2장

수학적 구성요소

"시도해보지 않고는 누구도 자신이 얼마만큼 해낼 수 있는지 알지 못한다"
푸블릴리우스 시루스

Outline

- A first example of a neural network
- Tensors and tensor operations
- How neural networks learn via backpropagation and gradient descent

2.1 A first look at a neural network

label : 0 2 4 3

Figure 2.1 MNIST sample digits (28*28)

- classify grayscale images of handwritten digits (28 × 28 pixels) into their 10 categories (0 through 9)
- MNIST dataset - a set of 60,000 training images, plus 10,000 test images

Listing 2.1 Loading the MNIST dataset in Keras

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

2.1 A first look at a neural network

- Let’s look at the training data:

```
>>> train_images.shape (60000, 28, 28)
>>> len(train_labels) 60000
>>> train_labels
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

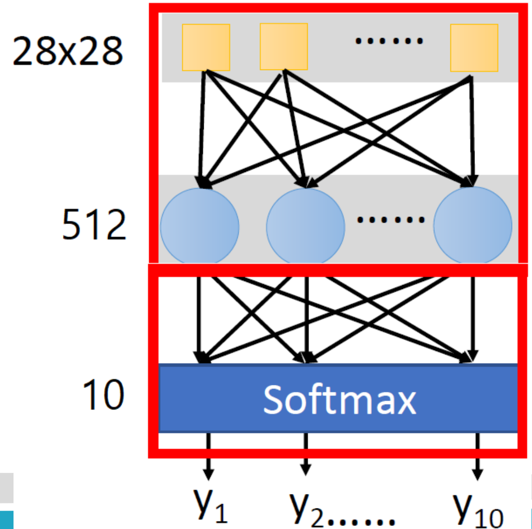
- And here’s the test data:

```
>>> test_images.shape (10000, 28, 28)
>>> len(test_labels) 10000
>>> test_labels
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

1. Network 구성:

Listing 2.2 The network architecture

```
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation='relu',
input_shape=(28 * 28,)))    # 784 개 input node
network.add(layers.Dense(10, activation='softmax'))
```



2. *compilation* step for training:

- *A loss function* —How the network will be able to measure its performance on the training data

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

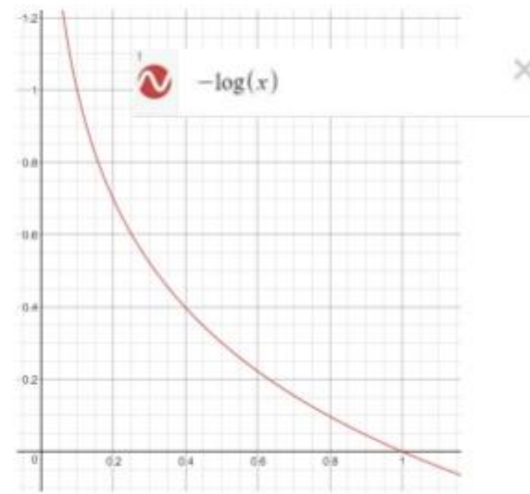
$$\mathrm{MAE} = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|$$

Cross Entropy Cost Function

$$D(\overline{Y}_i, Y_i) = -\sum Y_i \log \overline{Y}_i$$

$$\begin{bmatrix}\overline{Y}_A\\ \overline{Y}_B\\ \overline{Y}_C\end{bmatrix} = \begin{bmatrix}1\\0\\0\end{bmatrix} \qquad \begin{bmatrix}Y_A\\ Y_B\\ Y_C\end{bmatrix} = \begin{bmatrix}1\\0\\0\end{bmatrix}$$

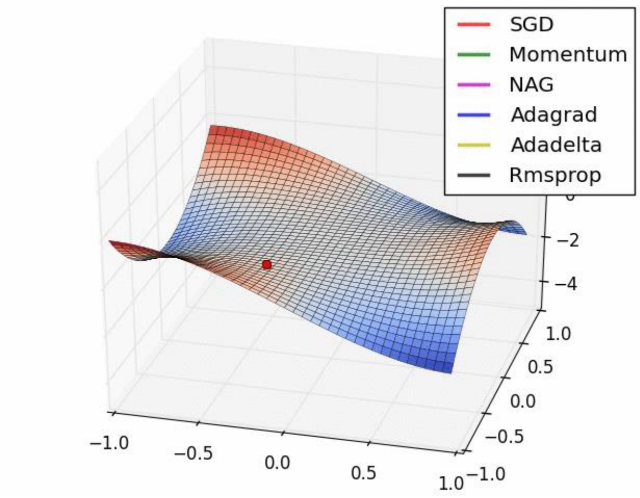
$$-\begin{bmatrix}1\\0\\0\end{bmatrix} \cdot \log\begin{bmatrix}1\\0\\0\end{bmatrix} = -\begin{bmatrix}1\\0\\0\end{bmatrix}\begin{bmatrix}0\\ \infty\\ \infty\end{bmatrix} = \begin{bmatrix}0\\0\\0\end{bmatrix} = 0$$



- *An optimizer* —The mechanism through which the network will update itself based on the data it sees and its loss function.
- *Metrics to monitor during training and testing*—Here, we'll only care about accuracy

Listing 2.3 The compilation step

```
network.compile(optimizer='rmsprop',
    loss='categorical_crossentropy',
    metrics=['accuracy'])
```



3. Data Preparation for training:

- scaling - `[0, 255]` interval → `[0, 1]` interval
- `(60000, 28, 28)` shape → `(60000, 28*28)` shape

Listing 2.4 Preparing the image data

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

4. Categorically encode the labels for training:
- One-Hot-Encoding으로 변환

$5 \rightarrow$ [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], …

Listing 2.5 Preparing the labels

```
from keras.utils import to_categorical
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

5. fit method—we fit the model to its training data :

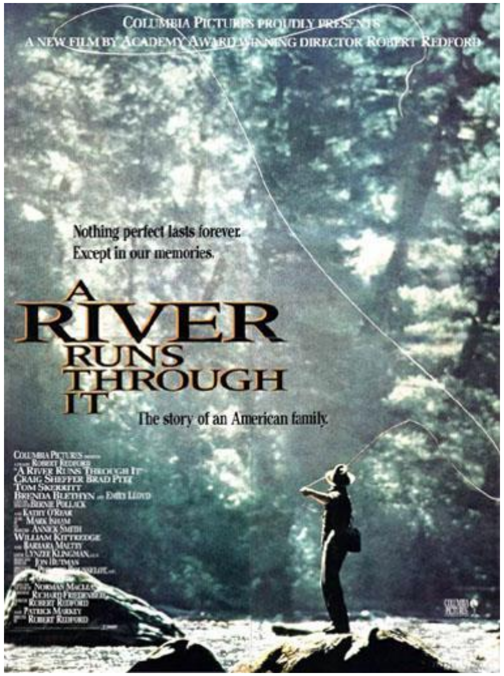
```
>>> network.fit(train_images, train_labels, epochs=5,
batch_size=128)
Epoch 1/5
60000/60000 [==============================] - 9s - loss:
0.2524 - acc: 0.9273
Epoch 2/5
60000/60000 [==============================] - ETA: 1s - loss:
0.1035 - acc: 0.9692

…

Epoch 5/5
60000/60000 [==============================] - ETA: 12s - loss:
0.0935 - acc: 0.9892
```

6. Test data : a bit lower than the training set accuracy

```
>>> test_loss, test_acc = network.evaluate(test_images, test_labels)
>>> print('test_acc:', test_acc)
test_acc: 0.9785
```

- *overfitting*: the fact that machine-learning models tend to perform worse on new data than on their training data.

2.2 Data representations for neural networks

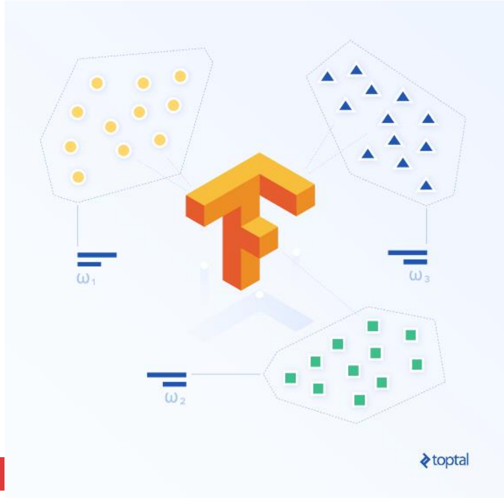


- A River Runs Through It - 아름다운 자연을 배경과 함께 강물의 흐름을 따라 고기를 잡는 플라이 낚시와 가족 간의 사랑과 아픔 그리고 삶을 은유

- 딥러닝과 대단위의 정보와 지식, 이를 통한 추론과 판단이 흐르는 강물과 유사

- 텐서(tensor) – 0~ n 차원(축, axis)까지의 데이터 클래스
  - 0차 텐서 : 스칼라(0차원)
  - 1차 텐서: 벡터(1차원)
  - 2차 텐서: 행렬(2차원),
  - 3차원 이상: *n*차 텐서

- 텐서가 입력에서 출력까지 흐르며 학습



1. ***Scalars (0D tensors)***

- A scalar tensor contains only one number
- In Numpy, a `float32` or `float64` number is a scalar tensor (or scalar array).
- `ndim` attribute - the number of axes in Numpy tensor
- a scalar tensor - 0 axes (or rank), (`ndim == 0`)
- Here's a Numpy scalar:

```
>>> import numpy as np
>>> x = np.array(12)    # scalar tensor
>>> x
array(12)
>>> x.ndim
0
```

2. *Vectors (1D tensors)*

- An array of numbers is called a vector, or 1D tensor.
- A 1D tensor is said to have exactly one axis.
- Following is a Numpy vector:

```
>>> x = np.array([12, 3, 6, 14])
>>> x
array([12, 3, 6, 14])
>>> x.ndim
1
```

3. *Matrices (2D tensors)*

- An array of vectors is called *matrix*, or 2D tensor.
- A matrix has two axes (often referred to *rows* and *columns*).
- This is a Numpy matrix:

```
>>> x = np.array([[5, 78, 2, 34, 0],   # first row of x
                  [6, 79, 3, 35, 1],
                  [7, 80, 4, 36, 2]])
>>> x.ndim          # first column of x
2
```

4. *3D tensors and higher-dimensional tensors (nD tensors)*

- If you pack such matrices in a new array, you obtain a 3D tensor, which you can visually interpret as a cube of numbers. Following is a Numpy 3D tensor:

```
>>> x = np.array([[[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]]])
>>> x.ndim
3
```

5. ***Key attributes – tensor의 정의***

- *Number of axes (rank)* — 3D tensor == 3 axes, matrix == 2 axes, `ndim` in Numpy

- *Shape* — a tuple of integers
  shape of a scalar - `()`
  shape of a vector - `(5,)`
  shape of 3D tensor - `(3, 3, 5)`

- *Data type* (usually called `dtype` in Python libraries) —type of the data contained in the tensor;
  ex. `float32`, `uint8`, `float64`, `char`, no string tensors

5. *Key attributes – tensor의 정의*

- The data in the MNIST dataset:

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

- The number of axes of the tensor `train_images`, the `ndim` attribute:

```
>>> print(train_images.ndim)
3
```

- Here's its shape:

```
>>> print(train_images.shape)
(60000, 28, 28)
```

- And this is its data type, the `dtype` attribute:

```
>>> print(train_images.dtype)
uint8
```

5. *Key attributes – tensor의 정의*

- Let's display the fourth digit in this 3D tensor, using the library Matplotlib; see figure 2.2.

Listing 2.6 Displaying the fourth digit

```
digit = train_images[4]

import matplotlib.pyplot as plt
plt.imshow(digit,cmap=plt.cm.binary)
plt.show()
```

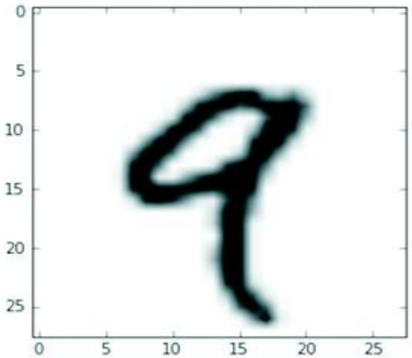


Figure 2.2 The fourth sample in our dataset

6. *Manipulating tensors in Numpy*

- *tensor slicing* - Selecting specific elements in a tensor

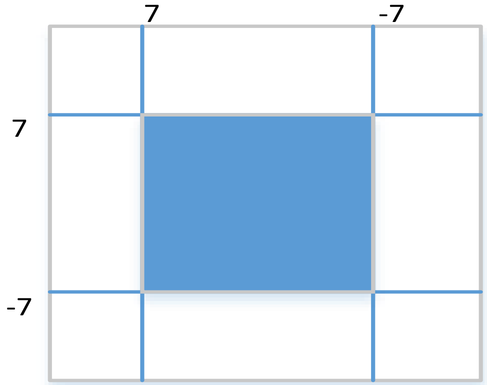
```
train_images[i]
```

- selects digits #10 to #100 (#100 isn't included, 10부터 100개) :

```
>>> my_slice = train_images[10:100]
>>> print(my_slice.shape) (90, 28, 28)

>>> my_slice = train_images[10:100, :, :]
>>> my_slice.shape
(90, 28, 28)

>>> my_slice = train_images[10:100, 0:28, 0:28]
>>> my_slice.shape
(90, 28, 28)
```

14 × 14 pixels centered in the middle

```
>>> my_slice = train_images[:, 7:-7, 7:-7]
7번째부터 끝에서 8번째까지
```



7. *The notion of data batches*

- here's one batch of our MNIST digits, with batch size of 128:
- the first axis (axis 0) is called the *batch axis* or *batch dimension*

```
batch = train_images[:128] # 0-127
batch = train_images[128:256] # 127-255
```

- the *n* th batch:

```
batch = train_images[128*n : 128*(n + 1)]
```

8. Real-world examples of data tensors

- *Vector data*—2D tensors of shape `(samples, features)`
- *Timeseries data or sequence data*—3D tensors of shape `(samples, timesteps, features)`
- *Images*—4D tensors of shape `(samples, height, width, channels)` or `(samples, channels, height, width)`
- *Video*—5D tensors of shape `(samples, frames, height, width, channels)` or `(samples, frames, channels, height, width)`

9. *Vector data*

the first axis is the *samples axis* and the second axis is the *features axis*

- dataset of people - age, ZIP code, and income.
  100,000 people - 2D tensor of shape `(100000, 3)`

- dataset of text documents - each document by the counts of how many times each word appears in it (out of a dictionary of 20,000 common words)
  500 documents - 2D tensor of shape `(500, 20000)`.

10. *Timeseries data or sequence data*

- A dataset of stock prices - 250 days , 390 minutes in a trading day, 3 features in a 3D tensor of shape `(250, 390, 3)`

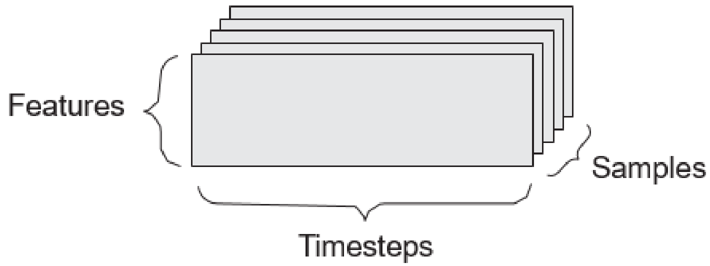


Figure 2.3 A 3D timeseries data tensor

- A dataset of tweets - 280 characters out of an alphabet of 128 unique characters `(280, 128)`,

dataset of 1 million tweets - `(1000000, 280, 128)`

11. *Image data*

- a batch of 128 color images could be stored in a tensor of shape `(128, 256, 256, 3)` (see figure 2.4)
- `(samples, height, width, color_depth).`

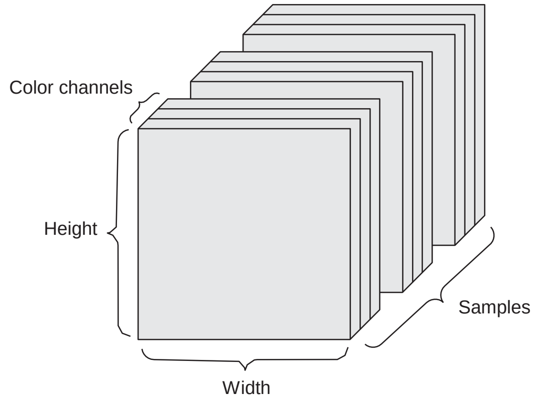


Figure 2.4 A 4D image data tensor (channels-first convention)

12. *Video data*

- a batch of different videos can be stored in a 5D tensor of shape `(samples, frames, height, width, color_depth)`
- a 60-second, 144 $\times$ 256 YouTube video clip sampled at 4 frames per second would have 240 frames – `(4, 240, 144, 256, 3)`

- building our network by stacking `Dense` layers

- A Keras layer instance looks like this:

```
keras.layers.Dense(512,
activation='relu')
```

- where `W` is a 2D tensor and `b` is a vector, both attributes of the layer:

```
output = relu(dot(W, input) + b
```

1. *Element-wise operations*

Matrix vs. Element-wise operations

```
>> A=[1 2; 4 5]
     1     2
     4     5
```

```
>> B = [10 20; 30 40]
    10    20
    30    40
```

- Matrix multiplication

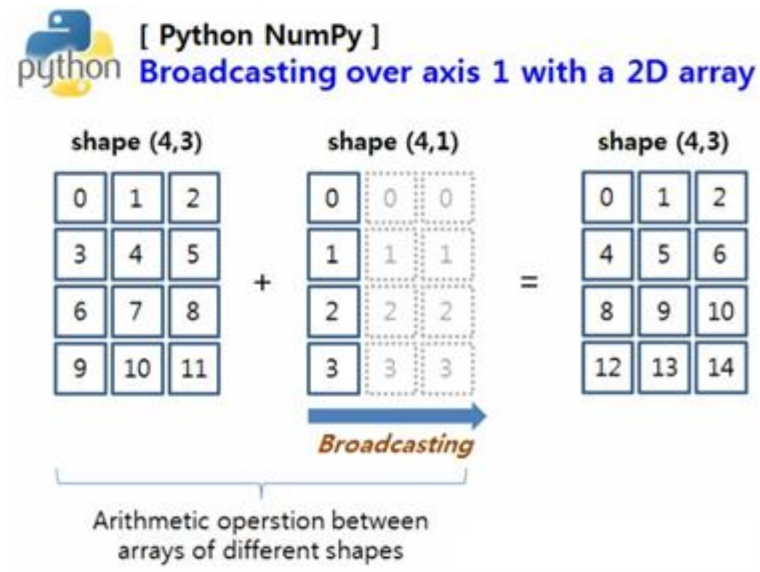
```
>> A * B
    70   100
   190   280
```

- Element-wise multiplication

```
>> A .* B
    10    40
   120   200
```

```
import numpy as np
z = x + y
z = np.maximum(z, 0.)
```

2. ***Broadcasting***

- 

- element-wise `maximum` operation to two tensors of different shapes via broadcasting:

```
import numpy as np
x = np.random.random((64, 3, 32, 10))
y = np.random.random((32, 10))
z = np.maximum(x, y)  # shape (64, 3, 32, 10) like x.
```

3. *Tensor dot*

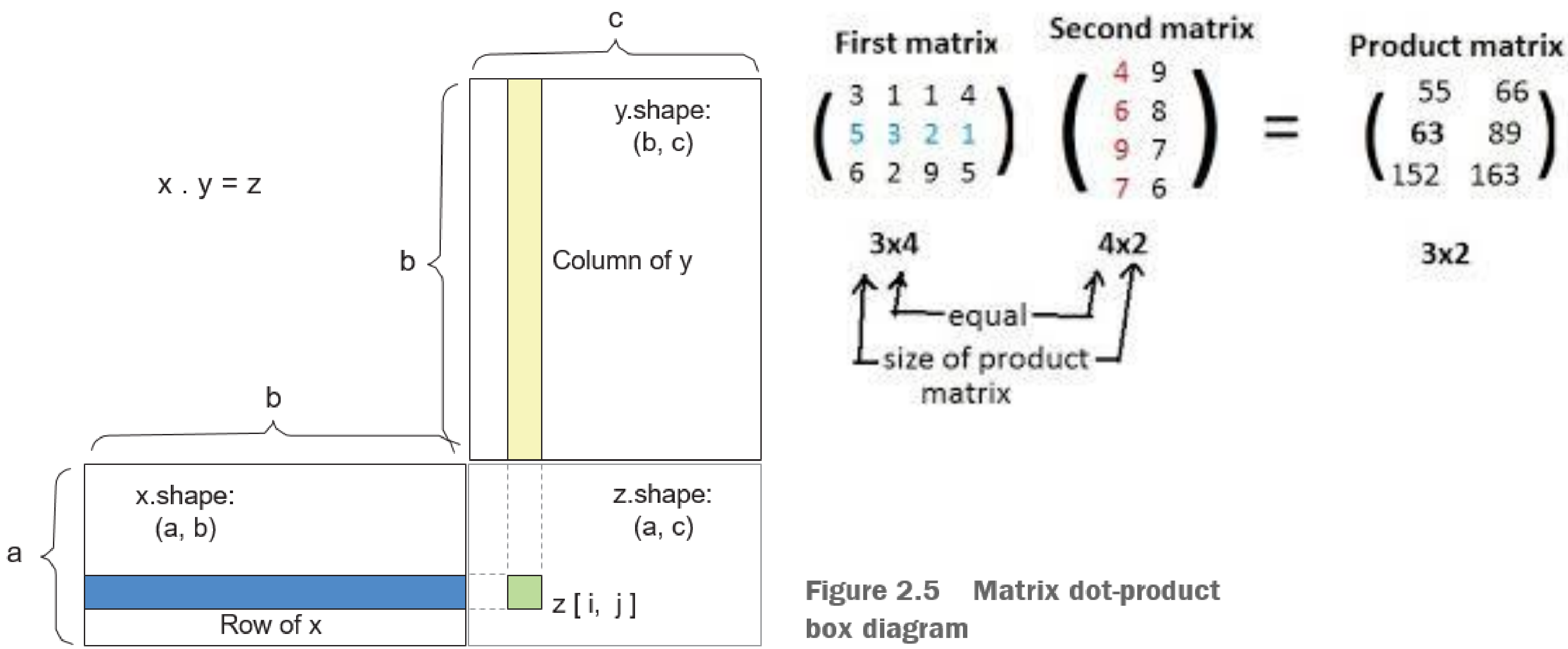


Figure 2.5 Matrix dot-product box diagram

4. ***Tensor reshaping***

```
train_images = train_images.reshape((60000, 28 * 28))
>>> x = np.array([[0., 1.],
                  [2., 3.],
                  [4., 5.]])
>>> print(x.shape)
(3, 2)
>>> x = x.reshape((6, 1))
>>> x
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.]])

>>> x = x.reshape((2, 3))
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
```

- A special case of reshaping that's commonly encountered is *transposition*. *Transposing* a matrix means exchanging its rows and its columns, so that `x[i, :]` becomes `x[:, i]`:

```
>>> x = np.zeros((300, 20))
>>> x = np.transpose(x)
>>> print(x.shape)
(20, 300)
```

1. *What's a derivative?*

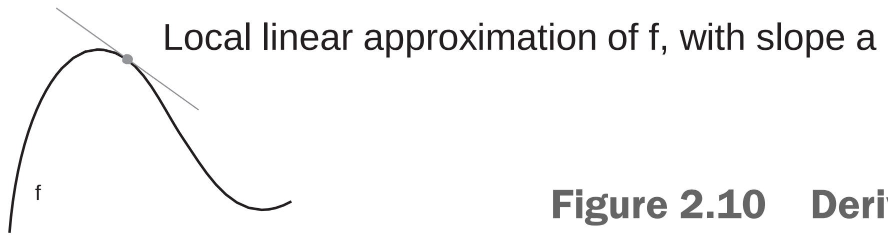


Figure 2.10 Derivative of `f` in `p`

- **Derivative:** A *gradient* is the derivative of a tensor operation.

2.4 The engine of neural networks: gradient-based optimization

3. *Stochastic gradient descent*

- The term *stochastic* refers to the fact that each batch of data is drawn at random.

1 Draw a batch of training samples `x` and corresponding targets `y`.
2 Run the network on `x` to obtain predictions `y_pred`.
3 Compute the loss of the network on the batch, a measure of the mismatch between `y_pred` and `y`.
4 Compute the gradient of the loss with regard to the network's parameters (a *backward pass*).
5 Move the parameters a little in the opposite direction from the gradient—for example `W -= step * gradient`—thus reducing the loss on the batch a bit.

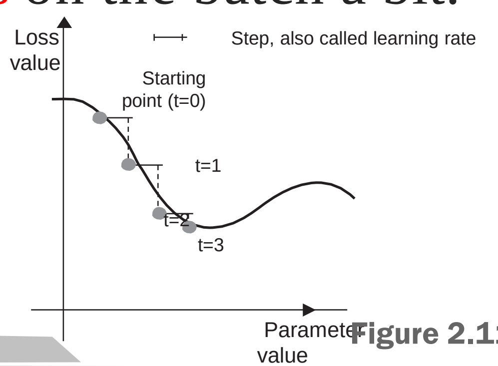


Figure 2.11 SGD down a 1D loss curve (one learnable parameter)

- If `step` is too small, the descent down the curve will take many iterations - stuck in a local minimum.
- If `step` is too large - completely random locations on the curve.

2.4 The engine of neural networks: gradient-based optimization

3. *Stochastic gradient descent*

- visualize gradient descent along a 2D loss surface, as shown in figure 2.12.
- can’t possibly visualize what the actual process of training a neural network looks like—1,000,000-dimensional space
- *Optimizers* - Adagrad, RMSProp, …

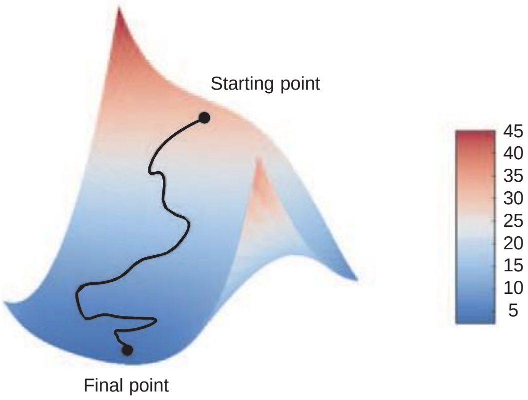


Figure 2.12 Gradient descent down a 2D loss surface (two learnable parameters)

3. *Stochastic gradient descent*

- Momentum with SGD: convergence speed and local minima

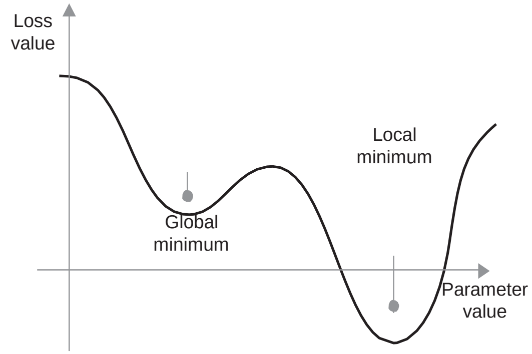


Figure 2.13 A local minimum and a global minimum

2.4 The engine of neural networks: gradient-based optimization

5. *Looking back at our first example*

- review each piece of it in the light of what you've learned in the previous three sections

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255

network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))

network.compile(optimizer='rmsprop',loss='categorical_crossentropy',metrics=['accuracy'])

network.fit(train_images, train_labels, epochs=5, batch_size=128)
```