian distribution).

- To develop a model for predicting house prices, we would need to get a dataset consisting of sales for which we know
  the sale price, area, and age for each home.
  - The dataset is called a *training dataset* or *training set.*
  - Each row (here the data corresponding to one sale) is called an *example* (or *data point, data instance, sample*).
  - The thing we are trying to predict (price) is called a *label* (or *target*).
  - The independent variables (age and area) upon which the predictions are based are called *features* (or *covariates*).

- We will use n to denote the number of examples in our dataset. We index the data examples by i, denoting each input
  as $\boldsymbol{x}^{(i)} = \left[x_1^{(i)}, x_2^{(i)}\right]^\top$ and the corresponding label as $y^{(i)}$.

Basic Elements of Linear Regression

Linear Model

Loss Function

Analytic Solution

Minibatch Stochastic Gradient Descent

Making Predictions with the Learned Model

Basic Elements of Linear Regression

Linear Model | Loss Function | Analytic Solution | Minibatch Stochastic Gradient Descent | Making Predictions with the Learned Model

- The linearity assumption says that the target (price) can be expressed as a weighted sum of the features (area and age):

$$price = w_{area} \cdot area + w_{age} \cdot age + b \tag{3.1.1}$$

w_{area} and w_{age} are called weights, and b is called a bias (also called an offset or intercept).

- The weights determine the influence of each feature on our prediction.
- The bias just says what value the predicted price should take when all of the features take value 0.
- (3.1.1) is an *affine transformation* of input features, which is characterized by a *linear transformation* of features via weighted sum, combined with a *translation* via the added bias.

Basic Elements of Linear Regression

Linear Model | Loss Function | Analytic Solution | Minibatch Stochastic Gradient Descent | Making Predictions with the Learned Model

- Given a dataset, our goal is to choose the weights $\boldsymbol{w}$ and the bias b such that on average, the predictions made according to our model best fit the true prices observed in the data.
- Models whose output prediction is determined by the affine transformation of input features are *linear models*
  - The affine transformation is specified by the chosen weights and bias.
- In machine learning, we usually work with high-dimensional datasets.
- When our inputs consist of d features, we express our prediction $\hat{y}$ (the "hat" symbol denotes estimates) as

$$\hat{y} = w_1 x_1 + \cdots + w_d x_d + b \tag{3.1.2}$$

Basic Elements of Linear Regression

Linear Model | Loss Function | Analytic Solution | Minibatch Stochastic Gradient Descent | Making Predictions with the Learned Model

- Collecting all features into a vector $\boldsymbol{x} \in R^d$ and all weights into a vector $\boldsymbol{w} \in R^d$, we can express our model using a dot product:

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + b \tag{3.1.3}$$

the vector $\boldsymbol{x}$ corresponds to features of a single data example.

- We refer to features of our entire dataset of n examples via the design matrix $\boldsymbol{X} \in R^{n \times d}$.
  - Here, $\boldsymbol{X}$ contains one row for every example and one column for every feature.
- For a collection of features $\boldsymbol{X}$, the predictions $\hat{y} \in R^n$ can be expressed via the matrix-vector product:

$$\hat{y} = \boldsymbol{X}\boldsymbol{w} + b \tag{3.1.4}$$

where broadcasting (see Section 2.1.3) is applied during the summation.

Basic Elements of Linear Regression

Linear Model | Loss Function | Analytic Solution | Minibatch Stochastic Gradient Descent | Making Predictions with the Learned Model

- Given features of a training dataset $\boldsymbol{X}$ and corresponding (known) labels y, the goal of linear regression is to find the weight vector $\boldsymbol{w}$ and the bias term b that given features of a new data example sampled from the same distribution as $\boldsymbol{X}$, the new example's label will (in expectation) be predicted with the lowest error.

- We **would not** expect to find a real-world dataset of n examples where $y^{(i)}$ exactly equals $\boldsymbol{w}^\top \boldsymbol{x}^{(i)} + b$ for all $1 \leq i \leq n$
  - Thus, even when we are confident that the underlying relationship is linear, we will incorporate a noise term to account for such errors.

- Before searching for the best parameters (or model parameters) $\boldsymbol{w}$ and b, we will need two more things:
  1. A quality measure for some given model.
  2. A procedure for updating the model to improve its quality.

Basic Elements of Linear Regression

- To think about how to *fit* data with our model, we need to determine a measure of *fitness*.
- The loss function quantifies the distance between the *real* and *predicted* value of the target.
  - The loss will be a non-negative number where smaller values are better.
  - Perfect predictions incur a loss of 0.
- The most popular loss function in regression problems is the squared error:

$$l^{(i)}(\boldsymbol{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 \tag{3.1.5}$$

$\hat{y}^{(i)}$ is the predicted label, $y^{(i)}$ is the corresponding true label for the i example.

- The constant $\frac{1}{2}$ makes no difference but will prove notationally convenient, canceling out when we take the derivative of the loss.

Basic Elements of Linear Regression

Linear Model | Loss Function | Analytic Solution | Minibatch Stochastic Gradient Descent | Making Predictions with the Learned Model

- The empirical error is only a function of the model parameters.
- Consider the example below where we plot a regression problem for a one-dimensional case as shown in Fig. 3.1.1.

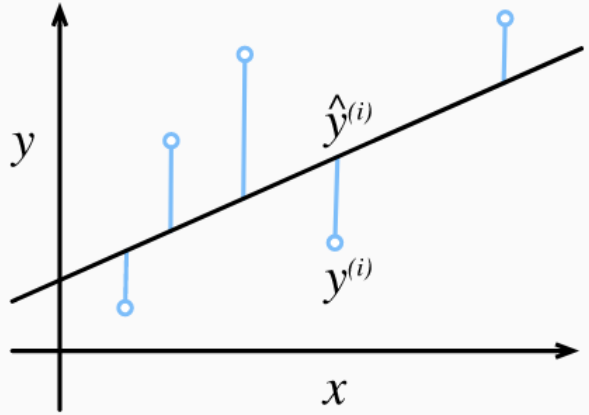


Fig. 3.1.1 Fit data with a linear model.

- Note that large differences between estimates $\hat{y}^{(i)}$ and observations $y^{(i)}$ lead to even larger contributions to the loss, due to the quadratic dependence.

Basic Elements of Linear Regression

- To measure the quality of a model on the entire dataset of n examples, we average (or equivalently, sum) the losses on the training set:

$$L(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\left(\boldsymbol{w}^\top x^{(i)} + b - y^{(i)}\right)^2 \tag{3.1.6}$$

- When training the model, we want to find parameters $(\boldsymbol{w}^*, b^*)$ that minimize the total loss across all training examples:

$$\boldsymbol{w}^*, b^* = argmin_{\boldsymbol{w},b}\, L(\boldsymbol{w}, b) \tag{3.1.7}$$

Basic Elements of Linear Regression

Linear Model | Loss Function | **Analytic Solution** | Minibatch Stochastic Gradient Descent | Making Predictions with the Learned Model

- Linear regression can be solved analytically by applying a simple formula:
  - Subsume the bias b into the parameter $\boldsymbol{w}$ by appending a column to the design matrix consisting of all ones.
  - Then our prediction problem is to minimize $\| y - \boldsymbol{X}\boldsymbol{w} \|^2$.
  - Take the loss surface to be the minimum of the loss over the entire domain.
  - Taking the derivative of the loss with respect to $\boldsymbol{w}$ and setting it equal to zero yields the analytic (closed-form) solution:

$$\partial_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 = 2\mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y}) = 0 \text{ and hence } \mathbf{X}^\top\mathbf{y} = \mathbf{X}^\top\mathbf{X}\mathbf{w}.$$

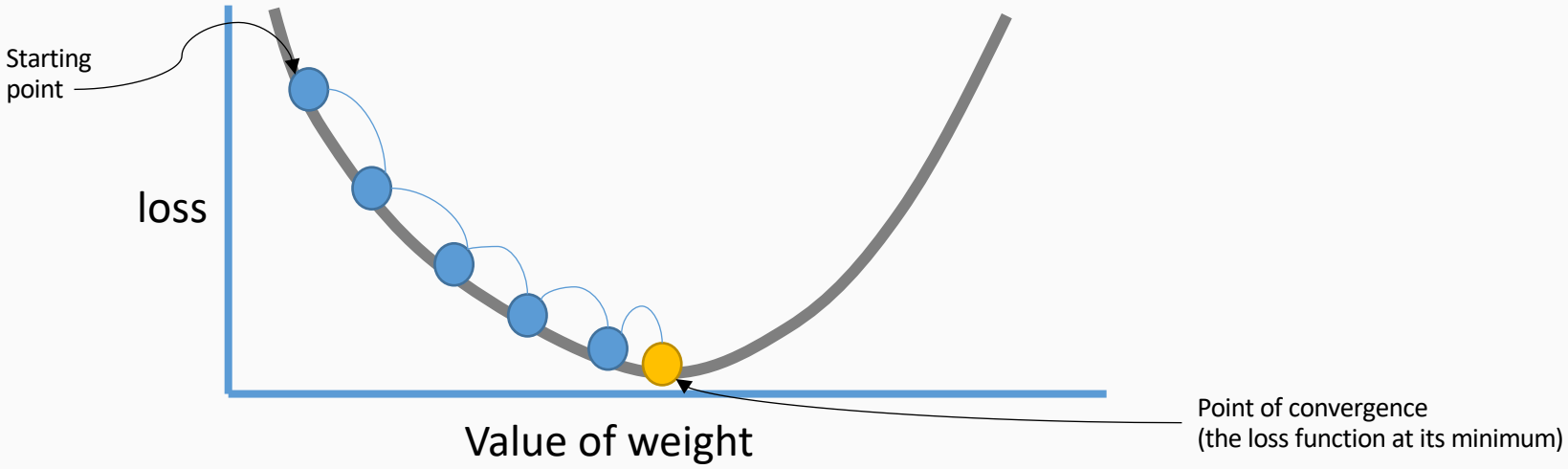
$$\boldsymbol{w}^* = (\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top y \tag{3.1.8}$$

- The requirement of an analytic solution is so restrictive that it would exclude all of deep learning.
  - Simple problems like linear regression may admit analytic solutions but, you should not get used to such good fortune.

* subsume: 포함하다

Basic Elements of Linear Regression

- In cases where we cannot solve the models analytically, it turns out that we can still train models effectively in practice.
- The key technique for optimizing nearly any deep learning model is called *gradient descents*.
  - Gradient descent iteratively reduces the error by updating the parameters in the direction that incrementally lowers the loss function.



Basic Elements of Linear Regression

- The most naive application of gradient descent consists of taking the derivative of the loss function, which is an average of the losses computed on every single example in the dataset.
  - This is extremely slow: we must pass over the entire dataset before making a single update.
  - Thus, we will sample a random minibatch of examples every time we need to compute the update, this variant called *minibatch stochastic gradient descent*.

- In each iteration:
  1. We first randomly sample a minibatch **B** consisting of a fixed number of training examples.
  2. We then compute the derivative (gradient) of the average loss on the minibatch with regard to the model parameters.
  3. Finally, we multiply the gradient by a predetermined positive value **η** and subtract the resulting term from the current parameter values.

Basic Elements of Linear Regression

- We can express the update mathematically as follows (∂ denotes the partial derivative):

$$(\boldsymbol{w}, b) \leftarrow (\boldsymbol{w}, b) - \frac{\eta}{|B|}\sum_{i \in B} \partial_{(\boldsymbol{w},b)} l^{(i)}(\boldsymbol{w}, b)\,.$$

$\boldsymbol{w}$ is the weights vector,
b is the bias,
η is predetermined positive value,
and the term "$\partial_{(\boldsymbol{w},b)} l^{(i)}(\boldsymbol{w}, b)$" means the partial derivative of the loss of ith element.

- To summarize the steps of the algorithm:

Randomly initialize the values of the model parameters → Iteratively sample random minibatches from the data → Update the parameters in the direction of the negative gradient

Basic Elements of Linear Regression

- For quadratic losses and affine transformations, we can write this out explicitly as follows:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \frac{\eta}{|B|}\sum_{i\in B} \partial_{\boldsymbol{w}} l^{(i)}(\boldsymbol{w}, b) = \boldsymbol{w} - \frac{\eta}{|B|}\sum_{i\in B} \boldsymbol{x}^{(i)}\left(\boldsymbol{w}^\top \boldsymbol{x}^{(i)} + b - y^{(i)}\right),$$

$$b \leftarrow b - \frac{\eta}{|B|}\sum_{i\in B} \partial_b l^{(i)}(\boldsymbol{w}, b) = b - \frac{\eta}{|B|}\sum_{i\in B}\left(\boldsymbol{w}^\top \boldsymbol{x}^{(i)} + b - y^{(i)}\right). \tag{3.1.10}$$

Note that $\boldsymbol{w}$ and $\boldsymbol{x}$ are vectors.
The set cardinality $|\boldsymbol{B}|$ represents the number of examples in each minibatch (the batch size).
η denotes the *learning rate*.

- The values of the batch size and learning rate are manually pre-specified and not typically learned through model training.
  - These parameters that are tunable but not updated in the training loop are called ***hyperparameters***.

Basic Elements of Linear Regression

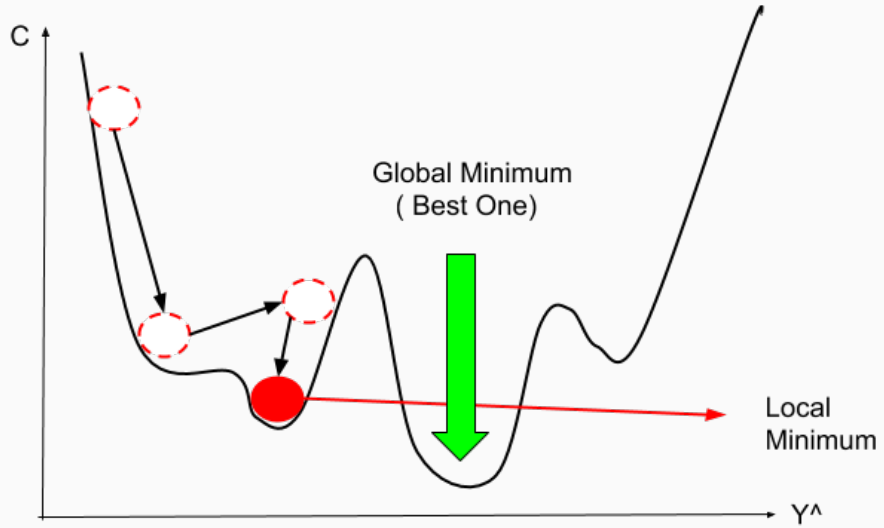
- *Hyperparameter tuning* is the process by which hyperparameters are chosen, and typically requires that we adjust them based on the results of the training loop as assessed on a separate *validation dataset*.

- After training for some predetermined number of iterations (or until some other stopping criteria are met), we record the estimated model parameters, denoted $\hat{\boldsymbol{w}}, \hat{b}$.
  - If our function is truly linear and noiseless, these parameters will not be the exact minimizers of the loss because, although the algorithm converges slowly towards the minimizers it cannot achieve it exactly in a finite number of steps.

Basic Elements of Linear Regression

Linear Model | Loss Function | Analytic Solution | **Minibatch Stochastic Gradient Descent** | Making Predictions with the Learned Model

- Linear regression happens to be a learning problem where there is only one minimum over the entire domain.
  - For more complicated models, like deep networks, the loss surfaces contain many minima.
- Deep learning practitioners seldom struggle to find parameters that minimize the loss on training sets.
- The more formidable task is to find parameters that will achieve low loss on data that we have not seen before.
  - A challenge called ***generalization***.
- Given the learned linear regression model $\hat{\boldsymbol{w}}^\top \boldsymbol{x} + \hat{b}$, we can estimate the price of a new house given its area x_1 and age x_2.
  - Estimating targets given features is commonly called ***prediction*** or ***inference***.



https://www.mltut.com/stochastic-gradient-descent-a-super-easy-complete-guide/

- seldom: 좀처럼 … 않는
- formidable: 만만치 않은

Vectorization for Speed

- When training our models, we typically want to process whole minibatches of examples simultaneously.
  - Doing this efficiently requires that we vectorize the calculations and leverage fast linear algebra libraries.

```
%matplotlib inline
import math
import time
Import numpy as np
Import torch
from d2l import torch as d2l
```

- We consider two methods for adding vectors.
  - To start we instantiate two 10000-dimensional vectors containing all ones.
  - In one method we will loop over the vectors with a Python for-loop.
  - In the other method we will rely on a single call to +.

```
n = 10000
a = np.ones(n)
b = np.ones(n)
```

Vectorization for Speed

- Now we can benchmark the workloads.
  - First, we add them, one coordinate at a time, using a for-loop.

```
c = np.zeros(n)
t = time.time()
for i in range(n):
    c[i] = a[i] + b[i]
f'{time.time() - t:.5f} sec'
```

  - Alternatively, we rely on the reloaded + operator to compute the elementwise sum.

```
t = time.time()
d = a + b
f'{time.time() - t:0.5f} sec'
```

- The second method is dramatically faster than the first.
  - Vectorizing code often yields order-of-magnitude speedups.

The Normal Distribution and Squared Loss

- There is a strong connection between the normal distribution (Gaussian) and linear regression.
- The probability density of a normal distribution with mean μ and variance σ^2 (standard deviation σ) is given as:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{1}{2\sigma^2}(x-\mu)^2) \tag{3.1.11}$$

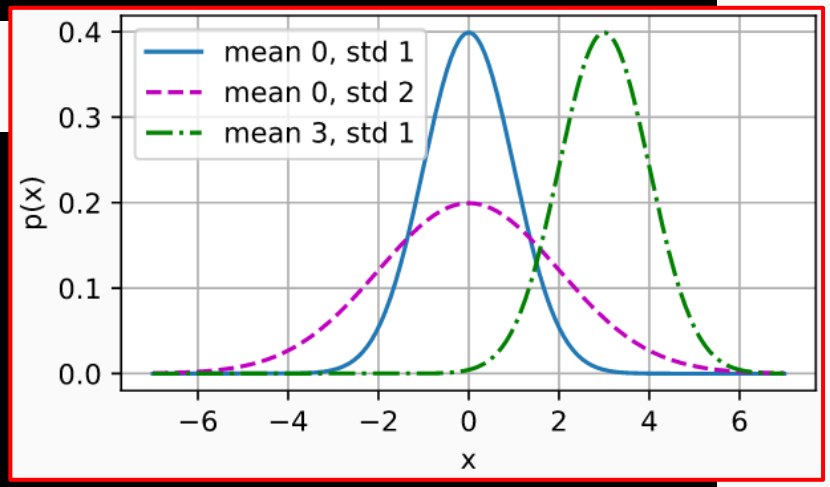
  - A Python function to compute the normal distribution:

```
def normal(x, mu, sigma):
  p = 1 / math.sqrt(2 * math.pi * sigma**2)
  return p * np.exp(-0.5 * (x-mu)**2 / sigma**2)
```

  - Visualize the normal distributions:

```
# Use numpy again for visualization
x = np.arange(-7, 7, 0.01)

# Mean and standard deviation pairs
params = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in params],
         xlabel='x',ylabel='p(x)', figsize=(4.5, 2.5),
         legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])
```



The Normal Distribution and Squared Loss

- To motivate linear regression with the squared loss function, assume that observations arise from noisy observations, where the noise is normally distributed as follows:

$$y = \boldsymbol{w}^\top \boldsymbol{x} + b + \epsilon \; where \; \epsilon \sim N(0, \sigma^2) \tag{3.1.12}$$

- Thus, we write out the likelihood of seeing a particular y for a given $\boldsymbol{x}$ via

$$P(y \mid \boldsymbol{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{1}{2\sigma^2}(y - \boldsymbol{w}^\top \boldsymbol{x} - b)^2) \tag{3.1.13}$$

- Now, according to the principle of maximum likelihood, the best values of parameters w and b are those that maximize the likelihood of the entire dataset:

$$P(\boldsymbol{y} \mid \boldsymbol{X}) = \prod_{i=1}^{n} = p(y^{(i)} | x^{(i)}) \tag{3.1.14}$$

- Estimators chosen according to the principle of maximum likelihood are called *maximum likelihood estimators*.

The Normal Distribution and Squared Loss

- Maximizing the product of many exponential functions is difficult.
  - So, we maximize the log of the likelihood instead.
  - For historical reasons, optimizations are expressed as minimization rather than maximization.

- So, We minimize the negative log-likelihood $-\log P(\boldsymbol{y} \mid \boldsymbol{X})$:

$$-\log P(\boldsymbol{y} \mid \boldsymbol{X}) = \sum_{i=1}^{n} \frac{1}{2} log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\left(y^{(i)} - w^\top x^{(i)} - b\right)^2 \tag{3.1.15}$$

  - We assume that σ is some fixed constant.
  - Thus we can ignore the first term because it does not depend on $\boldsymbol{w}$ or b.
  - Now the second term is identical to the squared error loss , $l^{(i)}(\boldsymbol{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$, except for the multiplicative constant $\frac{1}{\sigma^2}$.

- Minimizing the mean squared error is equivalent to maximum likelihood estimation of a linear model under the assumption of additive Gaussian noise.
  - The solution does not depend on σ.

From Linear Regression to Deep Networks

- We think of the linear model as a neural network by expressing it in the language of neural networks.
  - Neural networks cover a much richer family of models.
- We depict our linear regression model as a neural network.
  - These diagrams highlight the connectivity pattern such as how each input is connected to the output, but not the values taken by the weights or biases.
- For the neural network shown in Fig. 3.1.2,
  - The inputs are $x_1, \ldots, x_d$, so the number of inputs (or feature dimensionality) in the input layer is d.
  - The output of the network is o_1, so the number of outputs is 1.
  - The inputs are all given and there is just a single *computed neuron*.

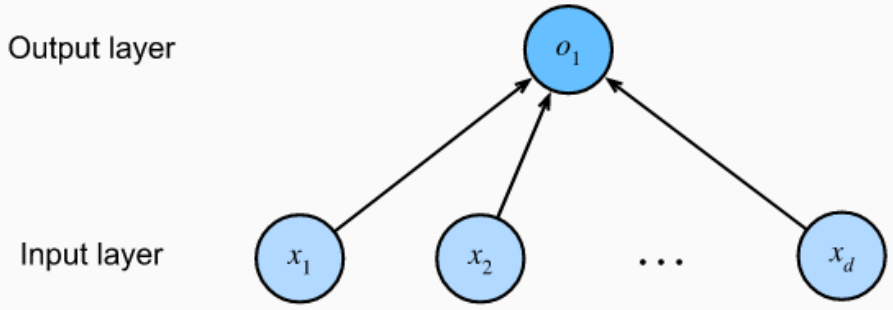


Fig. 3.1.2 Linear regression is a single-layer neural network.

- We do not consider the input layer when counting layers.
  - The number of layers for the neural network in Fig. 3.1.2 is 1.
  - We can think of linear regression models as neural networks consisting of just a single artificial neuron, or as single-layer neural networks.
  - Since for linear regression, every input is connected to every output, we can regard this transformation (the output layer in Fig. 3.1.2) as a *fully-connected layer* or *dense layer*.

From Linear Regression to Deep Networks

- Linear regression (invented in 1795) predates computational neuroscience.
  - Warren McCulloch and Walter Pitts began to develop models of artificial neurons.

- Consider the cartoonish picture of a biological neuron in Fig. 3.1.3
  - Dendrites = input terminals,
  - Nucleus = CPU,
  - Axon = output wire,
  - Axon terminals = output terminals, enable connections to other neurons via synapses.

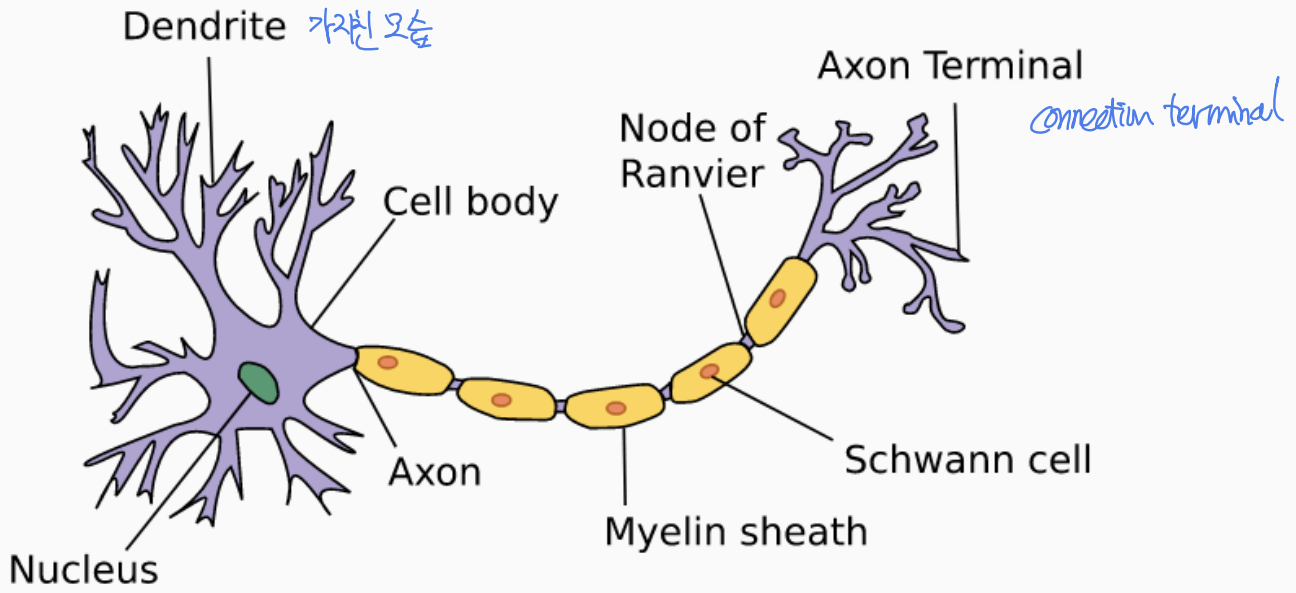
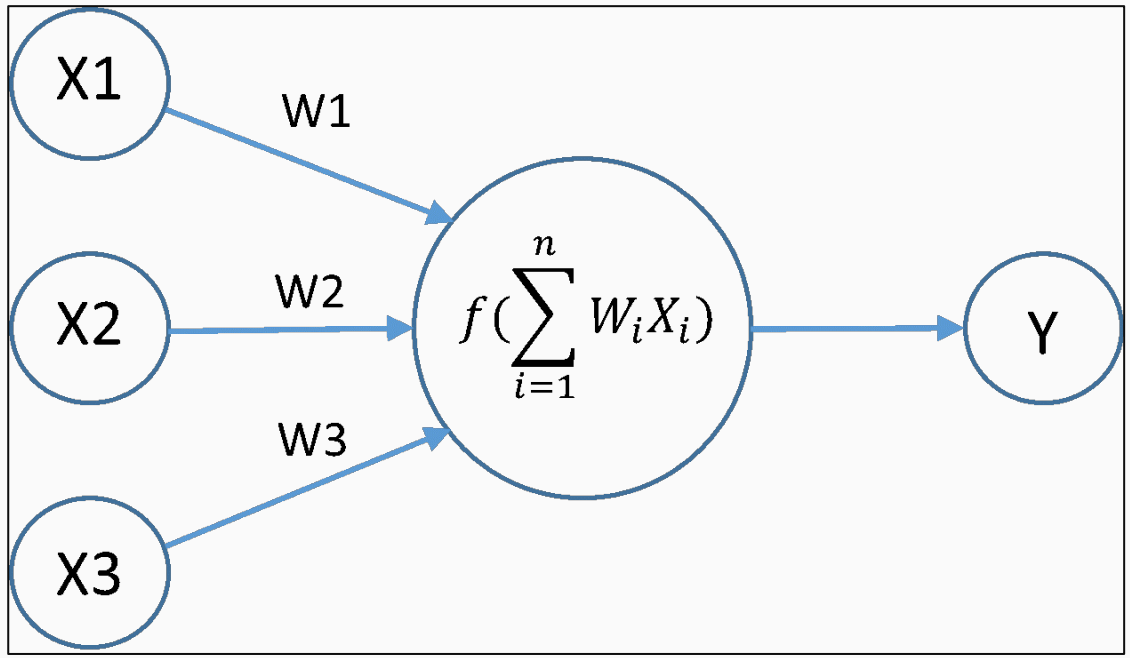


Fig. 3.1.3 The real neuron.

From Linear Regression to Deep Networks

- Information x_i arriving from other neurons (or environmental sensors such as the retina) is received in the dendrites.
  - That information is weighted by synaptic weights w_i determining the effect of the inputs (e.g., activation or inhibition via the product $x_i w_i$).
  - The weighted inputs arriving from multiple sources are aggregated in the nucleus as a weighted sum $y = \sum_i x_i w_i + b$.
  - This information is sent for further processing in the axon y, typically after nonlinear processing via $\sigma(y)$.
  - From there it either reaches its destination (e.g., a muscle) or is fed into another neuron via its dendrites.



https://medium.com/@abhismatrix/neural-network-model-training-tricks-61254a2a1f6b

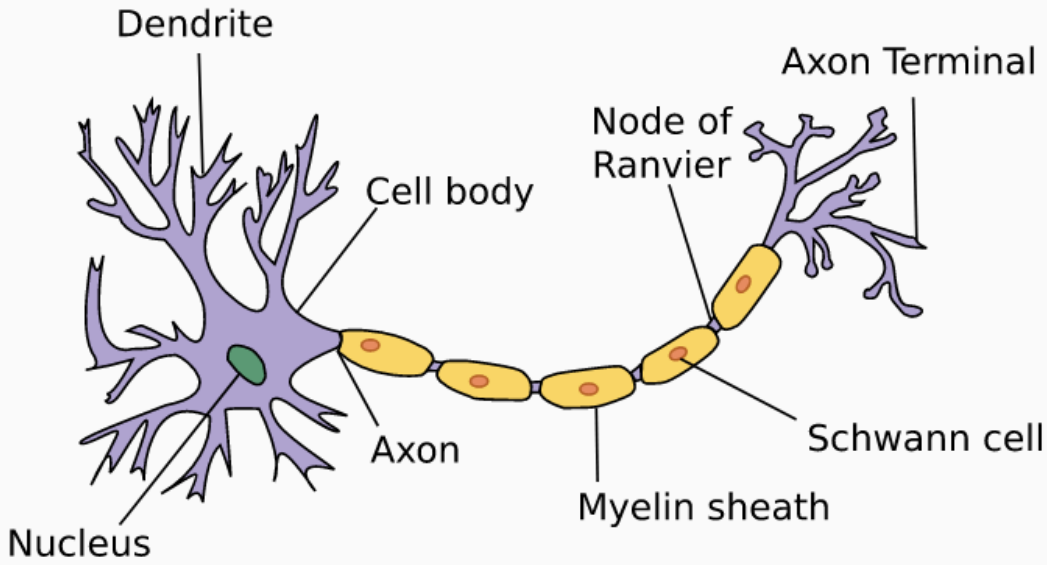
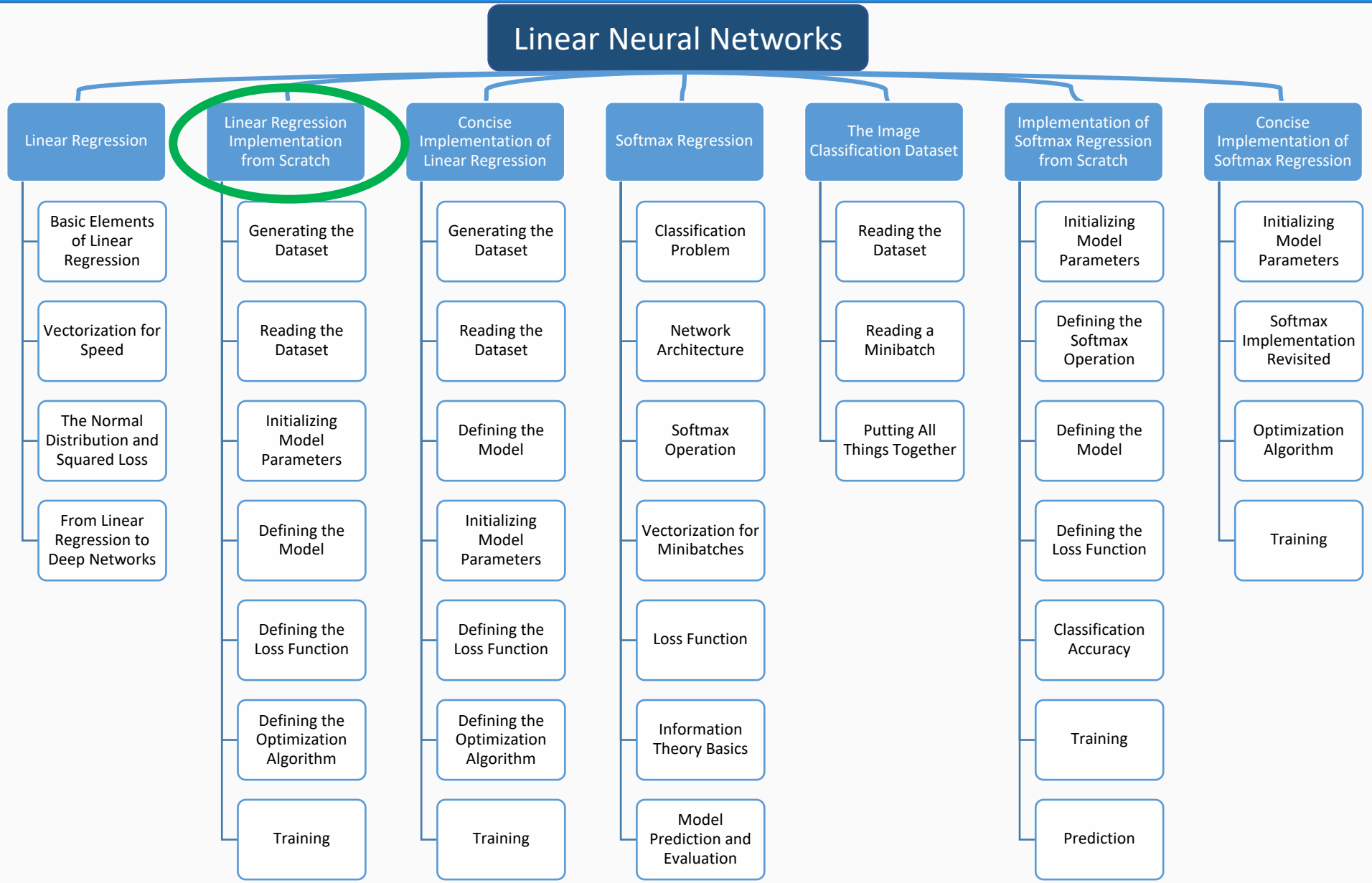


Fig. 3.1.3 The real neuron.

Summary

- Key ingredients in a machine learning model are training data, a loss function, an optimization algorithm, and quite obviously, the model itself.
- Vectorizing makes everything better (mostly math) and faster (mostly code).
- Minimizing an objective function and performing maximum likelihood estimation can mean the same thing.
- Linear regression models are neural networks, too.

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

Linear Regression Implementation from Scratch

- In this section, we will implement the entire method from scratch:
  - Data pipeline
  - Model
  - Loss function
  - Minibatch stochastic gradient descent optimizer.

- We will rely only on tensors and auto differentiation.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, np, npx
import random
npx.set_np()
```

Generating the Dataset

- We will construct an artificial dataset according to a linear model with additive noise.

- We generate a dataset containing 1000 examples, each consisting of 2 features sampled from a standard normal distribution.
  - Our synthetic dataset will be a matrix $\boldsymbol{X} \in R^{1000\times 2}$.

- The true parameters generating our dataset will be $\boldsymbol{w} = [2, -3.4]^\top$ and $b = 4.2$, and our synthetic labels will be assigned according to the following linear model with the noise term ϵ:

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w} + b + \epsilon. \tag{3.2.1}$$

  - Think of ϵ as capturing potential measurement errors on the features and labels.
  - Assume that ϵ obeys a normal distribution with mean of 0.
  - We will set its standard deviation to 0.01.

Generating the Dataset

- The following code generates our synthetic dataset.

```
def synthetic_data(w, b, num_examples):  #@save
    """Generate y = Xw + b + noise."""
    X = np.random.normal(0, 1, (num_examples, len(w)))
    y = np.dot(X, w) + b
    y += np.random.normal(0, 0.01, y.shape)
    return X, y.reshape((-1, 1))

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

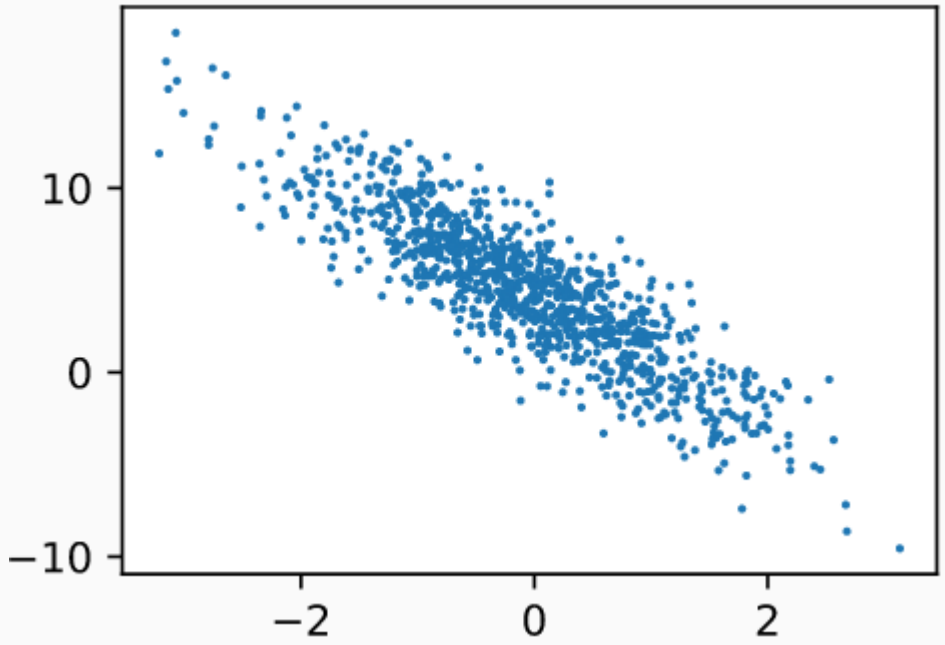
- Each row in features consists of a 2-dimensional data example.
- Each row in labels consists of a 1-dimensional label value (a scalar).

```
print('features:', features[0],'\nlabel:', labels[0])
```

Generating the Dataset

- Generate a scatter plot using the second feature features[:, 1] and labels.

```
d2l.set_figsize()
# The semicolon is for displaying the plot only
d2l.plt.scatter(d2l.numpy(features[:, 1]), d2l.numpy(labels), 1);
```



Reading the Dataset

- Training models consists of:
  - Making multiple passes over the dataset,
  - Grabbing one minibatch of examples at a time,
  - Using them to update our model.

- The training process is so fundamental to machine learning algorithms,
  - It is worth defining a utility function to shuffle the dataset and access it in minibatches.

- The $data\_iter$ function takes a batch size, a matrix of features, and a vector of labels, yielding minibatches of the size $batch\_size$.
  - Each minibatch consists of a tuple of features and labels.

```
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # The examples are read at random, in no particular order
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        batch_indices = np.array(
            indices[i: min(i + batch_size, num_examples)])
        yield features[batch_indices], labels[batch_indices]
```

Reading the Dataset

- We want to use reasonably sized minibatches to take advantage of parallelizing operations using GPU hardware.
  - Each example can be fed through our models in parallel and the gradient of the loss function for each example can also be taken in parallel.

- Let us read and print the first small batch of data examples.

```
batch_size = 10

for X, y in data_iter(batch_size, features, labels):
    print(X, '\n', y)
    break
```

- Notes:
  - The shape of the features in each minibatch gives the minibatch size and the number of input features.
  - Minibatch of labels will have a shape given by $batch\_size$.
  - As we run the iteration, we obtain distinct minibatches successively until the entire dataset has been exhausted.
  - The implemented iteration above is inefficient, it requires loading all the data in memory and performing lots of random memory access.
  - The built-in iterators implemented in a deep learning framework are more efficient and they can deal with both data stored in files and data fed via data streams.

Initializing Model Parameters

- In the following code, we initialize weights by sampling random numbers from a normal distribution with mean 0 and a standard deviation of 0.01, and setting the bias to 0.

```
w = np.random.normal(0, 0.01, (2, 1))
b = np.zeros(1)
w.attach_grad()
b.attach_grad()
```

- After initializing our parameters, our next task is to update them until they fit our data sufficiently well.
  - Each update requires taking the gradient of our loss function with respect to the parameters.
  - Given this gradient, we can update each parameter in the direction that may reduce the loss.
  - We will use automatic differentiation.

Defining the Model

- Next, we must define our model, relating its inputs and parameters to its outputs.

- To calculate the output of the linear model,
  - Take the matrix-vector dot product of the input features $\boldsymbol{X}$ and the model weights $\boldsymbol{w}$,
  - Add the offset b to each example.

- Note that $\boldsymbol{Xw}$ is a vector and b is a scalar.
  - So the broadcasting mechanism is applied (the scalar is added to each component of the vector).

```
def linreg(X, w, b):  #@save
    """The linear regression model."""
    return np.dot(X, w) + b
```

Defining the Loss Function

- Updating our model requires taking the gradient of our loss function.

- We will use the squared loss function.
  - We need to transform the true value y into the predicted value's shape y_hat.
  - The result returned by the following function will also have the same shape as y_hat.

```
def squared_loss(y_hat, y):  #@save
    """Squared loss."""
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

Defining the Optimization Algorithm

- At each step, using one minibatch randomly drawn from our dataset,
  - We will estimate the gradient of the loss with respect to our parameters.
  - Next, we will update our parameters in the direction that may reduce the loss.

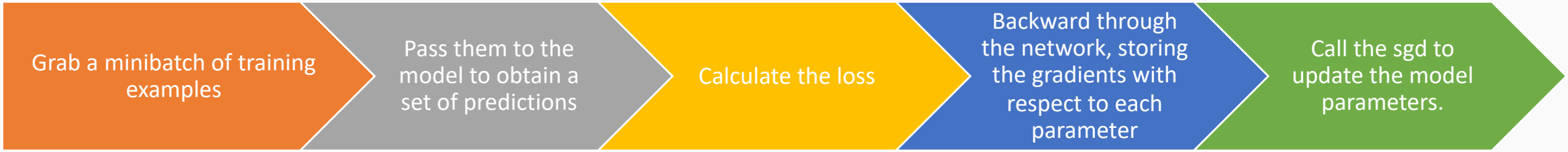
- The following code applies the minibatch stochastic gradient descent update, given a set of parameters, a learning rate (size of the update step, lr), and a batch size.

```
def sgd(params, lr, batch_size):  #@save
    """Minibatch stochastic gradient descent."""
    for param in params:
        param[:] = param - lr * param.grad / batch_size
```

- The loss is calculated as a sum over the minibatch of examples.
  - We normalize our step size by the batch size (batch_size), so that the magnitude of a typical step size does not depend heavily on our choice of the batch size.

Training

- We will execute the following loop:
  - Initialize parameters $(\boldsymbol{w}, b)$
  - Repeat until done
    - Compute gradient $\boldsymbol{g} \leftarrow \partial(\boldsymbol{w}, b) \frac{1}{|B|} \sum_{i \in B} l(x^{(i)}, y^{(i)}, \boldsymbol{w}, b)$
    - Update parameters $(\boldsymbol{w}, b) \leftarrow (\boldsymbol{w}, b) - \eta \boldsymbol{g}$



- In each *epoch*, iterate through the entire dataset (using the $data\_iter$ function) once passing through every example in the training dataset.
  - Assuming that the number of examples is divisible by the batch size.

- The number of epochs num_epochs and the learning rate lr are both hyperparameters, which we set to 3 and 0.03, respectively.
  - Setting hyperparameters is tricky and requires some adjustment by trial and error.

Training

```
lr = 0.03
num_epochs = 3
net = linreg
loss = squared_loss

for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, features, labels):
        with autograd.record():
            l = loss(net(X, w, b), y)  # Minibatch loss in `X` and `y`
        # Because `l` has a shape (`batch_size`, 1) and is not a scalar
        # variable, the elements in `l` are added together to obtain a new
        # variable, on which gradients with respect to [`w`, `b`] are computed
        l.backward()
        sgd([w, b], lr, batch_size)  # Update parameters using their gradient
    train_l = loss(net(features, w, b), labels)
    print(f'epoch {epoch + 1}, loss {float(train_l.mean()):f}')
```

Training

- Because we synthesized the dataset ourselves, we know precisely what the true parameters are.

- Thus, we can evaluate our success in training by comparing the true parameters with those that we learned through our training loop.
  - They turn out to be very close to each other.

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

- Note that we should not take it for granted that we are able to recover the parameters perfectly.
  - We are less concerned with recovering true underlying parameters, and more concerned with parameters that lead to highly accurate prediction.

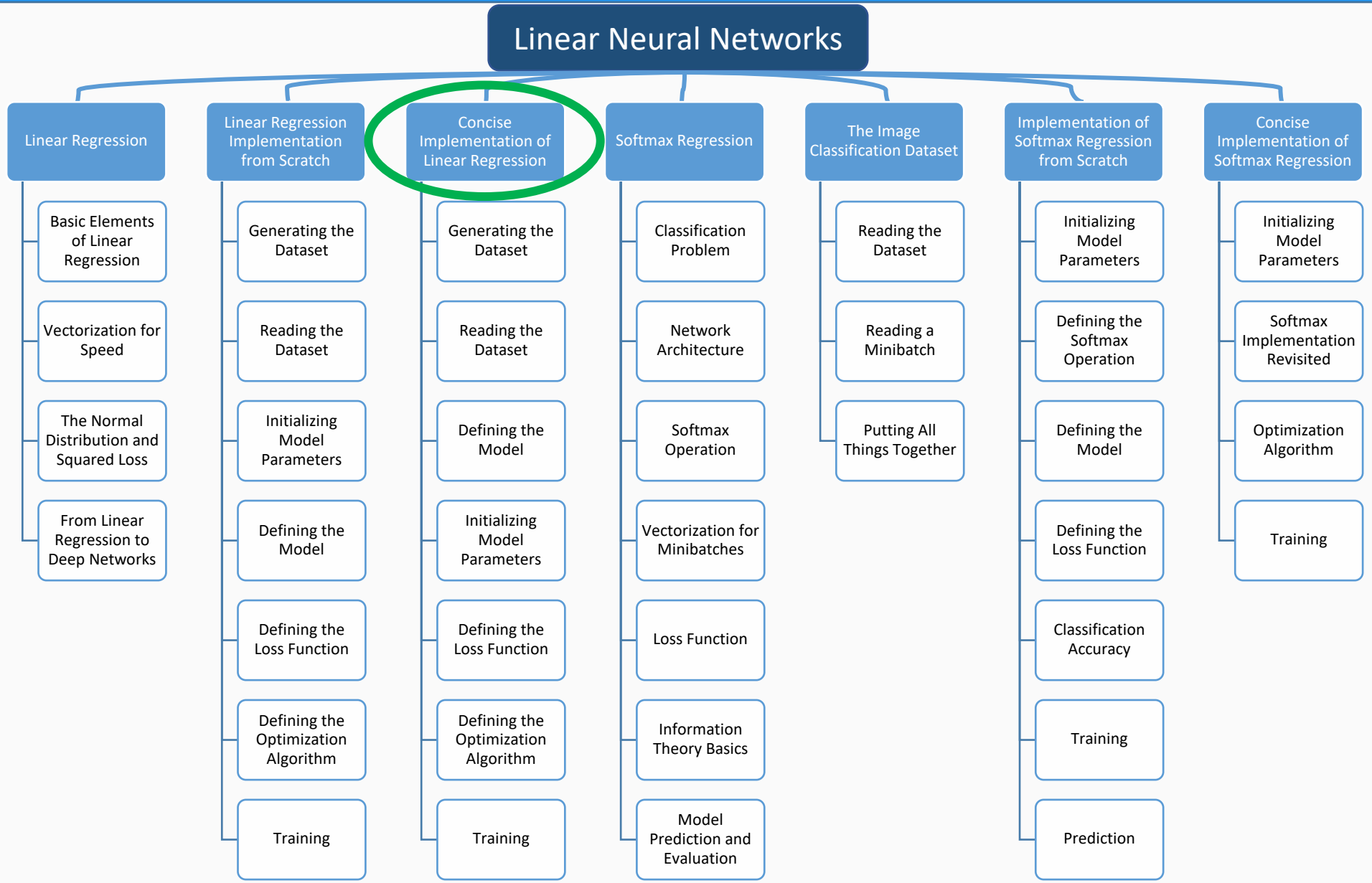
- Even on difficult optimization problems, stochastic gradient descent can often find remarkably good solutions.
  - Owing partly to the fact that, for deep networks, there exist many configurations of the parameters that lead to highly accurate prediction.

Summary

- We saw how a deep network can be implemented and optimized from scratch, using just tensors and auto differentiation, without any need for defining layers or fancy optimizers.

- This section only scratches the surface of what is possible. In the following sections, we will describe additional models based on the concepts that we have just introduced and learn how to implement them more concisely.

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

- In this section, we will show you how to implement the linear regression model concisely by using high-level APIs of deep learning frameworks.

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)
```

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.

- We will first define a model variable net, which will refer to an instance of the $Sequential$ class.
  - The $Sequential$ class defines a container for several layers that will be chained together.
  - A $Sequential$ instance passes its input through the first layer, then passing the output as the second layer's input and so forth.

- The layer in 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 Gluon, the fully-connected layer is defined in the $Dense$ class.
  - Since we only want to generate a single scalar output, we set that number to 1.
  - Gluon does not require us to specify the input shape for each layer.

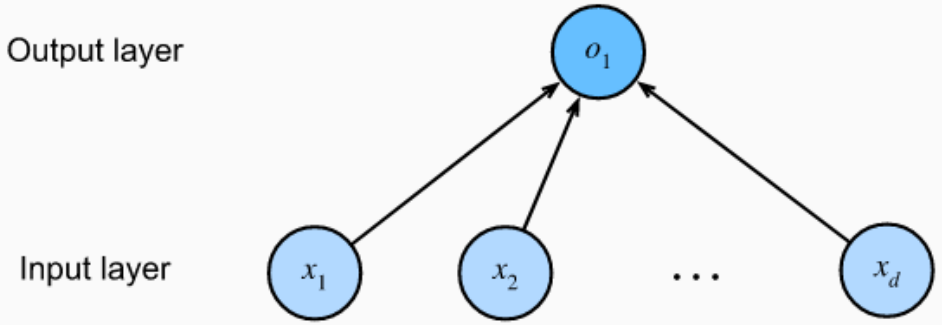


Fig. 3.1.2 Linear regression is a single-layer neural network.

```
# `nn` is an abbreviation for neural networks
from mxnet.gluon import nn
net = nn.Sequential()
net.add(nn.Dense(1))
```

Initializing Model Parameters

- Before using net, we initialize the model parameters, such as the weights and bias in the linear regression model.
  - Deep learning frameworks often have a predefined way to initialize the parameters.
  - We specify that each weight parameter should be randomly sampled from a normal distribution with mean 0 and standard deviation 0.01.
  - The bias parameter will be initialized to zero.

- We will import the $initializer$ module from MXNet.
  - Bias parameters are initialized to zero by default.

```
from mxnet import init
net.initialize(init.Normal(sigma=0.01))
```

- Note that we are initializing parameters for a network even though Gluon does not yet know how many dimensions the input will have! It might be 2 as in our example or it might be 2000.
  - Gluon lets us get away with this because behind the scene, the initialization is actually *deferred*.
  - The real initialization will take place only when we for the first time attempt to pass data through the network.

Defining the Loss Function

- In Gluon, the $loss$ module defines various loss functions.
  - In this example, we will use the Gluon implementation of squared loss ($L2Loss$).

```
loss = gluon.loss.L2Loss()
```

Defining the Optimization Algorithm

- Minibatch stochastic gradient descent is a standard tool for optimizing neural networks and thus Gluon supports it alongside a number of variations on this algorithm through its $Trainer$ class. We will specify:
  - Parameters to optimize over (obtainable from our model net via $net.collect\_params()$)
  - Optimization algorithm we wish to use (sgd)
  - A dictionary of hyperparameters required by our optimization algorithm ($learning\_rate$)

```
from mxnet import gluon
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.03})
```

Training

- For each minibatch, we go through the following ritual:
  - Generate predictions by calling $net(X)$ and calculate the loss l (the forward propagation).
  - Calculate gradients by running the backpropagation.
  - Update the model parameters by invoking our optimizer.

```
num_epochs = 3
for epoch in range(num_epochs):
    for X, y in data_iter:
        with autograd.record():
            l = loss(net(X), y)
        l.backward()
        trainer.step(batch_size)
    l = loss(net(features), labels)
    print(f'epoch {epoch + 1}, loss {l.mean().asnumpy():f}')
```

- We compare the model parameters learned by training and the actual parameters that generated our dataset.

```
w = net[0].weight.data()
print(f'error in estimating w: {true_w - w.reshape(true_w.shape)}')
b = net[0].bias.data()
print(f'error in estimating b: {true_b - b}')
```

Summary

- Using Gluon, we can implement models much more concisely.
- In Gluon, the $data$ module provides tools for data processing, the nn module defines a large number of neural network layers, and the $loss$ module defines many common loss functions.
- MXNet's module $initializer$ provides various methods for model parameter initialization.
- Dimensionality and storage are automatically inferred, but be careful not to attempt to access parameters before they have been initialized.