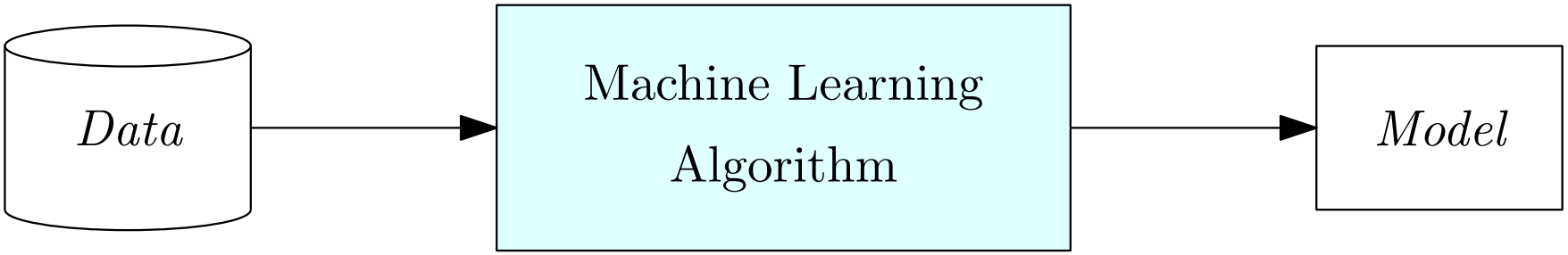
Chapter 1

Intro to Machine Learning and Perceptron

1.1 Machine Learning

Machine Learning (ML) studies how to make a computer system improve its performance using a collection of observed data. ML focuses on

1. building a model from a training set, and
2. using the model as a hypothesis about the world and a software to solve problems.



1.1.1 Types of Machine Learning

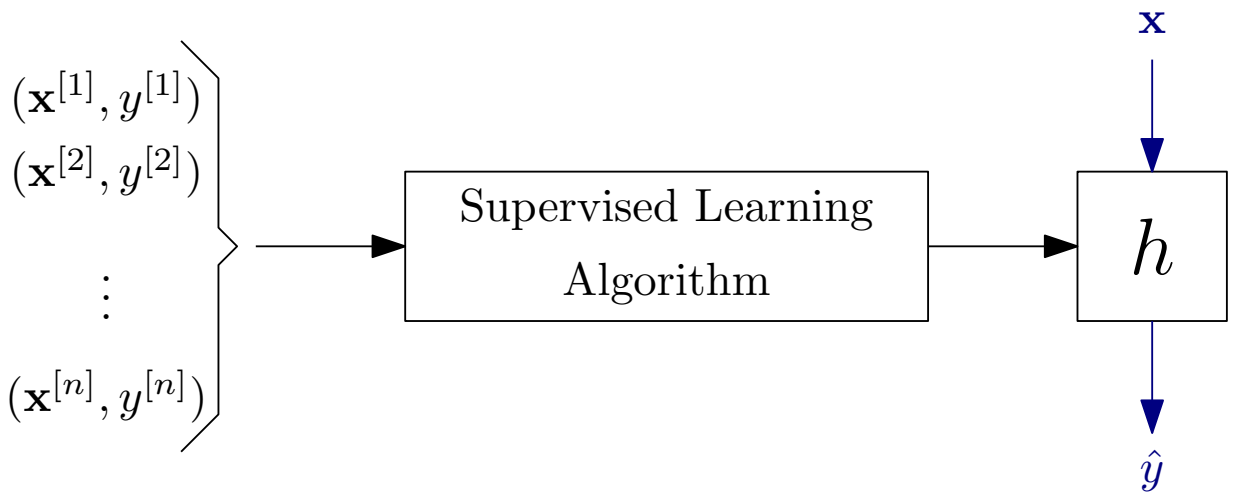
Several types of machine learning techniques have been developed until now, e.g.

- *Supervised learning* learns a function that maps from input to output from a set of input-output pairs.

  Given a training set of n input-output pairs

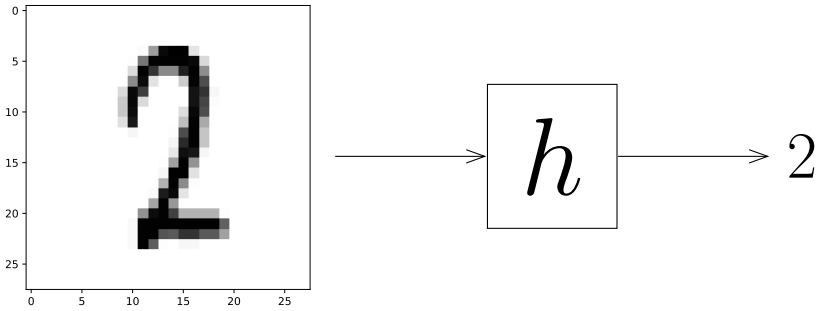
  $$(\mathbf{x}^{[1]}, y^{[1]}), (\mathbf{x}^{[2]}, y^{[2]}), \ldots, (\mathbf{x}^{[n]}, y^{[n]})$$

  where each input-output pair was generated by an unknown function $y = f(\mathbf{x})$, a supervised algorithm finds a function $h : \mathcal{X} \to \mathcal{Y}$ that approximates f.

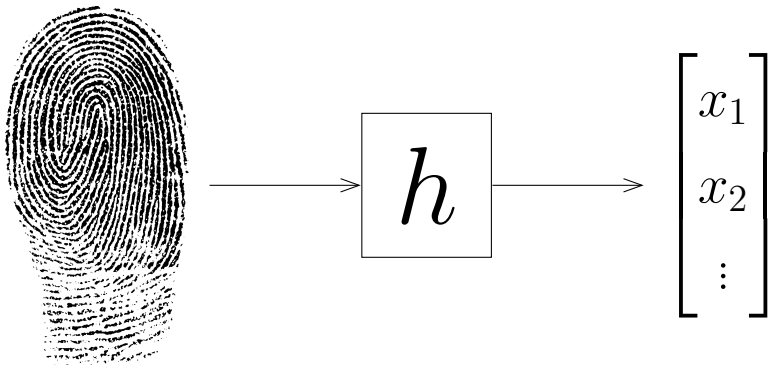


The hypothesis h can be used to predict an output $\hat{y}$ from an unlabeled input $\mathbf{x}$.

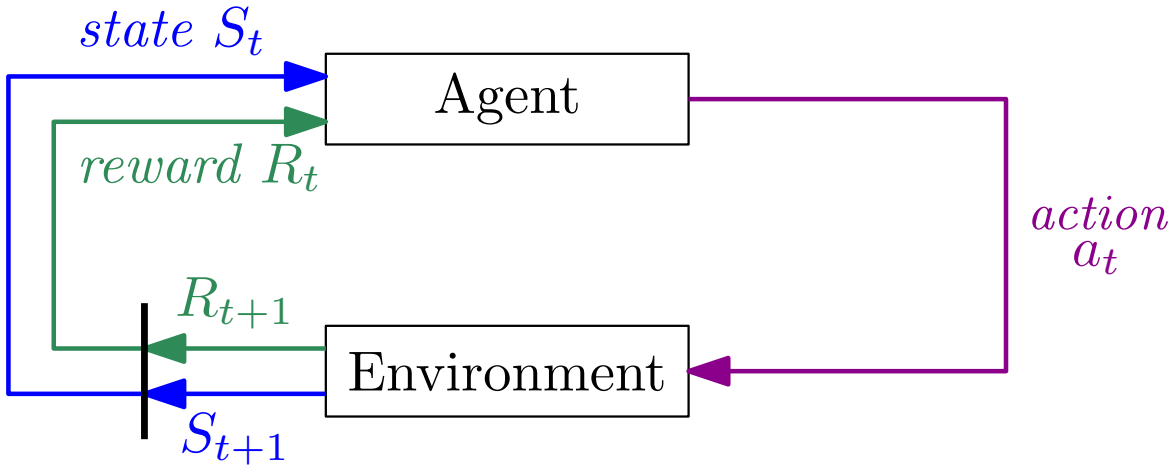
For example, given a set of images of handwritten digits, we can use a supervised learning algorithm to generate a hypothesis h that predicts a digit from an image.



- *Unsupervised learning* learns from a set of unlabeled examples. Tasks of an unsupervised learning are:
  - to learn a function $h : \mathcal{X} \rightarrow \mathcal{Z}$ that maps an unlabeled input $\mathbf{x}$ into new representations.
  - to learn a generative model such as a probability distribution to generate new examples.

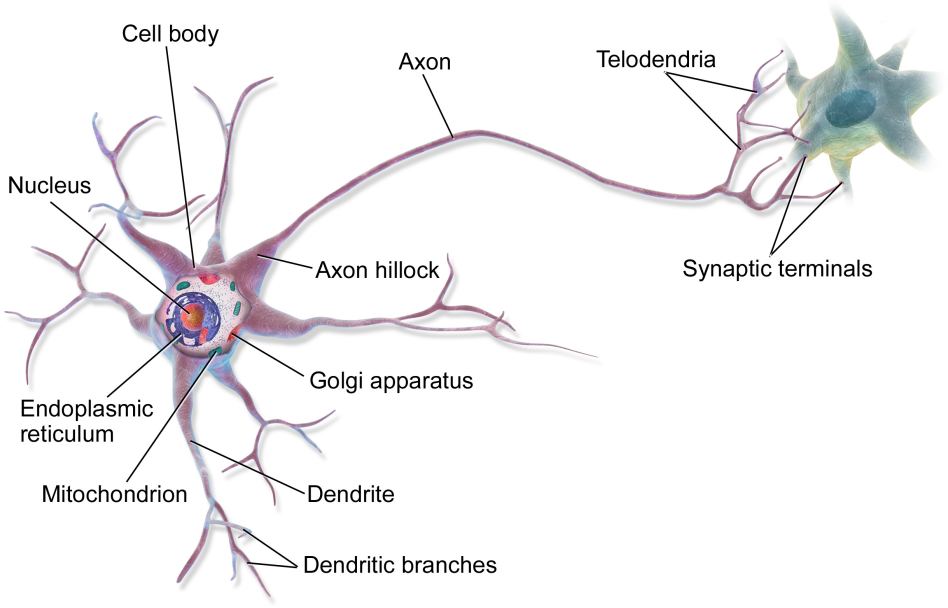


- *Reinforcement learning* learns from a series of reinforcements: rewards/punishments. For example, a chess playing agent receives a reward of 1 for a win, 0 for a lose, and 0.5 for a draw. The agent then uses the received reward to adjust its actions in order to maximize rewards in the future.



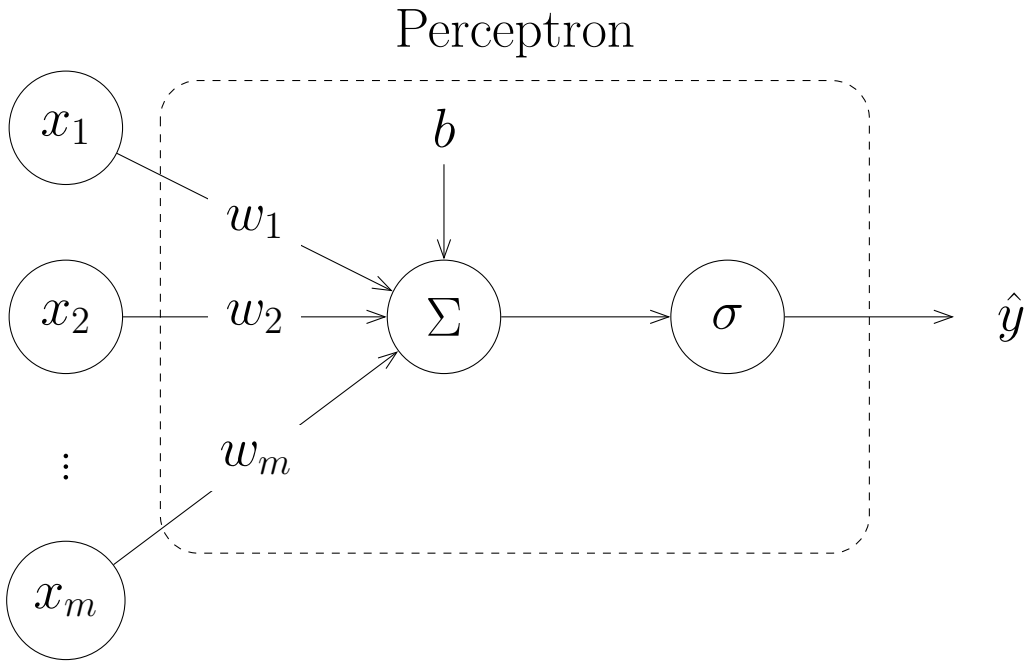
1.2 Perceptron

Perceptron is a supervised machine learning algorithm working as a *binary classifier*. It accepts *a real-valued vector*, and decides whether the vector belongs to a class 0 or 1. The perceptron was proposed by Frank Rosenblatt in 1957 based on the neuron model by Warren McCulloch and Walter Pitts and the findings of Donald Hebb. It was inspired by biological brains and neurons.



Source: `https://commons.wikimedia.org/wiki/File:Blausen_0657_MultipolarNeuron.png`

A perceptron can be represented as the following diagram.



A perceptron accepts m inputs, and produces 1 output. The m inputs of the perceptron is a real-valued vector, $\mathbf{x} \in \mathbb{R}^{m \times 1}$, i.e.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \tag{1.1}$$

or,

$$\mathbf{x}^\top = \begin{bmatrix} x_1 & x_2 & \dots & x_m \end{bmatrix} \tag{1.2}$$

where $\mathbf{x}^\top \in \mathbb{R}^{1 \times m}$. Each vector represents an object to be classified if it belongs to the class. The perceptron produces 1 output called $\hat{y} \in \{0, 1\}$.

Each perceptron has 2 types of parameters i.e.

1. a weight vector $\mathbf{w} \in \mathbb{R}^{m \times 1}$ and
2. a bias b.

Calculating an output Given an input vector $\mathbf{x}^\top = [x_1 \quad x_2 \quad \dots \quad x_m]$, the output of a perceptron is calculated from:

$$\hat{y} = \sigma\left(\sum_{i=1}^{m} x_i w_i + b\right) \tag{1.3}$$

$$= \sigma\left(x_1 w_1 + x_2 w_2 + \cdots + x_m w_m + b\right) \tag{1.4}$$

$$= \sigma\left(\mathbf{x}^\top \mathbf{w} + b\right) \tag{1.5}$$

Some textbook may refer to the bias b as w_0. Then,

$$\hat{y} = \sigma\left(\sum_{i=0}^{m} x_i w_i\right) \tag{1.6}$$

$$= \sigma\left(x_0 w_0 + x_1 w_1 + x_2 w_2 + \cdots + x_m w_m\right) \tag{1.7}$$

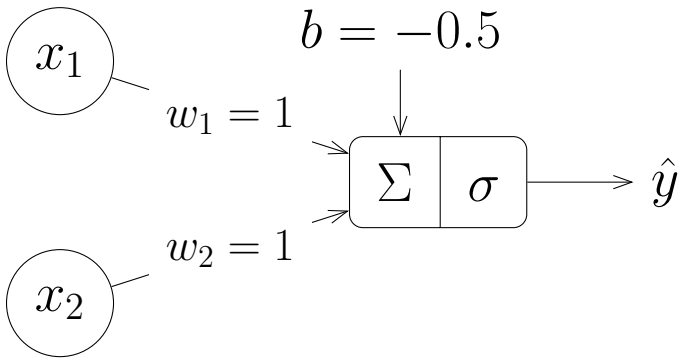
Note that x_0 is always 1.

Activation function $\sigma(\cdot)$ is an activation function. It is defined as:

$$\sigma(z) = \begin{cases} 0 & z \leq 0 \\ 1 & z > 0 \end{cases} \tag{1.8}$$

In summary, a perceptron accepts m real-valued inputs. Then, it calculates a *weighted sum* of the inputs. The sum is fed to the activation function that returns either 0 or 1. The output of the activation function is the output of the perceptron.

Example 1.1. What is the output of the following perceptron when we input each of the following vectors?



(1) $[0 \quad 0]^\top$

(2) $[0 \quad 1]^\top$

(3) $[1 \quad 0]^\top$

(4) $[1 \quad 1]^\top$

From the previous example with $w_1 = 1, w_2 = 1, b = 0.5$, the perceptron works like the OR gate.

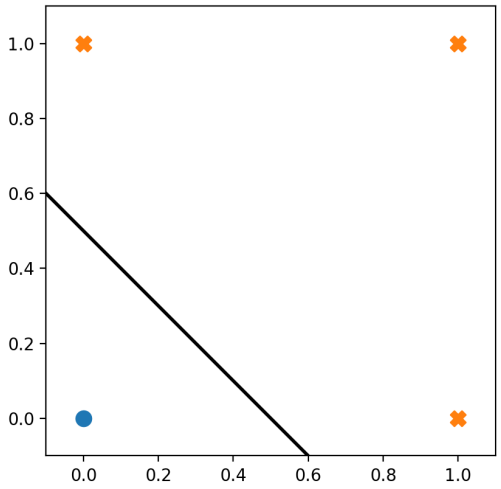
x_1	x_2	$\hat{y}$
0	0	0
0	1	1
1	0	1
1	1	1

1.3 Decision Boundary

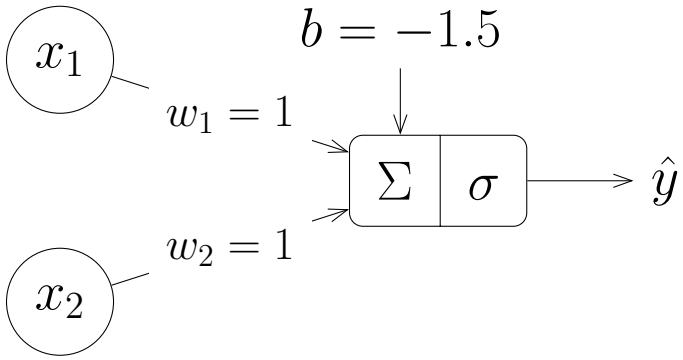
Since the activation function, $\sigma(\cdot)$, uses 0 as the threshold, the decision boundary of a perceptron is therefore

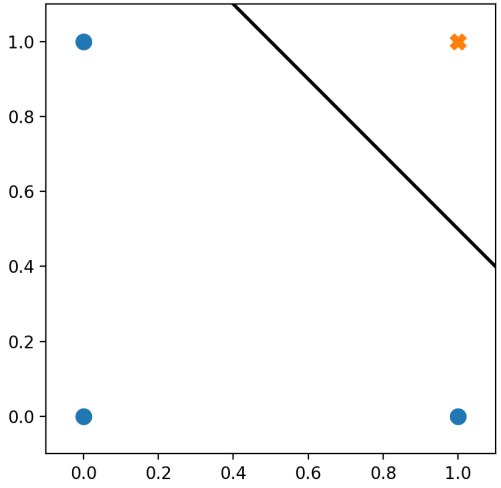
$$x_1 w_1 + x_2 w_2 + \cdots + x_m w_m + b = 0 \tag{1.9}$$

Example 1.2. The decision boundary of the previous perceptron is $x_1 + x_2 - 0.5 = 0$.

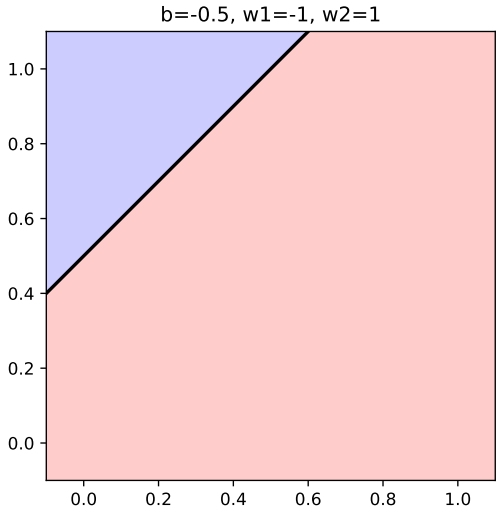


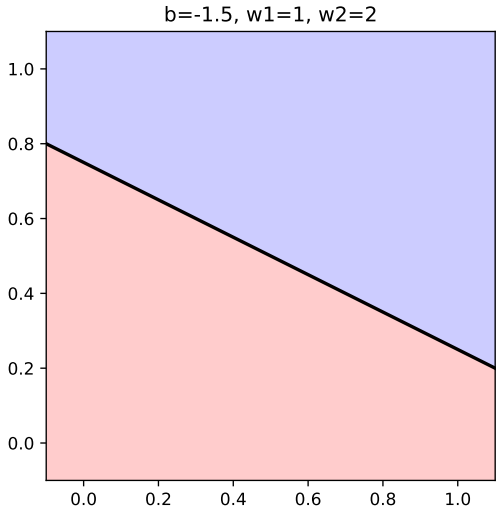
Example 1.3. What is the decision boundary of the following perceptron?

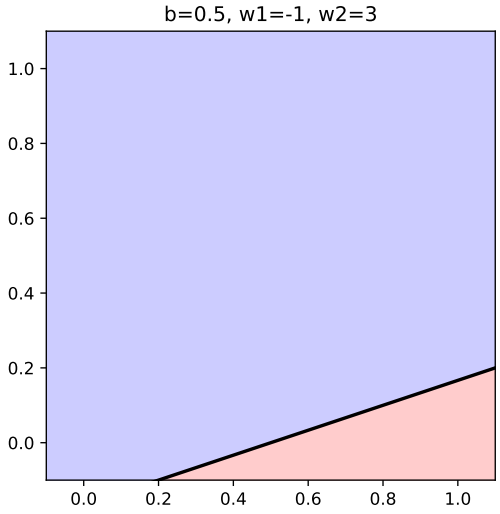


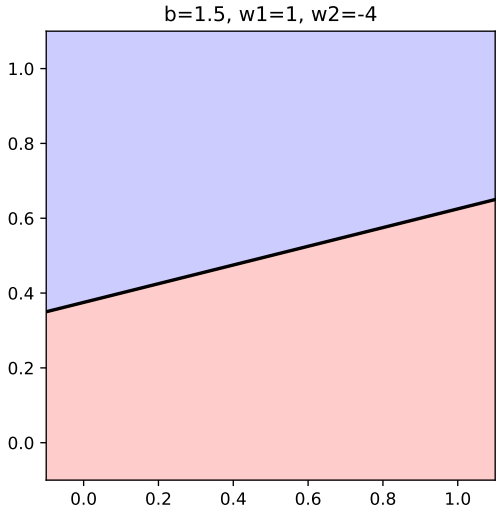


Different values of $\mathbf{w}$ and b results in different decision boundaries.



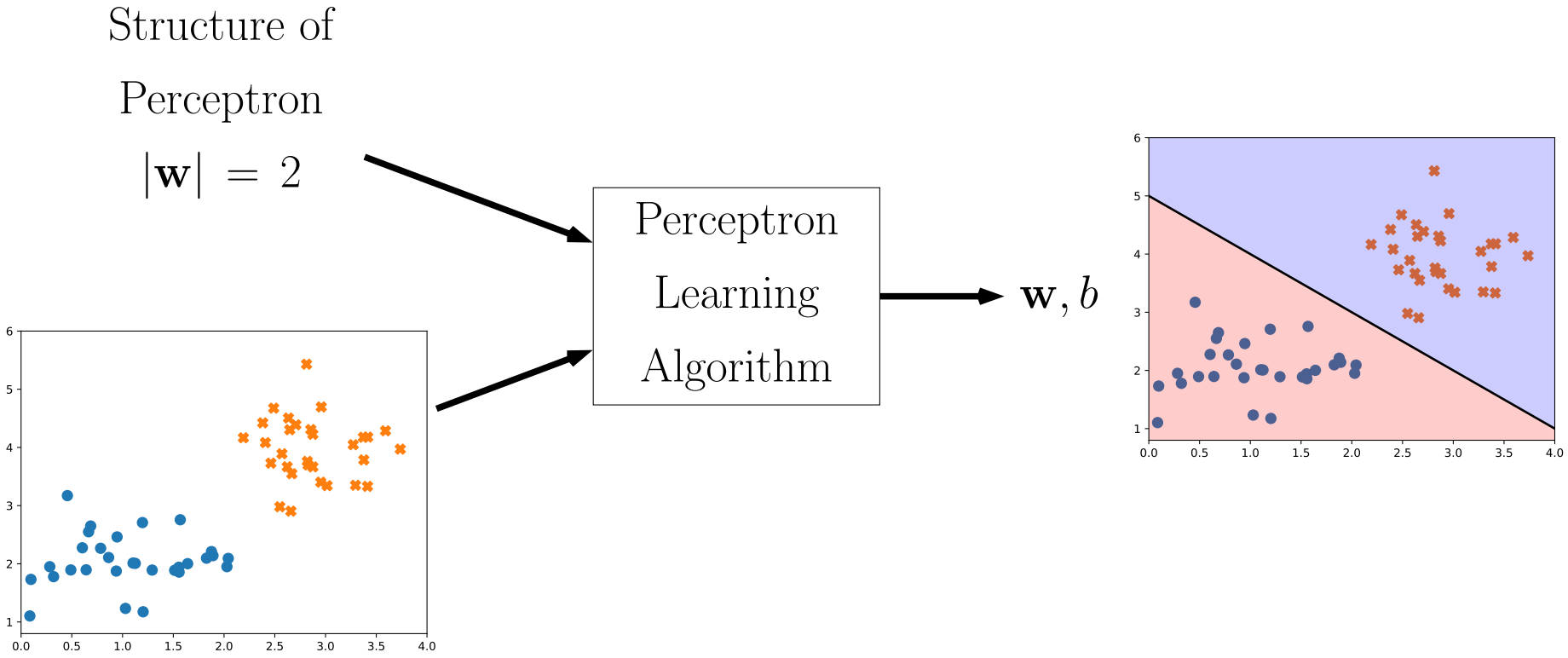






1.4 Training a perceptron

To use a perceptron in real-world datasets, the *perceptron learning algorithm* was developed to find appropriate values of perceptron parameters from a dataset.



1.4.1 Training Examples

One of the perceptron learning algorithm is a *set of training examples*. Each element of the set is a pair of *input vectors* and a *desired output*.

$$\mathcal{D} = \left\{ \langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, \langle \mathbf{x}^{[3]}, y^{[3]} \rangle, \ldots, \langle \mathbf{x}^{[n]}, y^{[n]} \rangle, \right\} \in (\mathbb{R}^m \times \{0, 1\})^n \tag{1.10}$$

where

- each $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle$ is a labeled example,
- each $\mathbf{x}^{[i]} = [x_1 \quad x_2 \quad \ldots \quad x_m]^\top$ is an input vector, and
- each $y^{[i]}$ is a desired or target output, when $i \in [0, 1, \ldots, n]$,

This setting is called:

- *regression* when the set of possible output is an infinite set, e.g. $\mathbb{R}$,
- *classification* when the set of possible output is a finite set,
- *binary classification* when there are only two different types of output, e.g. $\{0, 1\}$,
- *multi-class classification* when there more than two types of output.

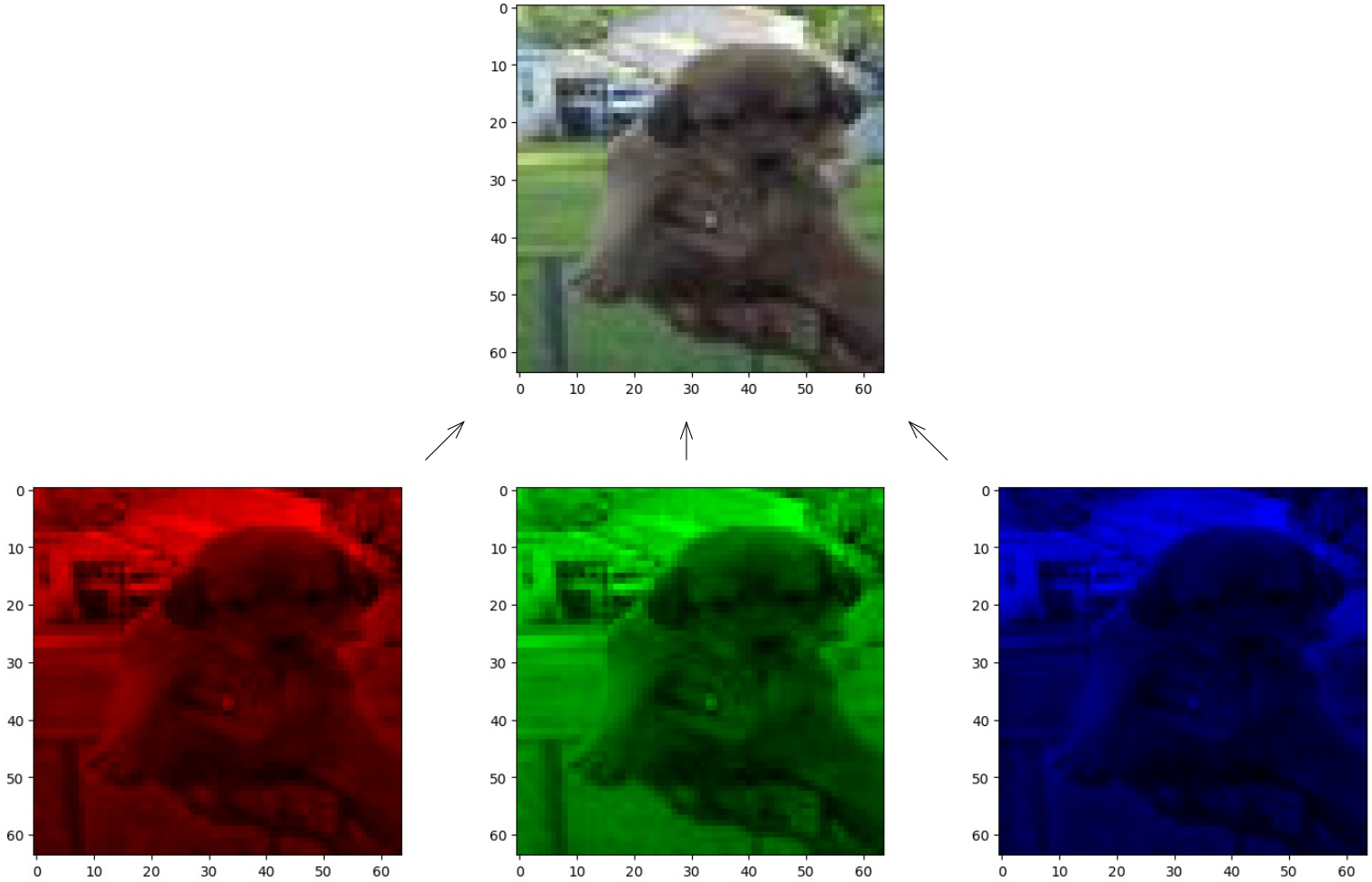
Example 1.4. Given a collection of dog and cat images[1], how can we design a perceptron to identify if an image is a cat image or a dog image?

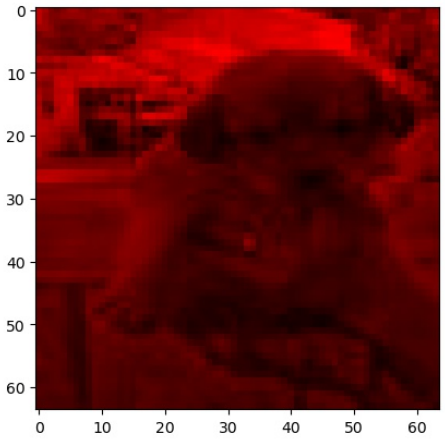
[1] https://www.kaggle.com/c/dogs-vs-cats/overview

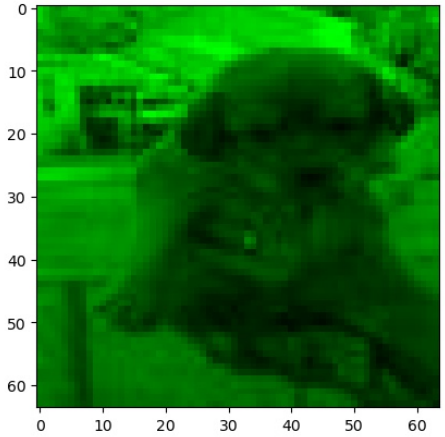
Since a perceptron accepts only a fixed-sized vector, we first need to resize all the images to make them the same size. We can, for example, set the size of our input to be 64×64. The size of each image may be set based on our computational power and memory size. However, the higher the image size is, the better the prediction results are.

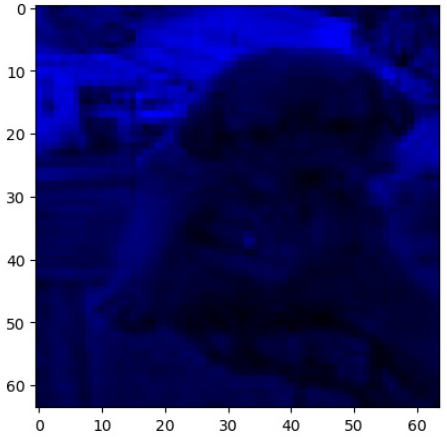
Each pixel in an image is represented by a combination of 3 values: R, G, and B.

$64 \times 64 \times 3$










```
[[0.75 0.64 0.5  0.44 0.52 ... 0.53 0.49 0.39 0.53 0.52]
 [0.74 0.6  0.49 0.5  0.55 ... 0.39 0.4  0.41 0.55 0.58]
 [0.73 0.62 0.51 0.47 0.45 ... 0.42 0.42 0.47 0.58 0.7 ]
 [0.62 0.62 0.55 0.44 0.38 ... 0.52 0.44 0.45 0.46 0.57]
 [0.47 0.53 0.53 0.47 0.42 ... 0.46 0.38 0.44 0.39 0.45]
 ...
 [0.28 0.29 0.31 0.32 0.31 ... 0.15 0.2  0.2  0.2  0.2 ]
 [0.31 0.31 0.31 0.33 0.34 ... 0.17 0.17 0.16 0.18 0.2 ]
 [0.31 0.3  0.29 0.3  0.32 ... 0.22 0.16 0.13 0.18 0.24]
 [0.28 0.29 0.27 0.27 0.28 ... 0.25 0.2  0.19 0.2  0.22]
 [0.28 0.29 0.29 0.27 0.27 ... 0.24 0.26 0.27 0.24 0.17]]
```

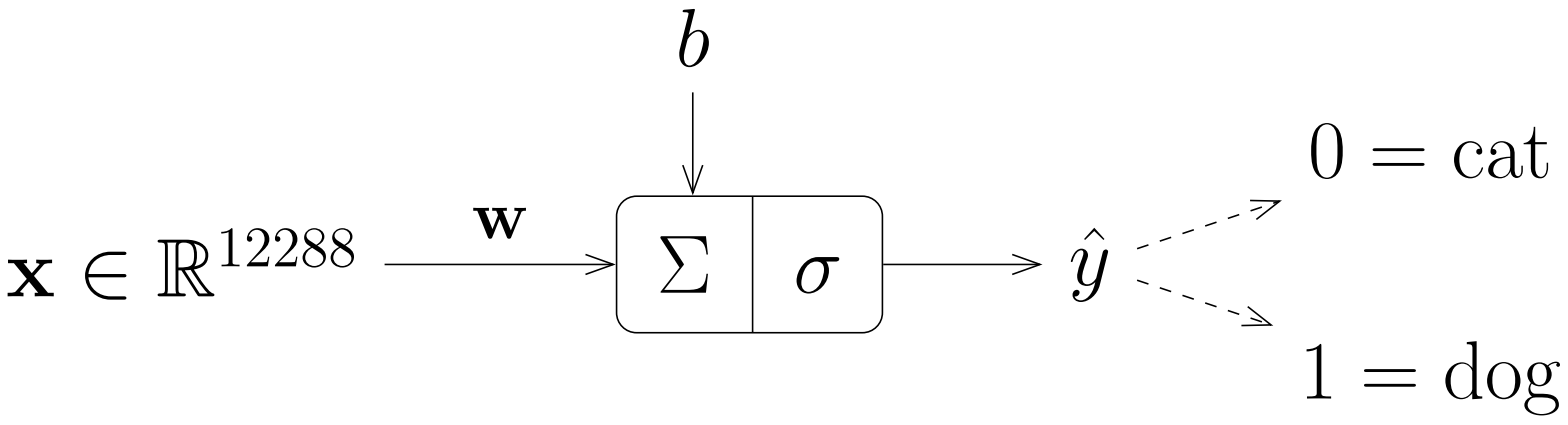
```
[[0.84 0.73 0.59 0.53 0.62 ... 0.64 0.61 0.51 0.64 0.64]
 [0.82 0.69 0.59 0.6  0.65 ... 0.5  0.51 0.52 0.66 0.7 ]
 [0.83 0.71 0.6  0.56 0.55 ... 0.53 0.53 0.57 0.69 0.8 ]
 [0.73 0.72 0.65 0.54 0.48 ... 0.62 0.54 0.55 0.55 0.67]
 [0.58 0.64 0.64 0.57 0.52 ... 0.53 0.46 0.52 0.47 0.53]
 ...
 [0.43 0.44 0.44 0.44 0.4  ... 0.11 0.15 0.16 0.16 0.16]
 [0.47 0.46 0.45 0.45 0.43 ... 0.13 0.13 0.13 0.15 0.17]
 [0.47 0.46 0.43 0.42 0.41 ... 0.18 0.12 0.1  0.15 0.2 ]
 [0.45 0.44 0.42 0.39 0.37 ... 0.21 0.16 0.15 0.17 0.18]
 [0.45 0.45 0.44 0.39 0.37 ... 0.2  0.22 0.24 0.2  0.14]]
```

```
[[0.56 0.46 0.33 0.27 0.36 ... 0.43 0.35 0.19 0.27 0.24]
 [0.57 0.44 0.34 0.35 0.42 ... 0.3  0.27 0.22 0.31 0.31]
 [0.6  0.49 0.38 0.35 0.35 ... 0.34 0.3  0.3  0.36 0.45]
 [0.55 0.55 0.47 0.36 0.3  ... 0.46 0.35 0.31 0.27 0.36]
 [0.46 0.51 0.52 0.44 0.39 ... 0.41 0.31 0.33 0.24 0.28]
 ...
 [0.22 0.25 0.3  0.34 0.34 ... 0.12 0.17 0.18 0.17 0.18]
 [0.25 0.27 0.29 0.35 0.38 ... 0.13 0.14 0.13 0.16 0.18]
 [0.25 0.26 0.27 0.31 0.36 ... 0.17 0.12 0.1  0.15 0.2 ]
 [0.23 0.24 0.25 0.29 0.32 ... 0.2  0.16 0.16 0.17 0.19]
 [0.23 0.25 0.27 0.29 0.31 ... 0.2  0.22 0.24 0.2  0.14]]
```

```
[0.75 0.84 0.56 0.64 0.73 ...  0.2 0.2 0.17 0.14 0.14]
```

$(64 \times 64 \times 3 = 12,288)$

Therefore, we use a perceptron with 12,288 inputs and outputs 1 value: 0 for cat and 1 for dog.



1.4.2 Perceptron Training Algorithm (1)

Given a training set, $\mathcal{D} = \{\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, \langle \mathbf{x}^{[3]}, y^{[3]} \rangle, \ldots, \langle \mathbf{x}^{[n]}, y^{[n]} \rangle, \}$

(1) initialize $\mathbf{w} \leftarrow 0^m = [0 \quad 0 \quad \ldots \quad 0]^\top, b = 0$, or small values around 0

(2) for every training epoch:

(2.1) for every labeled example $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$:

(2.1.1) feed $\mathbf{x}^{[i]}$ to the perceptron and calculate $\hat{y}^{[i]}$:

$$\hat{y}^{[i]} = \sigma\left(\sum_{i=1}^{m} x_i w_i + b\right)$$

(2.1.2) $err \leftarrow (y^{[i]} - \hat{y}^{[i]})$

(2.1.3) if $err = 0$, do nothing

else update $\mathbf{w} \leftarrow \mathbf{w} + err \times \mathbf{x}^{[i]}$ and $b \leftarrow b + err$

From the algorithm, there are two possible values of err:

- When $y^{[i]} = 0$, $\hat{y}^{[i]} = 1$, we have $err = -1$. The weight is updated by

$$\mathbf{w} \leftarrow \mathbf{w} - \mathbf{x}^{[i]} \tag{1.11}$$

$$b \leftarrow b - 1 \tag{1.12}$$

- When $y^{[i]} = 1$, $\hat{y}^{[i]} = 0$, we have $err = +1$. The weight is updated by

$$\mathbf{w} \leftarrow \mathbf{w} + \mathbf{x}^{[i]} \tag{1.13}$$

$$b \leftarrow b + 1 \tag{1.14}$$

In practice, the dataset are usually shuffle prior each epoch.

1.4.3 Perceptron Training Algorithm (2)

Given a training set, $\mathcal{D} = \{\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, \langle \mathbf{x}^{[3]}, y^{[3]} \rangle, \ldots, \langle \mathbf{x}^{[n]}, y^{[n]} \rangle, \}$

(1) initialize $\mathbf{w} \leftarrow 0^m = [0 \quad 0 \quad \ldots \quad 0]^\top, b = 0$, or small values around 0

(2) for every training epoch:

(2.1) shuffle $\mathcal{D}$

(2.2) for every labeled example $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$:

(2.2.1) feed $\mathbf{x}^{[i]}$ to the perceptron and calculate $\hat{y}^{[i]}$:

$$\hat{y}^{[i]} = \sigma\left(\sum_{i=1}^{m} x_i w_i + b\right)$$

(2.2.2) $err \leftarrow (y^{[i]} - \hat{y}^{[i]})$

(2.2.3) if $err = 0$, do nothing

else update $\mathbf{w} \leftarrow \mathbf{w} + err \times \mathbf{x}^{[i]}$ and $b \leftarrow b + err$

Example 1.5. Given the truth table of the OR gate,

$$\mathbf{x}^{[1]} = [0 \quad 0]^\top,\ y^{[1]} = 0,\ \mathbf{x}^{[2]} = [0 \quad 1]^\top,\ y^{[2]} = 1,$$
$$\mathbf{x}^{[3]} = [1 \quad 0]^\top,\ y^{[3]} = 1,\ \mathbf{x}^{[4]} = [1 \quad 1]^\top,\ y^{[4]} = 1,$$

calculate $\mathbf{w}, b$ after the first training epoch. Here, assume that the training examples are shuffled into $3 \to 4 \to 1 \to 2$. Initially, $\mathbf{w} = [0 \quad 0], b = 0$.

1. $\hat{y}^{[3]} = \sigma\left(1 \cdot 0 + 0 \cdot 0 + 0\right) = \sigma(0) = 0 \Rightarrow err = y^{[3]} - \hat{y}^{[3]} = 1 - 0 = 1$
   $\mathbf{w} \leftarrow [0 \quad 0]^\top + [1 \quad 0]^\top = [1 \quad 0]^\top, b \leftarrow 0 + 1 = 1$

2. $\hat{y}^{[4]} = \sigma\left(1 \cdot 1 + 1 \cdot 0 + 1\right) = \sigma(2) = 1 \Rightarrow err = y^{[4]} - \hat{y}^{[4]} = 1 - 1 = 0$
   no update

3. $\hat{y}^{[1]} = \sigma\left(0 \cdot 1 + 0 \cdot 0 + 1\right) = \sigma(1) = 1 \Rightarrow err = y^{[1]} - \hat{y}^{[1]} = 0 - 1 = -1$
   $\mathbf{w} \leftarrow [1 \quad 0]^\top - [0 \quad 0]^\top = [1 \quad 0]^\top, b \leftarrow 1 - 1 = 0$

4. $\hat{y}^{[2]} =$

1.4.4 Implementation of Perceptron using NumPy

```
1  import numpy as np
2  import random
3
4  random.seed(1101)
5
6  def s(z):
7      if z<=0:
8          return 0
9      return 1
10
11 # Training examples
12 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
13 y = np.array([0, 1, 1, 1])
14 # Weights and bias are initialized 0.
15 w = np.array([0, 0])
16 b = 0
17
```

```
18 # list of indices
19 indices = list(range(4))
20 # repeat at most 10 epochs
21 for r in range(10):
22     print("Epoch %d:" % (r+1))
23     # shuffle indices
24     random.shuffle(indices)
25     n_updates = 0
26     # repeat for each example
27     for i in indices:
28         # calculate a prediction
29         y_hat = s(np.dot(X[i], w) + b)
30         # calculate err
31         err = y[i] - y_hat
32         # update w, b if err is non-zero
33         if err != 0:
34             w += err*X[i]
35             b += err
36             n_updates += 1
```

```
37            print(w, b)
38        print()
39        # stop if no updates occur in this epoch
40        if n_updates == 0:
41            break
42 print("="*20)
43 # calculate and display predicted results
44 for i in range(4):
45        y_hat = s(np.dot(X[i], w) + b)
46        print(X[i], y[i], y_hat)
```

1.4.5 Prediction Accuracy

Prediction accuracy is a measure to evaluate the performance of the trained perceptron. It is calculated from

$$acc = \frac{tp + tn}{n} = \frac{correctly\ predicted}{n} \tag{1.15}$$

where

- tp is the number of true positives,
- tn is the number of true negatives, and
- n is the total number of examples.

Note that the accuracy calculated from the training example is usually better than the real accuracy since we evaluate the perceptron by the data used to train the perceptron. Each perceptron should be evaluated by a separated dataset.

1.5 Dogs vs. Cats Dataset

1.5.1 Preprocessing

The dataset contains 12,500 cat images, i.e. `cat.0.jpg`, ..., `cat.12499.jpg`, and 12,500 dog images, i.e. `dog.0.jpg`, ..., `dog.12499.jpg`. The images have varying dimensions. As explained previously, we need to resize them into the same dimension.

```
1  import numpy as np
2  from PIL import Image
3
4  # number of images
5  N = 12500
6  # prepare an empty array
7  # 1 row for 1 image
8  D = np.empty((2*N, 12288))
9
10 for i, t in enumerate(['cat', 'dog']):
11     for j in range(N):
```

```
12          fname = './train/%s.%d.jpg' % (t, j)
13          print(fname)
14          # load an image
15          im = Image.open(fname)
16          # resize the image
17          im = im.resize((64, 64))
18          # convert the image into a numpy array
19          im = np.array(im)
20          # reshape the image array into a row vector
21          # (64, 64, 3) -> (12288,)
22          im = im.flatten()
23          # append the image into the dataset
24          D[N*i+j, :] = im
25
26  # originally each color value is in [0, 255]
27  # make it in [0.0, 1.0]
28  D = D/255
29  # save the dataset
30  np.save("catdog_train.npy", D)
```

1.5.2 Training

```
1  import numpy as np
2  import random
3  import matplotlib.pyplot as plt
4
5  random.seed(1101)
6
7  def s(z):
8      if z <= 0:
9          return 0
10     return 1
11
12 # load dataset
13 X = np.load("catdog_train.npy")
14 n = X.shape[0]//2
15 y = np.concatenate((np.zeros(n), np.ones(n)))
16
17 # initialize w, b
```

```
18 w = np.zeros(X.shape[1])
19 b = 0
20
21 indices = list(range(n*2))
22 for r in range(100):
23     print("Epoch %d" % (r+1))
24     # shuffle indices
25     random.shuffle(indices)
26     n_updates = 0
27     # repeat for each example
28     for i in indices:
29         # calculate a prediction
30         y_hat = s(np.dot(X[i], w) + b)
31         # calculate err
32         err = y[i] - y_hat
33         # update w, b if err is non-zero
34         if err != 0:
35             w += err*X[i]
36             b += err
```

```
37                 n_updates += 1
38         # stop if no updates occur in this epoch
39         if n_updates == 0:
40             break
41
42 # calculate accuracy
43 n_corrects = 0
44 for i in range(n*2):
45         y_hat = s(np.dot(X[i], w) + b)
46         if y[i] == y_hat:
47             n_corrects += 1
48 print("Training accuracy = %.2f" % (n_corrects/(n*2)*100))
49
50 # save weights to w.npy
51 np.save('w', w)
52 # save bias to b.npy
53 np.save('b', np.array([b]))
```

1.5.3 Evaluating

Another set of 12,500 images is provided separately. We can use it the evaluate the trained perceptron. Since the true label of each test image is not provided, we can only conduct predictions and display images.

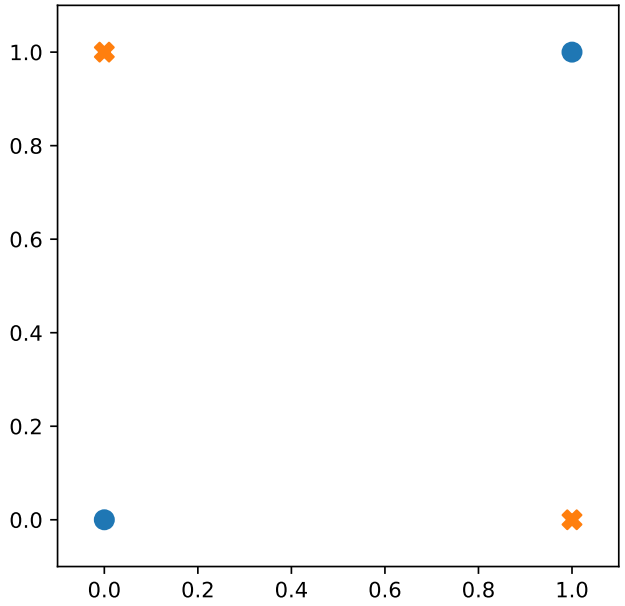
```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def s(z):
5      if z <= 0:
6          return 0
7      return 1
8
9  # load weights and bias
10 w = np.load("w.npy")
11 b = np.load("b.npy")
12 b = b[0]
13
14 # load the test set prepared with the same setting
```

```
15 X_test = np.load("catdog_test.npy")
16
17 n_rows, n_cols = 3, 3
18 fig, axs = plt.subplots(n_rows, n_cols)
19 axs = axs.ravel()
20 for i in range(n_rows*n_cols):
21     # display an image
22     im = X_test[i].reshape((64, 64, 3))
23     axs[i].imshow(im)
24     # predict X_test[i]
25     y_hat = s(np.dot(X_test[i], w) + b)
26     if y_hat == 0:
27         axs[i].set_title("predicted output: cat")
28     else:
29         axs[i].set_title("predicted output: dog")
30 plt.show()
```

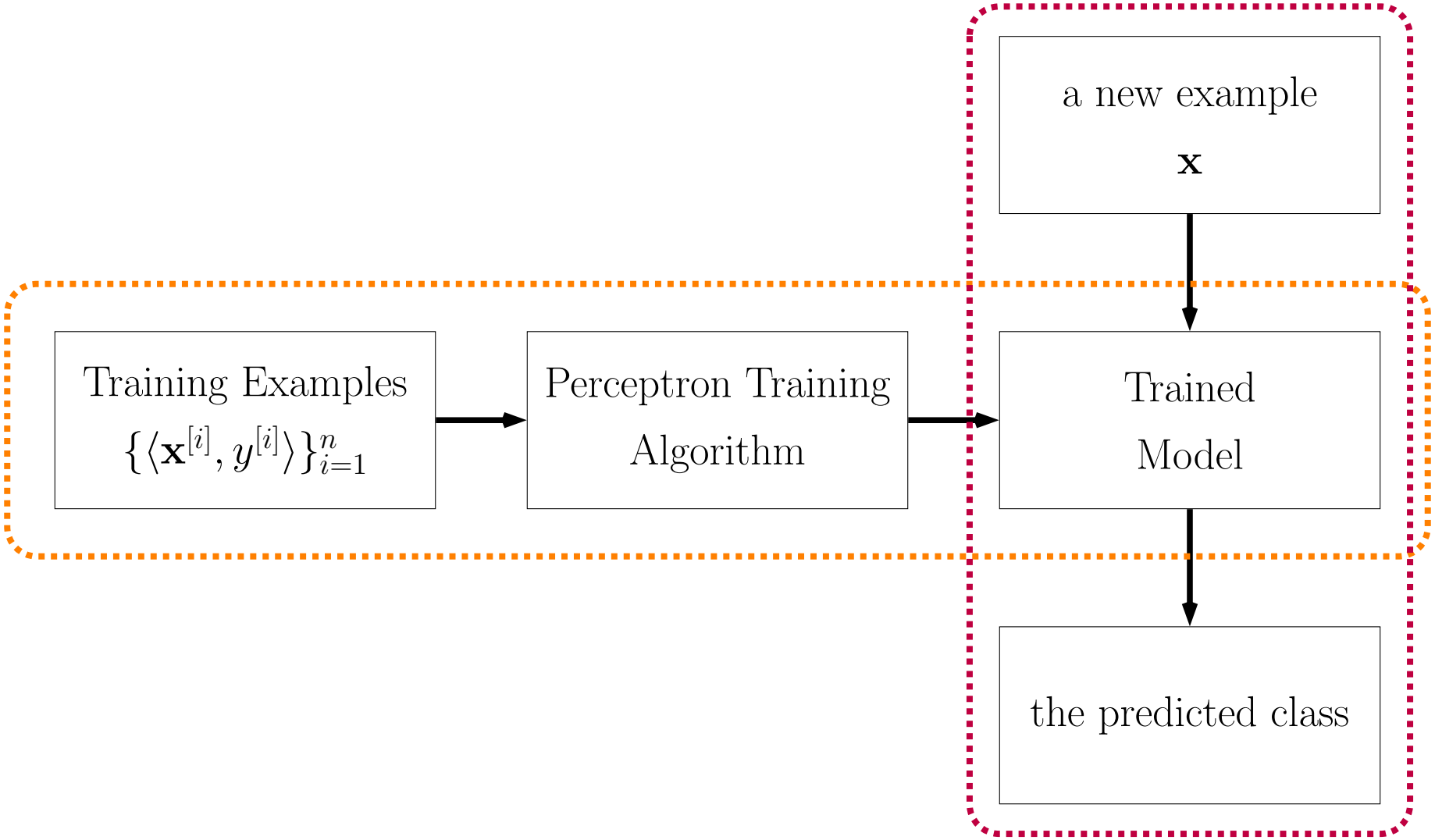

predicted output: cat
predicted output: dog
predicted output: dog
predicted output: dog
predicted output: dog
predicted output: cat
predicted output: cat
predicted output: cat
predicted output: cat

1.6 Limitation of Single-layer Perceptron

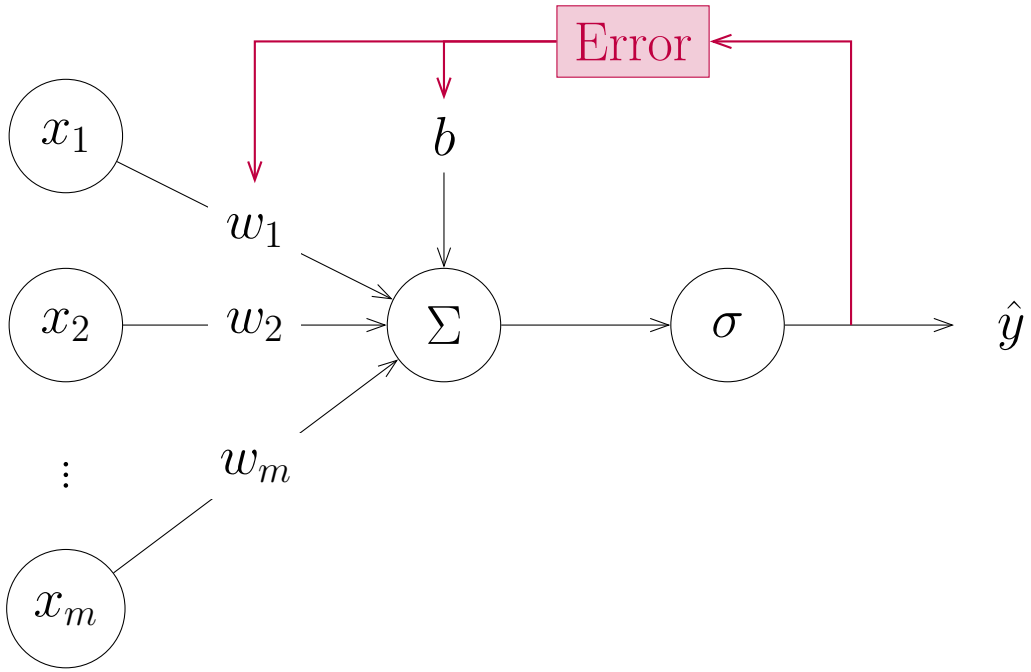
A single-layer perceptron represents a linear classifier. Therefore, it is capable of learning *linearly separable* patterns. One of the non-linearly separable examples since the early development of perceptron is the XOR gate.



1.7 Summary


a new example
$\mathbf{x}$
Training Examples
$\{\langle \mathbf{x}^{[i]}, y^{[i]} \rangle\}_{i=1}^{n}$
Perceptron Training Algorithm
Trained Model
the predicted class

Perceptron Training



Chapter 2

Gradient Descent and Multilayer Perceptron

2.1 Gradient Descent

Gradient descent is an optimization technique using the gradient of an objective function to find the parameter values that minimizes or maximizes the objective function. This technique works in a similar manner as a local search algorithm, but it works in a continuous space.

The gredient of the objective function, ∇f, gives the magnitude and the direction of the steepest slope.

Given a function $f(w_0, w_1, w_2)$, we have

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial w_0} & \frac{\partial f}{\partial w_1} & \frac{\partial f}{\partial w_2} \end{bmatrix}^\top \tag{2.1}$$

The steepest-descent hill climbing is performed by updating the current value with

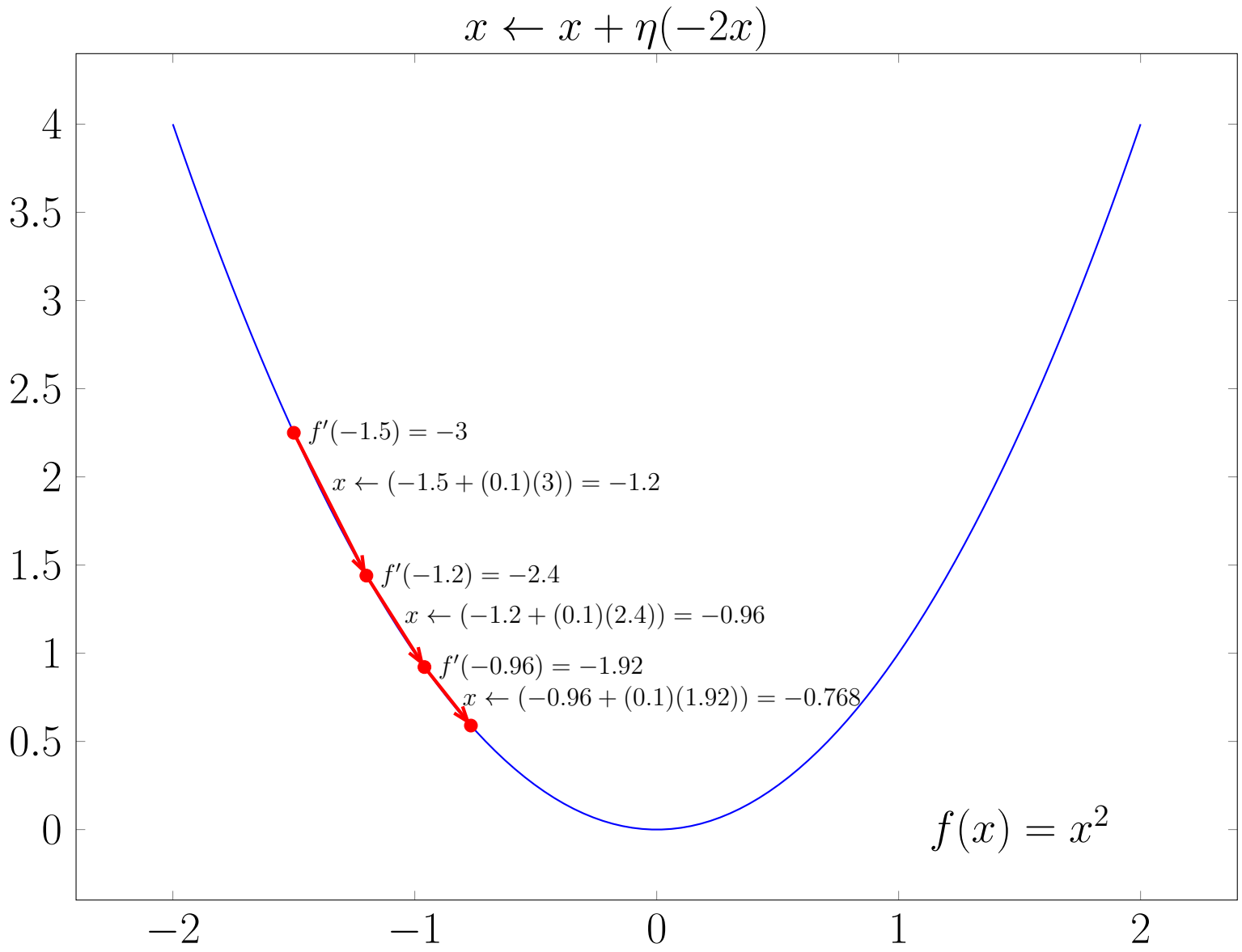
$$\mathbf{w} \leftarrow \mathbf{w} + \eta(-\nabla f\left(\mathbf{w}\right)) \tag{2.2}$$

where $\mathbf{w} = [w_0 \quad w_1 \quad w_2]^\top$, and η is a small positive constant called the *step size* or the *learning rate*, e.g. 0.01, 0.001, etc.

Example 2.1. Given a function $f(x) = x^2$, we can find the value of x that minimizes this function by

1. Start from a random value x_0, e.g. -1.5, and let $\eta = 0.1$.
2. Let $i \leftarrow 0$
3. Compute the gradient, i.e. $\nabla f(x) = 2x$.
4. Repeat
   (a) $x_{i+1} \leftarrow x_i + \eta(-\nabla f(x_i))$
   (b) $d \leftarrow f(x_i) - f(x_{i+1})$
   (c) $i \leftarrow i + 1$

   Until $d < \varepsilon$
5. Output x_i


$x \leftarrow x + \eta(-2x)$
4
3.5
3
2.5
2
1.5
1
0.5
0
−2
−1
0
1
2
$f'(-1.5) = -3$
$x \leftarrow (-1.5 + (0.1)(3)) = -1.2$
$f'(-1.2) = -2.4$
$x \leftarrow (-1.2 + (0.1)(2.4)) = -0.96$
$f'(-0.96) = -1.92$
$x \leftarrow (-0.96 + (0.1)(1.92)) = -0.768$
$f(x) = x^2$

2.2 Gradient Descent and Perceptron

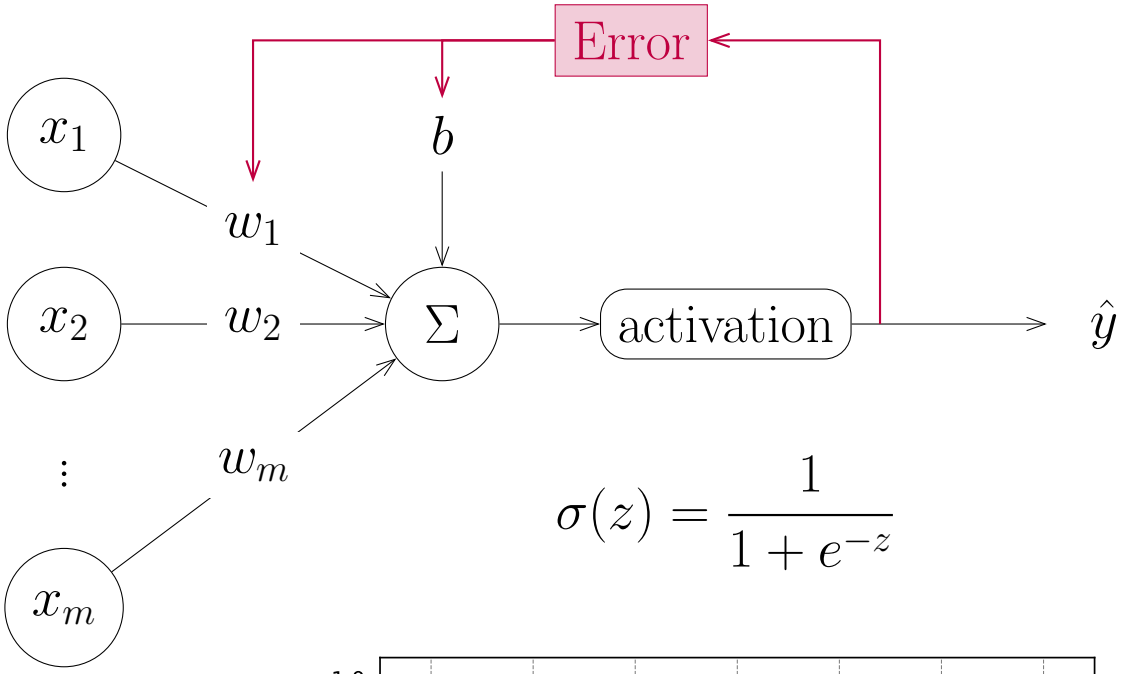
A model similar to the original perceptron is created using the *sigmoid* function as the activation function.

$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad \sigma(z) = \begin{cases} 0 & z \leq 0 \\ 1 & z > 0 \end{cases} \tag{2.3}$$

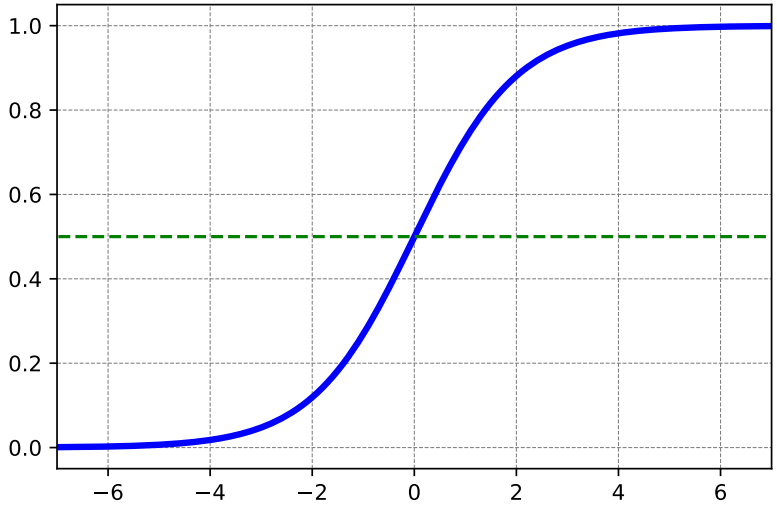
The sigmoid function maps from any real value into a value between 0 and 1.

To generate a binary output, a threshold function is needed to convert an output from the sigmoid function into a predicted output.

$$t(p) = \begin{cases} 1 & p \geq 0.5 \\ 0 & p < 0.5 \end{cases} \tag{2.4}$$



$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



2.2.1 Objective Function

To train a perceptron using the gradient descent technique, we need an objective function. Since the objective is to minimize prediction error, we can define a *mean squared error loss* by aggregating the squared loss over the training set:

$$\mathcal{L}_{mse}(\mathbf{w}, b) = \frac{1}{2n} \sum_{i=1}^{n} (\hat{y}^{[i]} - y^{[i]})^2 \tag{2.5}$$

Note that the scale of $\frac{1}{2}$ is added for convenience.

2.2.2 Training Perceptron using Batch Gradient Descent

(1) initialize $\mathbf{w} \leftarrow 0^m = [0 \quad 0 \quad \dots \quad 0]^\top, b = 0,$

(2) for every training epoch:

(2.1) for every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$:

(2.1.1) calculate $\hat{y}^{[i]} = \sigma\left(\mathbf{w}^\top \mathbf{x}^{[i]} + b\right)$

(2.2) calculate $\nabla_{\mathbf{w}} \mathcal{L}$ and $\nabla_b \mathcal{L}$

(2.3) update $\mathbf{w} \leftarrow \mathbf{w} + \eta(-\nabla_{\mathbf{w}} \mathcal{L})$ and $b \leftarrow b + \eta(-\nabla_b \mathcal{L})$

where $\nabla_{\mathbf{w}} \mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, \dots, \frac{\partial \mathcal{L}}{\partial w_m}\right]^\top$, and $\nabla_b \mathcal{L} = \frac{\partial \mathcal{L}}{\partial b}$

From $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, we have

$$
\begin{aligned}
\frac{\partial \mathcal{L}_{mse}}{\partial w_j} &= \frac{\partial}{\partial w_j} \left[\frac{1}{2n} \sum_{i=1}^{n} \left(\hat{y}^{[i]} - y^{[i]} \right)^2 \right] \\
&= \frac{1}{2n} \sum_{i=1}^{n} \frac{\partial}{\partial w_j} \left[\left(\hat{y}^{[i]} - y^{[i]} \right)^2 \right] \\
&= \frac{1}{2n} \sum_{i=1}^{n} \left[2 \left(\hat{y}^{[i]} - y^{[i]} \right) \right] \frac{\partial}{\partial w_j} \left(\hat{y}^{[i]} - y^{[i]} \right) \\
&= \frac{2}{2n} \sum_{i=1}^{n} \left(\hat{y}^{[i]} - y^{[i]} \right) \frac{\partial}{\partial w_j} \left(\hat{y}^{[i]} - y^{[i]} \right) \\
&= \frac{1}{n} \sum_{i=1}^{n} \left(\hat{y}^{[i]} - y^{[i]} \right) \left(\frac{\partial \hat{y}^{[i]}}{\partial w_j} - \frac{\partial y^{[i]}}{\partial w_j} \right) \\
&= \frac{1}{n} \sum_{i=1}^{n} \left(\hat{y}^{[i]} - y^{[i]} \right) \frac{\partial \hat{y}^{[i]}}{\partial w_j} \\
&= \frac{1}{n} \sum_{i=1}^{n} \left(\hat{y}^{[i]} - y^{[i]} \right) \frac{\partial \sigma(\mathbf{w}^\top \mathbf{x}^{[i]} + b)}{\partial w_j}
\end{aligned}
$$

$$
\begin{aligned}
&= \frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}^{[i]} - y^{[i]}\right)\left(\sigma'(\mathbf{w}^\top \mathbf{x}^{[i]} + b)\right)\frac{\partial(\mathbf{w}^\top \mathbf{x}^{[i]} + b)}{\partial w_j} \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}^{[i]} - y^{[i]}\right)\left((\hat{y}^{[i]})(1-\hat{y}^{[i]})\right)\frac{\partial(\mathbf{w}^\top \mathbf{x}^{[i]} + b)}{\partial w_j} \\
&= \frac{1}{n}\sum_{i=1}^{n}(\hat{y}^{[i]} - y^{[i]})(\hat{y}^{[i]})(1-\hat{y}^{[i]})(x_j^{[i]}) \qquad (2.6)
\end{aligned}
$$

Similarly,

$$
\frac{\partial \mathcal{L}_{mse}}{\partial b} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}^{[i]} - y^{[i]})(\hat{y}^{[i]})(1-\hat{y}^{[i]})(1) \qquad (2.7)
$$

Example 2.2. From the following training set, calculate $\mathbf{w}, b$ for 1 epoch. Initially, $\mathbf{w} = [0.2, 0.1]^\top, b = -0.1$.

$$\mathcal{D} = \left\{([0,0]^\top, 0), ([0,1]^\top, 1), ([1,0]^\top, 1), ([1,1]^\top, 1)\right\}$$

(2.1) For every $(\mathbf{x}^{[i]}, y^{[i]}) \in \mathcal{D}$, calculate $\hat{y}^{[i]}$

$$\begin{aligned}
\hat{y}^{[1]} &= \sigma(0 \cdot 0.2 + 0 \cdot 0.1 - 0.1) = \sigma(-0.1) = 0.4750 \\
\hat{y}^{[2]} &= \sigma(0 \cdot 0.2 + 1 \cdot 0.1 - 0.1) = \sigma(0.0) \quad = 0.5000 \\
\hat{y}^{[3]} &= \sigma(1 \cdot 0.2 + 0 \cdot 0.1 - 0.1) = \sigma(0.1) \quad = 0.5250 \\
\hat{y}^{[4]} &= \sigma(1 \cdot 0.2 + 1 \cdot 0.1 - 0.1) = \sigma(0.2) \quad = 0.5498
\end{aligned}$$

(2.2) Calculate $\nabla_{\mathbf{w}}\mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}\right]^\top$ and $\nabla_b \mathcal{L} = \frac{\partial \mathcal{L}}{\partial b}$

$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}^{[i]} - y^{[i]})(\hat{y}^{[i]})(1 - \hat{y}^{[i]})(x_j^{[i]})$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_1} = [&(0.4750 - 0)(0.4750)(1 - 0.4750)(0) + \\ &(0.5000 - 1)(0.5000)(1 - 0.5000)(0) + \\ &(0.5250 - 1)(0.5250)(1 - 0.5250)(1) + \\ &(0.5498 - 1)(0.5498)(1 - 0.5498)(1)]/4 = -0.0575\end{aligned}$$

From $\hat{y}^{[1]} = 0.4750, \quad \hat{y}^{[2]} = 0.5000, \quad \hat{y}^{[3]} = 0.5250, \quad \hat{y}^{[4]} = 0.5498,$

$$\frac{\partial \mathcal{L}}{\partial w_2} =$$

From $\hat{y}^{[1]} = 0.4750, \quad \hat{y}^{[2]} = 0.5000, \quad \hat{y}^{[3]} = 0.5250, \quad \hat{y}^{[4]} = 0.5498,$

$$\frac{\partial \mathcal{L}}{\partial b} =$$

(2.3) Update $\mathbf{w}$ and b using $\eta = 0.1$,

$$\begin{aligned} \mathbf{w} &\leftarrow \begin{bmatrix} 0.2 \\ 0.1 \end{bmatrix} + (0.1)\left(- \begin{bmatrix} -0.0575 \\ -0.0591 \end{bmatrix} \right) && = \begin{bmatrix} 0.2058 \\ 0.1059 \end{bmatrix} \\ b &\leftarrow (-0.1) + (0.1)(-(-0.0591)) && = -0.0941 \end{aligned}$$

2.2.3 Implementation of Batch Gradient Descent

```
1  import numpy as np
2  np.set_printoptions(floatmode='fixed', precision=4, suppress=True)
3
4  def sigmoid(z):
5      return 1/(1+np.exp(-z))
6
7  X = np.array([[0,0], [0,1], [1,0], [1,1]])
8  y = np.array([0, 1, 1, 1])
9
10 w = np.array([0.2, 0.1])
11 b = -0.1
12
13 n = X.shape[0]
14 eta = 0.1
15
16 for i in range(5):
17     print(f"Epoch {i+1}:")
```

```
18      y_hat = sigmoid(np.matmul(X, w) + b)
19      loss = np.sum((y_hat - y)**2)/(2*n)
20      print(f"y_hat = {y_hat}, loss = {loss:.4f}")
21
22      grad_w = np.matmul((y_hat - y)*(y_hat)*(1-y_hat), X)/n
23      grad_b = np.sum((y_hat - y)*(y_hat)*(1-y_hat))/n
24      print(f"grad_w = {grad_w}, grad_b = {grad_b:.4f}")
25
26      w = w + eta*(-grad_w)
27      b = b + eta*(-grad_b)
28      print(f"w = {w}, b = {b:.4f}")
29
30      print()
31  y_hat = sigmoid(np.matmul(X, w) + b)
32  print(f"predicted class = {np.round(y_hat)}")
```

```
Epoch 1:
y_hat = [0.4750 0.5000 0.5250 0.5498], loss = 0.1130
grad_w = [-0.0575 -0.0591], grad_b = -0.0591
w = [0.2057 0.1059], b = -0.0941

Epoch 2:
y_hat = [0.4765 0.5030 0.5279 0.5542], loss = 0.1120
grad_w = [-0.0570 -0.0586], grad_b = -0.0583
w = [0.2114 0.1118], b = -0.0883

...

Epoch 5:
y_hat = [0.4808 0.5116 0.5364 0.5669], loss = 0.1090
grad_w = [-0.0554 -0.0571], grad_b = -0.0559
w = [0.2282 0.1290], b = -0.0712

predicted class = [0.0000 1.0000 1.0000 1.0000]
```

2.2.4 Stochastic Gradient Descent

The standard gradient descent calculates the gradient for the entire training set. This process is time consuming and requires much memory.
Stochastic gradient descent calculates a gradient from one labeled example.

(1) initialize $\mathbf{w} \leftarrow 0^m = [0 \quad 0 \quad \dots \quad 0]^\top, b = 0,$

(2) for every training epoch:

(2.1) shuffle $\mathcal{D}$

(2.2) for every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$:

(2.2.1) calculate $\hat{y}^{[i]} = \sigma\left(\mathbf{x}^{[i]\top}\mathbf{w} + b\right)$

(2.2.2) calculate $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{x}^{[i]}, y^{[i]})$ and $\nabla_b\mathcal{L}(\mathbf{x}^{[i]}, y^{[i]})$

(2.2.3) update $\mathbf{w} \leftarrow \mathbf{w} - \eta\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{x}^{[i]}, y^{[i]})$ and $b \leftarrow b - \eta\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{x}^{[i]}, y^{[i]})$

We define a loss function

$$\mathcal{L}_{mse}(\mathbf{x}^{[i]}, y^{[i]}; \mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{[i]} - y^{[i]}\right)^2$$

$$\begin{aligned}\frac{\partial}{\partial w_j}\left(\mathcal{L}_{mse}(\mathbf{x}^{[i]}, y^{[i]}; \mathbf{w}, b)\right) &= \frac{\partial}{\partial w_j}\left(\frac{1}{2}\left(\hat{y}^{[i]} - y^{[i]}\right)^2\right) \\ &= \left(\hat{y}^{[i]} - y^{[i]}\right)\left(\hat{y}^{[i]}\right)\left(1 - \hat{y}^{[i]}\right)\left(x_j^{[i]}\right)\end{aligned}$$

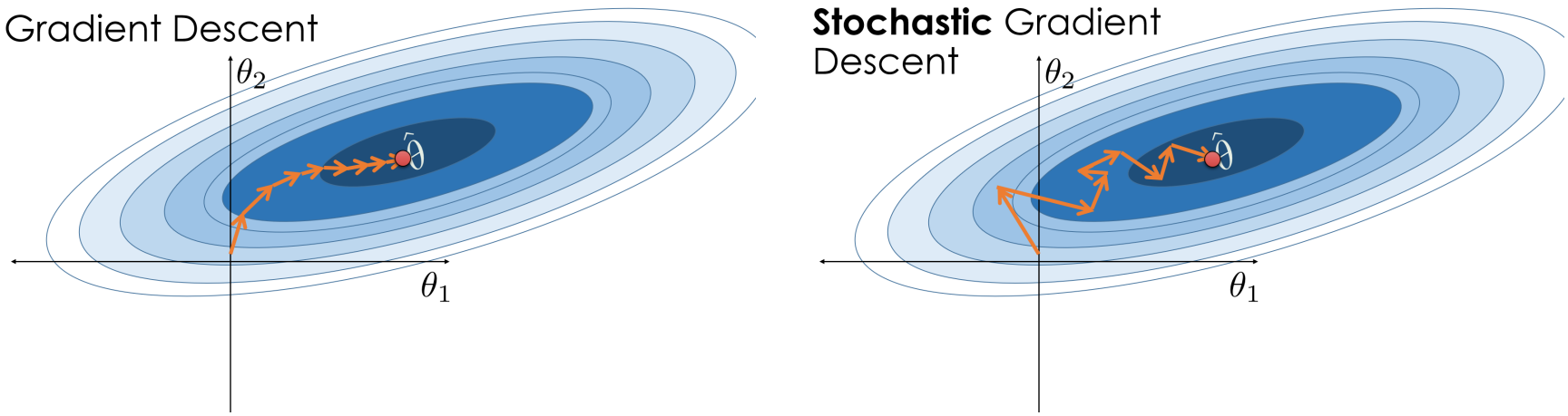
Therefore, each weight w_j is updated by

$$w_j \leftarrow w_j - \eta\left(\hat{y}^{[i]} - y^{[i]}\right)\left(\hat{y}^{[i]}\right)\left(1 - \hat{y}^{[i]}\right)\left(x_j^{[i]}\right)$$

The bias is updated by

$$b \leftarrow b - \eta\left(\hat{y}^{[i]} - y^{[i]}\right)\left(\hat{y}^{[i]}\right)\left(1 - \hat{y}^{[i]}\right)$$

Behavior of Stochastic Gradient Descent



Source: `https://www.textbook.ds100.org/ch/20/gradient_stochastic.html`

- Batch Gradient Descent (BGD) smoothly updates the parameters since the gradient is calculated from the entire dataset.
- Stochastic Gradient Descent (SGD) noisily updates the parameters, but it updates more often.

2.2.5 Minibatch Stochastic Gradient Descent

(1) initialize $\mathbf{w} \leftarrow 0^m = [0 \quad 0 \quad \dots \quad 0]^\top, b = 0,$

(2) for every training epoch:

(2.1) shuffle $\mathcal{D}$

(2.2) for every $\left\{ \langle \mathbf{x}^{[i]}, y^{[i]} \rangle, \dots, \langle \mathbf{x}^{[i+k-1]}, y^{[i+k-1]} \rangle \right\} \subset \mathcal{D}$:

(2.2.1) calculate $\hat{y}^{[i]} = \sigma\left(\mathbf{w}^\top \mathbf{x}^{[i]} + b\right)$

(2.3) calculate $\nabla_{\mathbf{w}} \mathcal{L}$ and $\nabla_b \mathcal{L}$

$$\nabla_{\mathbf{w}} \mathcal{L} = \frac{1}{k} \sum_{j=0}^{k-1} \left(\hat{y}^{[i+j]} - y^{[i+j]}\right) \left(\hat{y}^{[i+j]}\right) \left(1 - \hat{y}^{[i+j]}\right) \mathbf{x}^{[i+j]}$$

$$\nabla_b \mathcal{L} = \frac{1}{k} \sum_{j=0}^{k-1} \left(\hat{y}^{[i+j]} - y^{[i+j]}\right) \left(\hat{y}^{[i+j]}\right) \left(1 - \hat{y}^{[i+j]}\right)$$

(2.4) update $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{L}$ and $b \leftarrow b - \eta \nabla_b \mathcal{L}$

Minibatch stochastic gradient descent is between BGD and SGD by calculating gradient from a set of k examples. It improves the calculation speed and reduce gradient variance.

2.3 Cross-Entropy Loss Function

The sigmoid function converts a weighted sum (a real number) into a probability.

$$P(y|\mathbf{x}) = \begin{cases} \sigma(\mathbf{w}^\top \mathbf{x} + b) & \text{if } y = 1 \\ 1 - \sigma(\mathbf{w}^\top \mathbf{x} + b) & \text{if } y = 0 \end{cases}$$

It can be rewritten as

$$P(y|\mathbf{x}) = \big(\sigma(\mathbf{w}^\top \mathbf{x} + b)\big)^y \big(1 - \sigma(\mathbf{w}^\top \mathbf{x} + b)\big)^{1-y}$$

We are therefore looking for the parameters that maximizes the probability:

$$\begin{aligned} P(\mathbf{y}|\mathbf{x};\mathbf{w};b) &= \prod_{i=1}^{n} P(y^{[i]}|\mathbf{x}^{[i]};\mathbf{w};b) \\ &= \prod_{i=1}^{n} \Big(\sigma(\mathbf{w}^\top \mathbf{x}^{[i]} + b)\Big)^{y^{[i]}} \Big(1 - \sigma(\mathbf{w}^\top \mathbf{x}^{[i]} + b)\Big)^{(1-y^{[i]})} \end{aligned} \tag{2.8}$$

It is easier to maximize the log of the probability:

$$\begin{aligned} \log P(\mathbf{y}|\mathbf{x};\mathbf{w};b) &= \sum_{i=1}^{n} \Big[y^{[i]} \log(\sigma(\mathbf{w}^\top \mathbf{x}^{[i]} + b)) + (1 - y^{[i]}) \log(1 - \sigma(\mathbf{w}^\top \mathbf{x}^{[i]} + b))\Big] \\ &= \sum_{i=1}^{n} \Big[y^{[i]} \log(\hat{y}^{[i]}) + (1 - y^{[i]}) \log(1 - \hat{y}^{[i]})\Big] \end{aligned} \tag{2.9}$$

This is called the *log-likelihood* function.

It is more convenient to minimize *negative log-likelihood*. A new loss function can be defined:

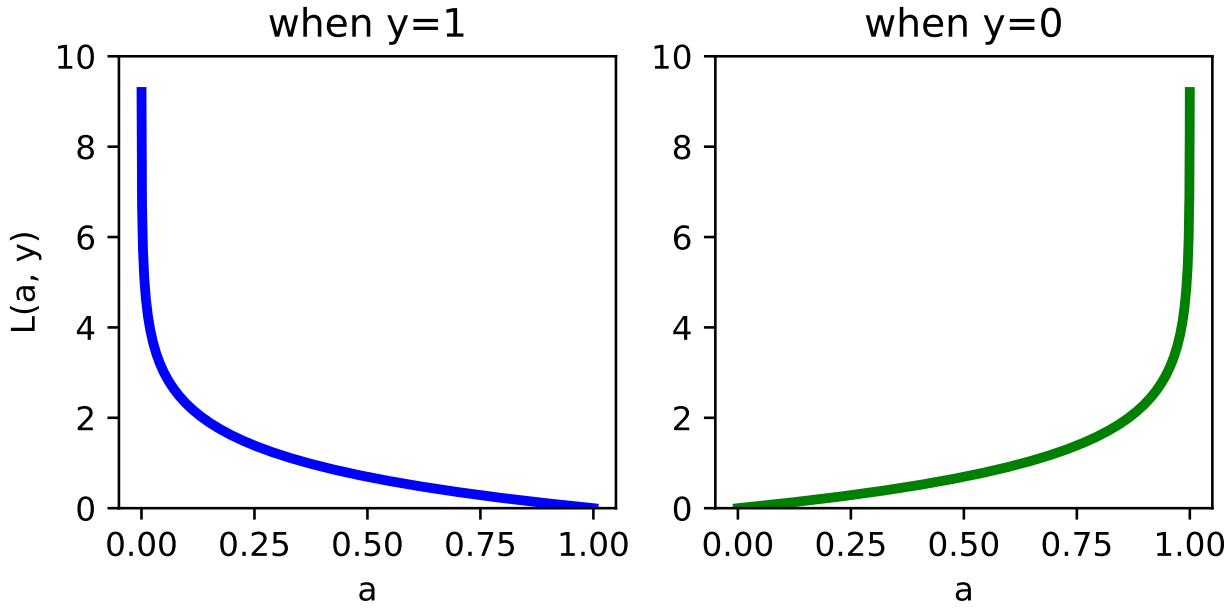
$$\mathcal{L}_{log}(\mathbf{w}, b) = -\frac{1}{n}\sum_{i=1}^{n}\left[y^{[i]}\log(\hat{y}^{[i]}) + (1-y^{[i]})\log(1-\hat{y}^{[i]})\right] \tag{2.10}$$

This loss function is also called the *cross-entropy loss function*.

Given an input-output pair $(\mathbf{x}, y)$,

$$\mathcal{L}_{log}(\mathbf{x}, y; \mathbf{w}, b) = \begin{cases} -\log(\hat{y}) & \text{if } y = 1 \\ -\log(1 - \hat{y}) & \text{if } y = 0 \end{cases} \tag{2.11}$$

where $\hat{y} = \sigma(\mathbf{x}^\top \mathbf{w} + b)$.



Gradient of the cross-entropy loss

$$
\begin{aligned}
\frac{\partial \mathcal{L}_{log}}{\partial w_j} &= \frac{\partial}{\partial w_j}\Big(- \big(y \log(\hat{y}) + (1-y)\log(1-\hat{y})\big)\Big) \\
&= -\left(\frac{\partial}{\partial w_j}\big(y \log(\hat{y})\big) + \frac{\partial}{\partial w_j}\big((1-y)\log(1-\hat{y})\big)\right) \\
&= -\left(y\frac{\partial}{\partial w_j}\big(\log(\hat{y})\big) + (1-y)\frac{\partial}{\partial w_j}\big(\log(1-\hat{y})\big)\right) \\
&= -\left(\Big(y\Big)\Big(\frac{1}{\hat{y}}\Big)\Big(\frac{\partial}{\partial w_j}(\hat{y})\Big) + \Big(1-y\Big)\Big(\frac{1}{1-\hat{y}}\Big)\Big(\frac{\partial}{\partial w_j}(1-\hat{y})\Big)\right) \\
&= -\Big(\Big(y\Big)\Big(\frac{1}{\hat{y}}\Big)\Big(\frac{\partial}{\partial w_j}\big(\sigma(\mathbf{w}^\top \mathbf{x} + b)\big)\Big) + \\
&\qquad \Big(1-y\Big)\Big(\frac{1}{1-\hat{y}}\Big)\Big(\frac{\partial}{\partial w_j}\big(1-\sigma(\mathbf{w}^\top \mathbf{x} + b)\big)\Big)\Big) \\
&= -\Big(\Big(y\Big)\Big(\frac{1}{\hat{y}}\Big)\Big(\hat{y}\Big)\Big(1-\hat{y}\Big)\Big(\frac{\partial}{\partial w_j}\big(\mathbf{w}^\top \mathbf{x} + b\big)\Big) + \\
&\qquad \Big(1-y\Big)\Big(\frac{1}{1-\hat{y}}\Big)\Big[-\Big(\hat{y}\Big)\Big(1-\hat{y}\Big)\Big(\frac{\partial}{\partial w_j}\big(\mathbf{w}^\top \mathbf{x} + b\big)\Big)\Big]\Big)
\end{aligned}
$$

$$
\begin{aligned}
&= -\Big(\Big(y\Big)\Big(\frac{1}{\hat{y}}\Big)\Big(\hat{y}\Big)\Big(1-\hat{y}\Big)\Big(x_j\Big)+ \\
&\qquad \Big(1-y\Big)\Big(\frac{1}{1-\hat{y}}\Big)\Big[-\Big(\hat{y}\Big)\Big(1-\hat{y}\Big)\Big(x_j\Big)\Big]\Big) \\
&= -\big(\,(y)\,(1-\hat{y})\,(x_j) - (1-y)\,(\hat{y})\,(x_j)\,\big) \\
&= -\big(\,[(y)(1-\hat{y}) - (1-y)(\hat{y})]\,(x_j)\,\big) \\
&= -\big(\,[y - y\hat{y} - \hat{y} + y\hat{y}]\,(x_j)\,\big) \\
&= -\big((y-\hat{y})(x_j)\big)
\end{aligned}
$$

Stochastic Gradient Descent for Sigmoid Unit using Cross-entropy Loss

(1) initialize $\mathbf{w} \leftarrow 0^m = [0 \quad 0 \quad \dots \quad 0]^\top, b = 0,$

(2) for every training epoch:

(2.1) shuffle $\mathcal{D}$

(2.2) for every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$:

(2.2.1) calculate $\hat{y}^{[i]} = \sigma\left(\mathbf{x}^{[i]\top}\mathbf{w} + b\right)$

(2.2.2) calculate

$$\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{x}^{[i]}, y^{[i]}) = -(y^{[i]} - \hat{y}^{[i]})(\mathbf{x}^{[i]})$$
$$\nabla_b\mathcal{L}(\mathbf{x}^{[i]}, y^{[i]}) = -(y^{[i]} - \hat{y})$$

(2.2.3) update

$$\mathbf{w} \leftarrow \mathbf{w} + \eta(y^{[i]} - \hat{y}^{[i]})\mathbf{x}^{[i]}$$
$$b \leftarrow b + \eta(y^{[i]} - \hat{y}^{[i]})$$

2.4 Tensorflow

Tensorflow (`https://www.tensorflow.org`) is a machine learning library developed by Google. Tensorflow provides functions to support operations required by learning algorithms. It implements computation graph and supports automatic differentiation. It also supports general-purpose computation on GPU which drastically improves the training speed.

```
1  import tensorflow as tf
2
3  n = 4
4  # define training examples
5  X = tf.constant([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])
6  y = tf.constant([[0.0], [0.0], [0.0], [1.0]])
7  # define w, b
8  w = tf.Variable([[2.0], [2.0]])
9  b = tf.Variable(-3.3)
10
11 # calculate a = g(x.w + b)
```

```
12 a = tf.math.sigmoid(tf.matmul(X, w) + b)
13
14 # calculate y_hat = t(a)
15 y_hat = tf.round(a)
16 print("Predicted output =")
17 tf.print(y_hat)
18
19 Llog = tf.reduce_sum(-(y*tf.math.log(a) +
20                        (1-y)*tf.math.log(1-a)))/n
21 print("Log loss =", end=" ")
22 tf.print(Llog)
```

```
Predicted output =
[[0]
 [0]
 [0]
 [1]]
Log loss = 0.23035562
```

The gradient descent algorithm can be easily implemented using the automatic differentiation feature.

```
1  import tensorflow as tf
2
3  n = 4
4
5  # define training examples
6  X = tf.constant([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])
7  y = tf.constant([[0.0], [0.0], [0.0], [1.0]])
8
9  # define w, b
10 w = tf.Variable([[0.01], [0.01]])
11 b = tf.Variable(0.01)
12
13 eta = 0.01
14
15 for r in range(10):
16     # define output and loss
17     with tf.GradientTape() as tape:
```

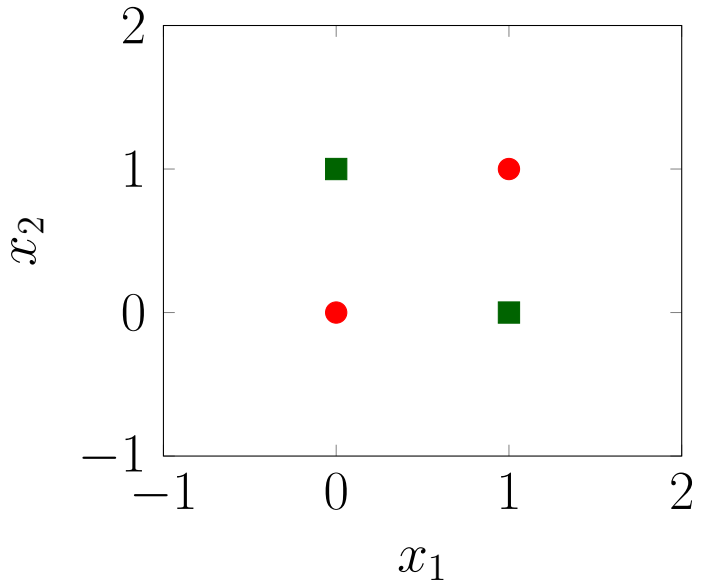
```
18          a = tf.math.sigmoid(tf.matmul(X, w) + b)
19          L = tf.reduce_sum(-(y*tf.math.log(a) +
20                              (1-y)*tf.math.log(1-a)))/n
21
22      # calculate gradients
23      [gra_w, gra_b] = tape.gradient(L, [w, b])
24
25      # update w and b
26      w.assign_add(-eta*gra_w)
27      b.assign_add(-eta*gra_b)
28
29      print("w = [%+.4f %+.4f], b = %+.4f" % (w[0], w[1], b))
30
31  # calculate y_hat = t(a)
32  a = tf.math.sigmoid(tf.matmul(X, w) + b)
33  y_hat = tf.round(a)
34  print("Predicted output =")
35  tf.print(y_hat)
```

```
w = [+0.0100 +0.0100], b = +0.0075
w = [+0.0099 +0.0099], b = +0.0049
w = [+0.0099 +0.0099], b = +0.0024
w = [+0.0099 +0.0099], b = -0.0002
w = [+0.0099 +0.0099], b = -0.0027
w = [+0.0099 +0.0099], b = -0.0052
w = [+0.0098 +0.0098], b = -0.0077
w = [+0.0098 +0.0098], b = -0.0102
w = [+0.0098 +0.0098], b = -0.0127
w = [+0.0098 +0.0098], b = -0.0152
Predicted output =
[[0]
 [0]
 [0]
 [1]]
```

Google Colab already supports Tensorflow and we can also change the *Runtime Type* to use GPU in the computation.

2.5 Multi-layer Perceptron

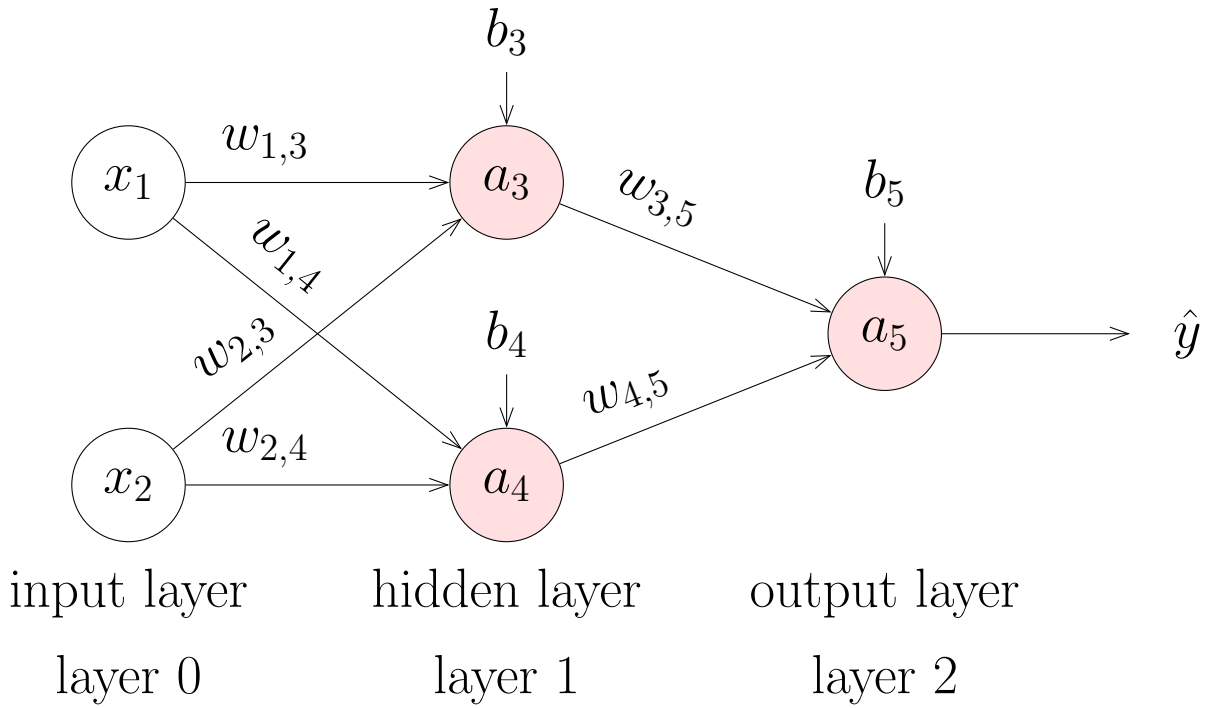
Since a perceptron represents a linear model for classification, it works well on a training set that is **linearly separable**. However, a perceptron cannot properly deal with the training set that cannot be separated by just a line.



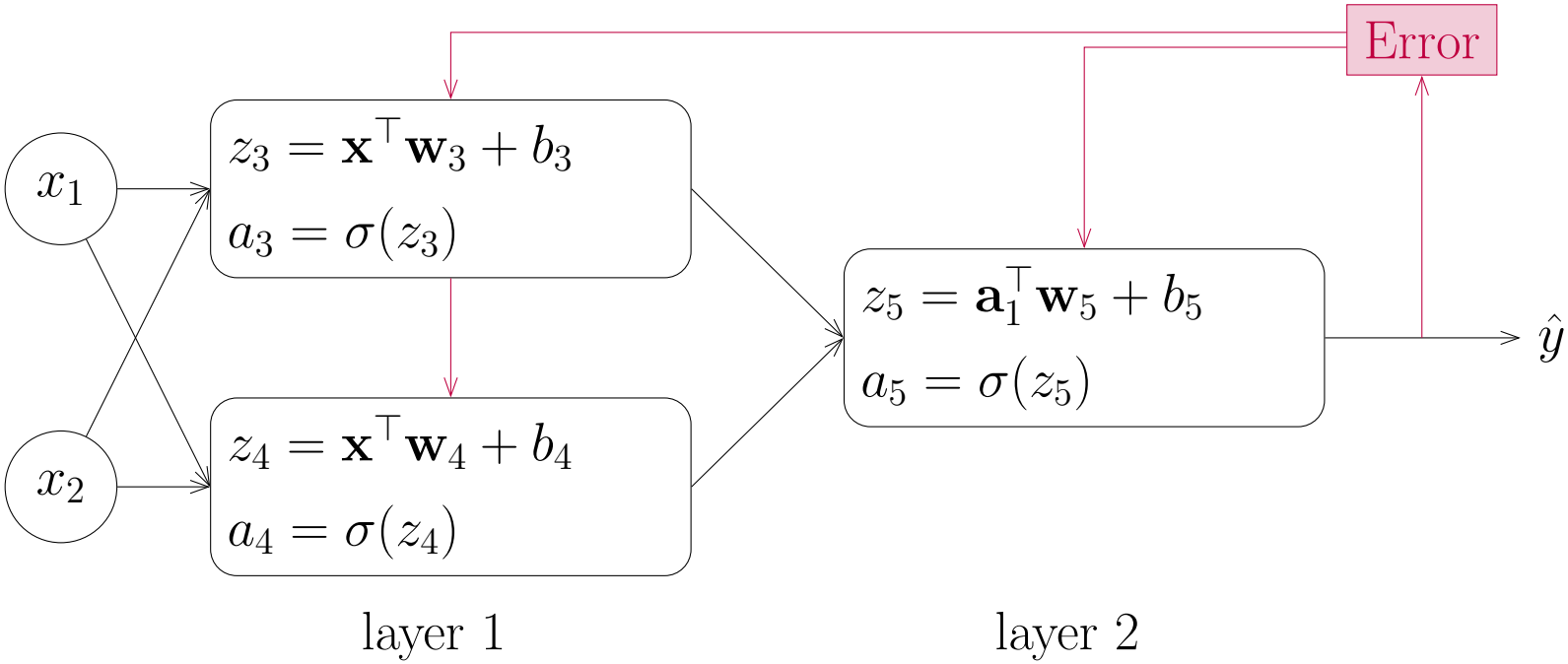
To solve the non-linearly separable problem, we connect multiple units to from a *neural network*.

The units are arranged into layers. The network is also called *multilayer perceptron*.

Example 2.3. Design a multilayer perceptron for the XOR problem.



From the diagram, a_3, a_4, and a_5 are sigmoid units. Layers are fully connected.



where $\mathbf{w}_3 = \begin{bmatrix} w_{1,3} \\ w_{2,3} \end{bmatrix}$, $\mathbf{w}_4 = \begin{bmatrix} w_{1,4} \\ w_{2,4} \end{bmatrix}$, $\mathbf{w}_5 = \begin{bmatrix} w_{3,5} \\ w_{4,5} \end{bmatrix}$, $\mathbf{a}_1 = \begin{bmatrix} a_3 \\ a_4 \end{bmatrix}$.

Therefore,

$$\begin{aligned}
\hat{y} &= a_5 \\
&= \sigma(w_{3,5}a_3 + w_{4,5}a_4 + b_5) \\
&= \sigma(w_{3,5}\sigma(w_{1,3}x_1 + w_{2,3}x_2 + b_3) + w_{4,5}\sigma(w_{1,4}x_1 + w_{2,4}x_2 + b_4) + b_5)
\end{aligned}$$

From the above formulas, an input vector ($\mathbf{x}$) is fed into layer 1 using two linear classifiers to produce two intermediate outputs ($\mathbf{a}_1 = [a_3 \quad a_4]^\top$). The output vector of layer 1 is fed to layer 2 which produces a_5 before passing to the threshold function to generate $\hat{y}$. We can simply consider $\hat{y}$ as a nested function:

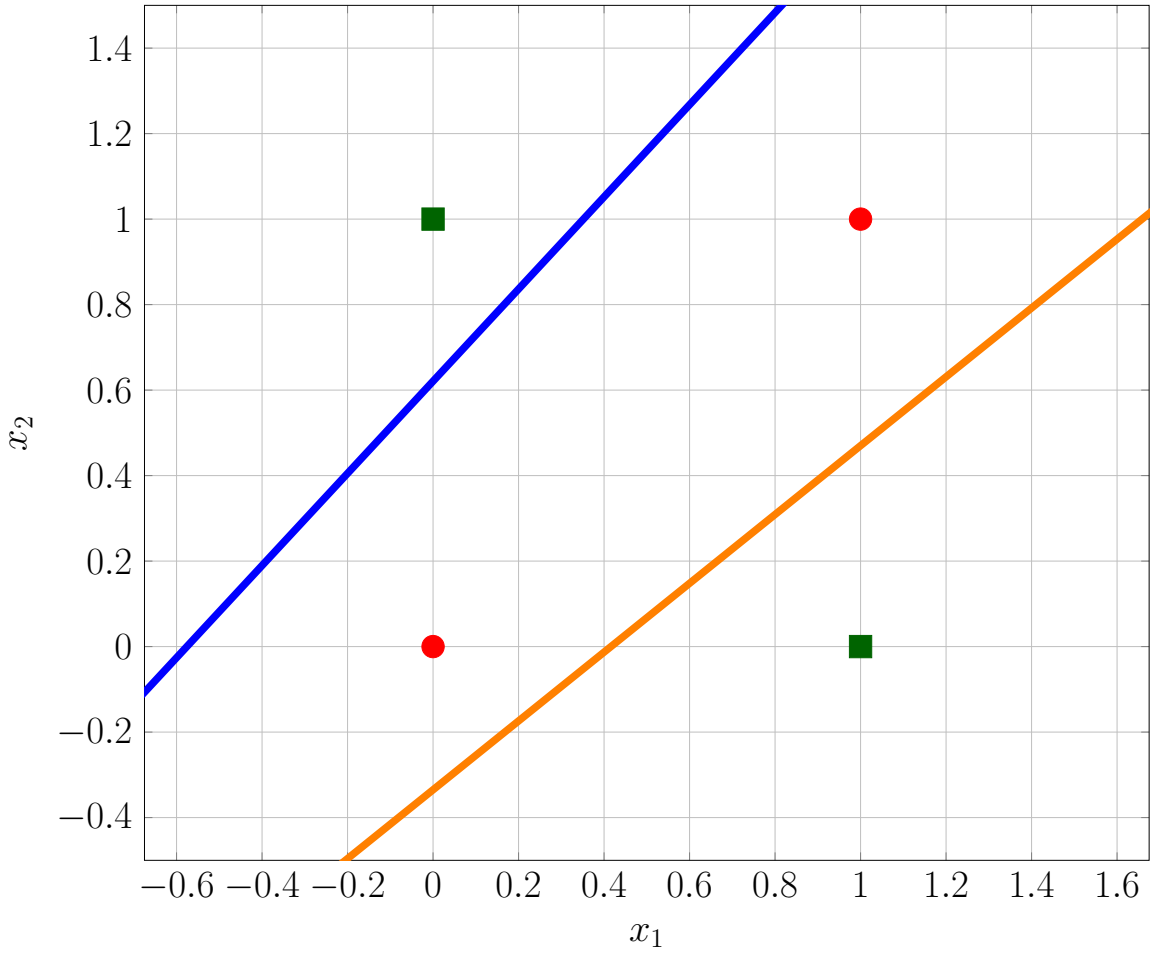
$$\hat{y} = f_2(\boldsymbol{f}_1(\mathbf{x}))$$

where $\boldsymbol{f}_1$ is a vector function representing layer 1, and f_2 is a scalar function representing layer 2.

$$\begin{aligned}
\mathbf{a}_1 = \begin{bmatrix} a_3 \\ a_4 \end{bmatrix} &= \boldsymbol{f}_1(\mathbf{x}) = \sigma\left(\begin{bmatrix} w_{1,3} & w_{2,3} \\ w_{1,4} & w_{2,4} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_3 \\ b_4 \end{bmatrix}\right) \\
\hat{y} = a_5 \quad &= f_2(\mathbf{a}_1) = \sigma\left(\begin{bmatrix} w_{3,5} & w_{4,5} \end{bmatrix}\begin{bmatrix} a_3 \\ a_4 \end{bmatrix} + b_5\right)
\end{aligned}$$

Example 2.4. What is the output of the following network when we input a vector $[0 \quad 1]^\top$? Here, $\mathbf{w}_3 = [1.61 \quad -2.00]^\top$, $b_3 = -0.67$, $\mathbf{w}_4 = [4.16 \quad -3.85]^\top, b_4 = 2.39$, $\mathbf{w}_5 = [3.13 \quad -3.70]^\top, b_5 = 1.70$.

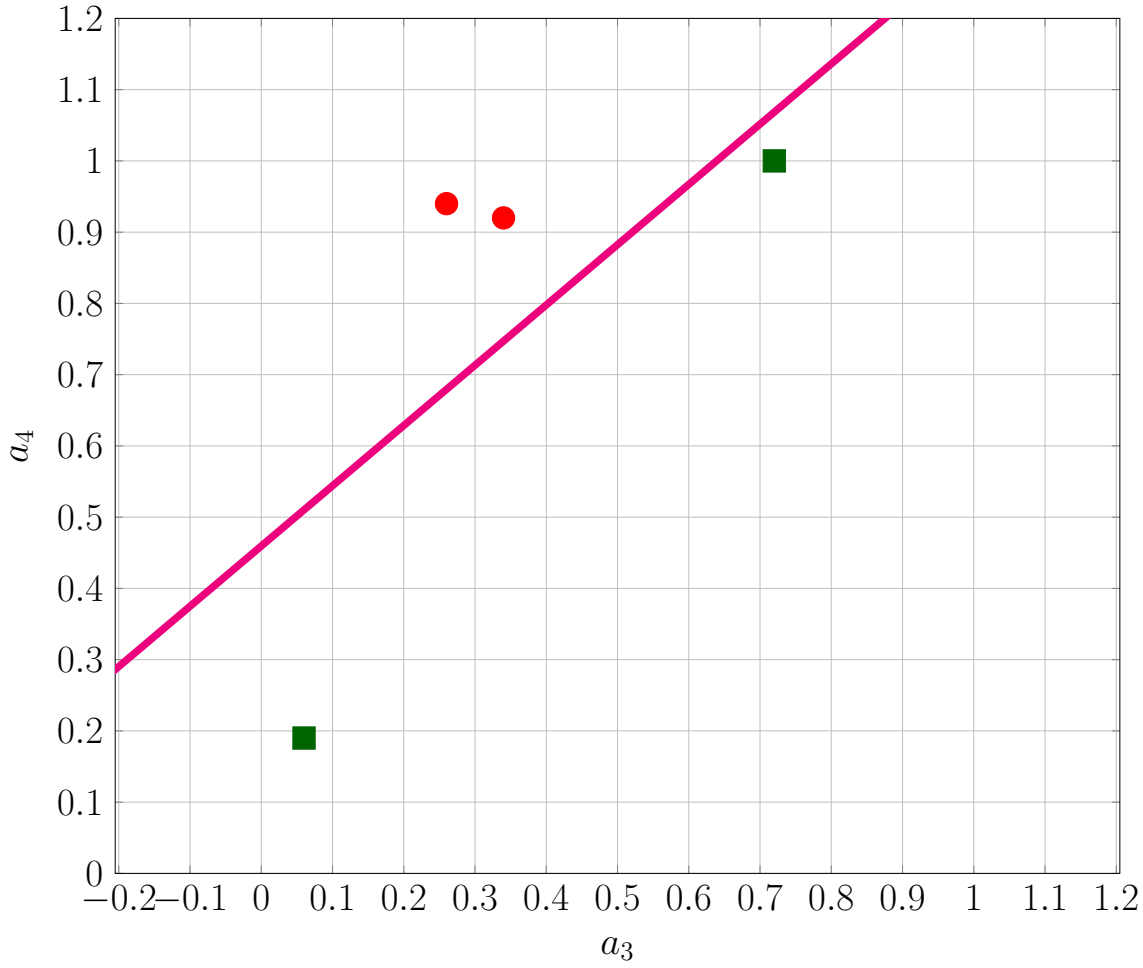
Example 2.5. This is how the multilayer perceptron work for the XOR problem. From $\mathbf{w}_3 = [1.61 \ -2.00]^\top, b_3 = -0.67$, $\mathbf{w}_4 = [4.16 \ -3.85]^\top, b_4 = 2.39$, $\mathbf{w}_5 = [3.13 \ -3.70]^\top, b_5 = 1.70$, the hidden layer (or layer 1) forms two linear classifiers.



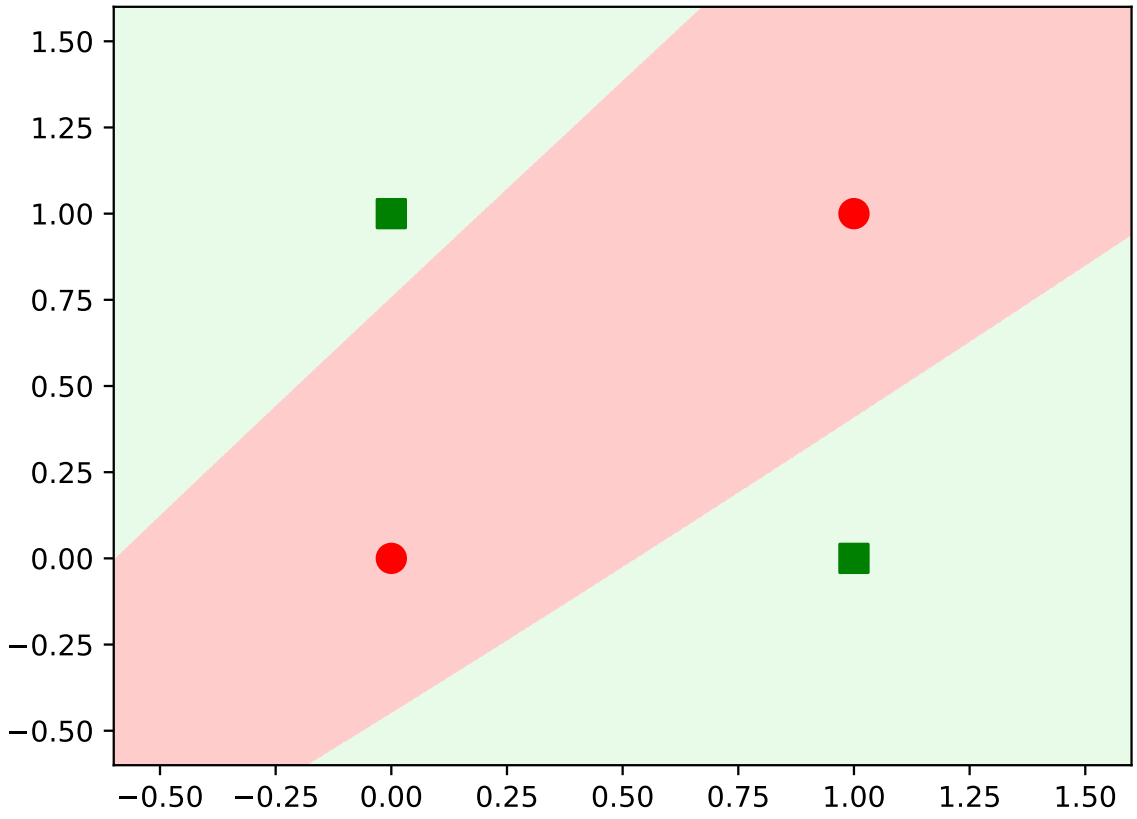
The output of the hidden layer are shown in the following table.

x_1	x_2	a_3	a_4
0	0	0.34	0.92
0	1	0.06	0.19
1	0	0.72	1.00
1	1	0.26	0.94

The output layer then classifies the intermediate outputs.

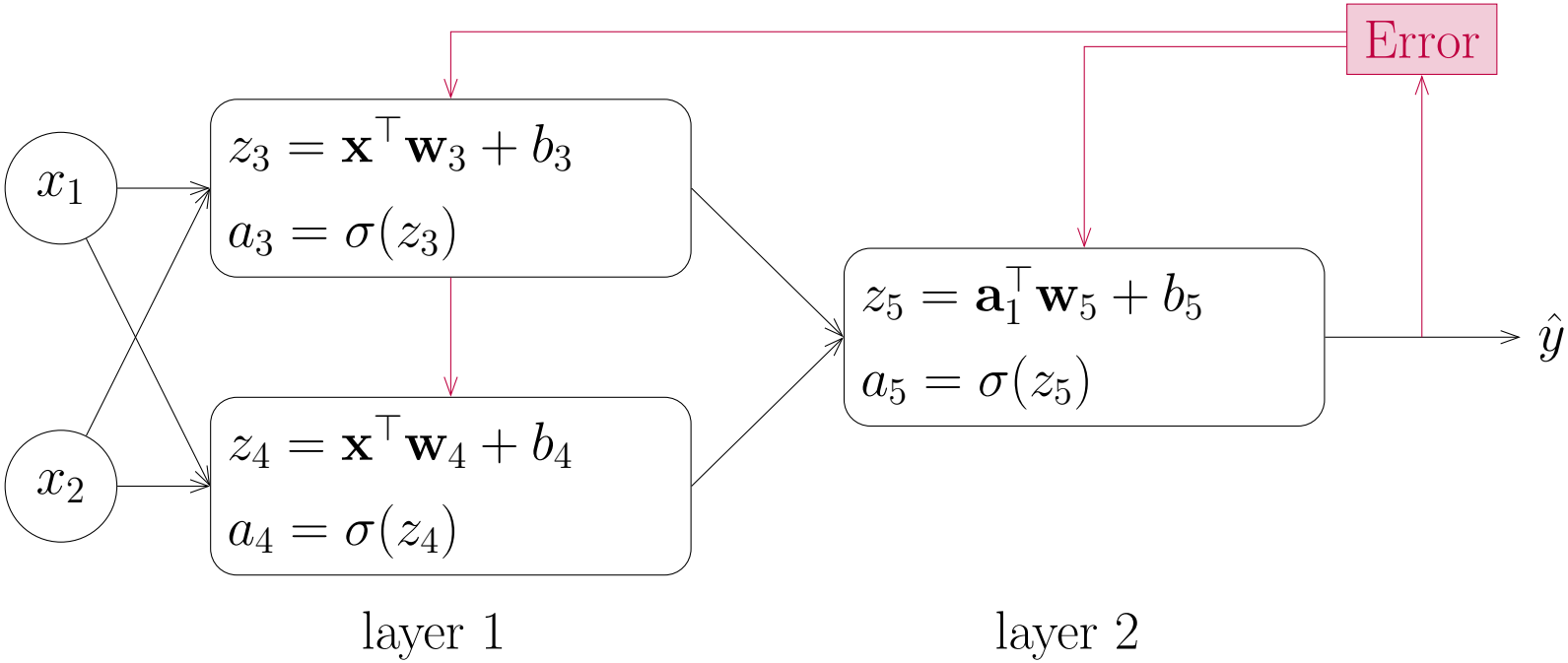


Decision boundary



2.5.1 Training Multilayer Perceptron

The original algorithm for training multilayer perceptron is called *backpropagation*. It is based on the gradient descent and the chain rule.



Given a loss function $\mathcal{L}$ and an input-output pair, we can calculate the gradient for updating the output layer by

$$\frac{\partial \mathcal{L}}{\partial w_{3,5}} = \frac{\partial \mathcal{L}}{\partial a_5} \cdot \frac{\partial a_5}{\partial z_5} \cdot \frac{\partial z_5}{\partial w_{3,5}}$$

$$\frac{\partial \mathcal{L}}{\partial w_{1,3}} = \frac{\partial \mathcal{L}}{\partial a_5} \cdot \frac{\partial a_5}{\partial z_5} \cdot \frac{\partial z_5}{\partial a_3} \cdot \frac{\partial a_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_{1,3}}$$

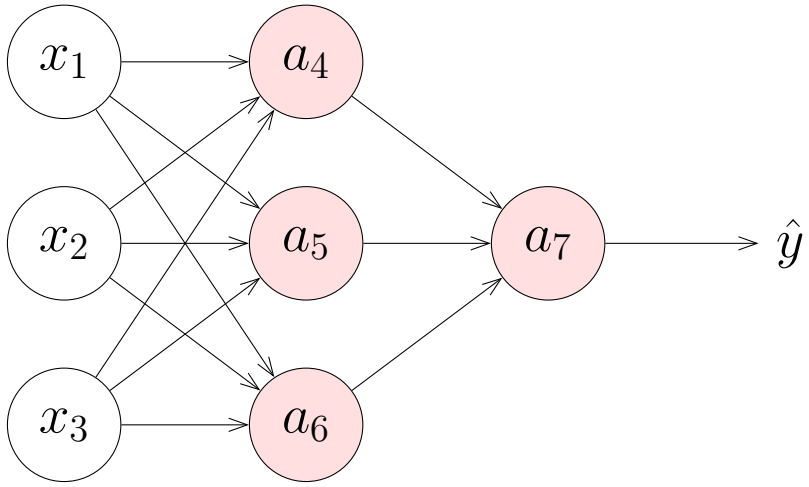
```
1  import tensorflow as tf
2
3  n = 4
4
5  # define training examples
6  X = tf.constant([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])
7  y = tf.constant([[0.0], [1.0], [1.0], [0.0]])
8
9  # define layer 1
10 W1 = tf.Variable(tf.random.normal((2, 2)))
11 b1 = tf.Variable(tf.random.normal((1, 2)))
12
13 # define layer 2
14 W2 = tf.Variable(tf.random.normal((2, 1)))
15 b2 = tf.Variable(tf.random.normal((1, 1)))
16
17 eta = 1
18
```

```
19 for r in range(500):
20     # define output and loss
21     with tf.GradientTape() as tape:
22         a1 = tf.math.sigmoid(tf.matmul(X, W1) + b1)
23         a2 = tf.math.sigmoid(tf.matmul(a1, W2) + b2)
24         L = tf.reduce_sum(-(y*tf.math.log(a2) +
25                             (1-y)*tf.math.log(1-a2)))/n
26         print("%5d %.6f" % (r, L))
27
28     # calculate gradients
29     [gw1, gb1, gw2, gb2] = tape.gradient(L, [W1, b1, W2, b2])
30
31     # update w and b
32     W2.assign_add(-eta*gw2)
33     b2.assign_add(-eta*gb2)
34     W1.assign_add(-eta*gw1)
35     b1.assign_add(-eta*gb1)
36
37
```

```
38 # calculate y_hat
39 a1 = tf.math.sigmoid(tf.matmul(X, W1) + b1)
40 a2 = tf.math.sigmoid(tf.matmul(a1, W2) + b2)
41 y_hat = tf.round(a2)
42 print("Predicted output =")
43 tf.print(y_hat)
```

```
    0 0.689692
    1 0.680214
    2 0.676115
    ...
  497 0.027812
  498 0.027736
  499 0.027660
Predicted output =
[[0]
 [1]
 [1]
 [0]]
```

Example 2.6. Given the following multilayer perceptron, write an expression for the output $\hat{y}$.



2.6 Keras API

Tensorflow also includes the Keras API (`https://keras.io/`) in order to make the implementations of machine learning applications become easier. Most of machine learning engineers use an API or a library like this. They usually do not implement machine learning algorithms unless they works a research scientist.

Using the Keras API, we only need to define a model composing of multiple layers, specify the number of units in each layer, and decide the training algorithm as well as its parameters.

```
1  import tensorflow as tf
2
3  n = 4
4
5  # define training examples
6  X = tf.constant([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])
7  y = tf.constant([[0.0], [1.0], [1.0], [0.0]])
8
9  # define the model
10 model = tf.keras.models.Sequential([
11     tf.keras.layers.Input(shape=(2,)),
12     tf.keras.layers.Dense(2, activation='sigmoid'),
13     tf.keras.layers.Dense(1, activation='sigmoid')
14 ])
15
16
17
18
```

```
19 # define the training algorithm and loss function
20 model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=1),
21                loss='binary_crossentropy',
22                metrics=['accuracy'])
23
24 # train the model
25 model.fit(X, y, epochs=500, batch_size=1)
26
27 # predict
28 a = model.predict(X)
29 y_hat = tf.round(a)
30 print(y_hat)
31
32 # evaluate the model
33 model.evaluate(X, y)
```

```
Epoch 1/500
4/4 [==========] - 0s 2ms/step - loss: 0.8344 - accuracy: 0.4667
Epoch 2/500
4/4 [==========] - 0s 3ms/step - loss: 0.8137 - accuracy: 0.6333
Epoch 3/500
4/4 [==========] - 0s 2ms/step - loss: 0.8782 - accuracy: 0.1667
...
Epoch 499/500
4/4 [==========] - 0s 2ms/step - loss: 0.0072 - accuracy: 1.0000
Epoch 500/500
4/4 [==========] - 0s 3ms/step - loss: 0.0080 - accuracy: 1.0000
tf.Tensor(
[[0.]
 [1.]
 [1.]
 [0.]], shape=(4, 1), dtype=float32)
1/1 [==========] - 0s 343ms/step - loss: 0.0078 - accuracy: 1.0000
[0.007761340588331225, 1.0]
```

```
import tensorflow as tf
import numpy as np

from google.colab import drive
drive.mount('/content/drive')

# load dataset
X = np.load('/content/drive/My Drive/css324s20/catdog_train.npy')
n = X.shape[0]  # the number of examples
m = X.shape[1]  # the number of features in each example
y = np.concatenate((np.zeros(n//2), np.ones(n//2)))

# define the model
model = tf.keras.models.Sequential([
    tf.keras.layers.Input(shape=(m,)),
    tf.keras.layers.Dense(m//4, activation='sigmoid'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```
19
20 # define the training algorithm and loss function
21 model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
22                loss='binary_crossentropy',
23                metrics=['accuracy'])
24
25 # train the model
26 model.fit(X, y, epochs=200, batch_size=64)
27
28 # evaluate the model
29 model.evaluate(X, y)
30
31 model.save('/content/drive/My Drive/css324s20/catdog_mlp')
```

```
Epoch 1/200
391/391 [==========] - 3s 7ms/step - loss: 0.9237 - accuracy: 0.5075
Epoch 2/200
391/391 [==========] - 3s 7ms/step - loss: 0.7166 - accuracy: 0.5442
Epoch 3/200
391/391 [==========] - 3s 7ms/step - loss: 0.6962 - accuracy: 0.5567
...
Epoch 199/200
391/391 [==========] - 3s 7ms/step - loss: 0.4841 - accuracy: 0.7641
Epoch 200/200
391/391 [==========] - 3s 7ms/step - loss: 0.4819 - accuracy: 0.7669
782/782 [==========] - 2s 3ms/step - loss: 0.4751 - accuracy: 0.7698
```

```
1  import tensorflow as tf
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  # load test set
6  X_test = np.load('../perceptron/catdog/catdog_test.npy')
7
8  # load model
9  model = tf.keras.models.load_model('./catdog_mlp')
10
11 n_rows, n_cols = 3, 3
12 fig, axs = plt.subplots(n_rows, n_cols)
13 fig.set_size_inches(10, 10)
14 axs = axs.ravel()
15 y_hat = tf.round(model.predict(X_test[:(n_rows*n_cols), :]))
16 for i in range(n_rows*n_cols):
17     # display an image
18     im = X_test[i].reshape((64, 64, 3))
```

```
19      axs[i].imshow(im)
20      if y_hat[i] == 0:
21          axs[i].set_title("predicted output: cat")
22      else:
23          axs[i].set_title("predicted output: dog")
24  plt.savefig('catdog_test_mlp.pdf', bbox_inches='tight')
```


predicted output: cat
predicted output: dog
predicted output: dog
predicted output: dog
predicted output: cat
predicted output: cat
predicted output: cat
predicted output: cat
predicted output: cat

Chapter 3

Deep Neural Networks

3.1 Shallow versus Deep Neural Networks

Given a multilayer perceptron, it is

- a *shallow neural network* when the number of hidden layers is 1 or 2, or
- a *deep neural network* when the number of hidden layers is more than 2.

Is a deep neural network necessary?

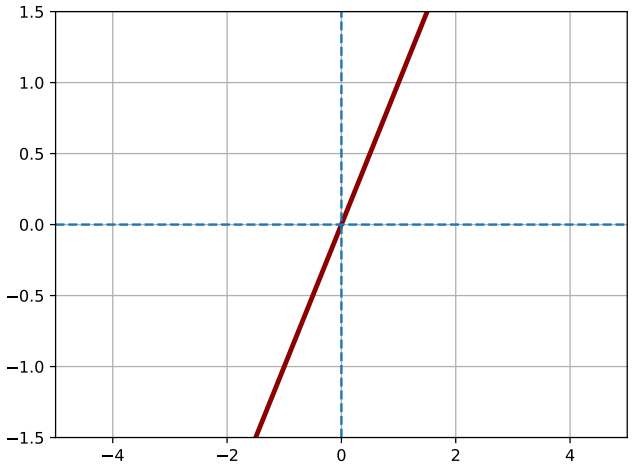
> a feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly. In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error.
>
> Goodfellow et al. (`https://www.deeplearningbook.org`)

3.2 Activation Functions

An activation function transforms the weighted sum of the input into an output.

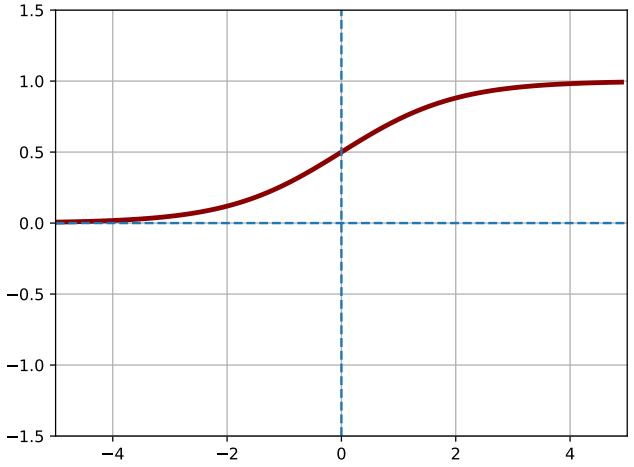
- The *identity* or *linear* function,

$$g_{linear}(z) = z$$
$$g'_{linear}(z) = 1$$



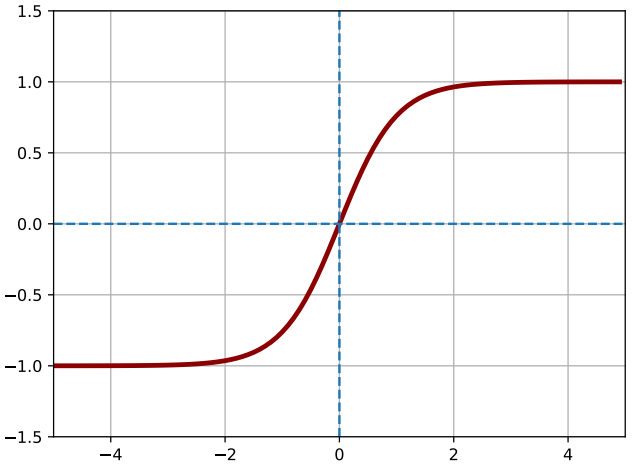
- The *sigmoid* or *logistic* function,

$$g_\sigma(z) = \frac{1}{1 + e^{-z}}$$
$$g'_\sigma(z) = g_\sigma(z)(1 - g_\sigma(z))$$



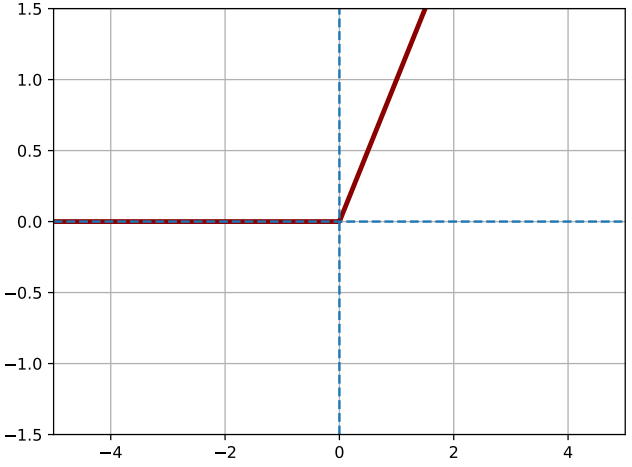
- The *hyperbolic tangent* function,

$$g_{tanh}(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$
$$= 2g_\sigma(2z) - 1$$
$$g'_{tanh}(z) = 1 - \tanh^2(z)$$



- The *ReLU (Rectified Linear Unit)* function,

$$g_{relu}(z) = \max(0, z)$$
$$g'_{relu}(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$



Remarks on activation functions

- A network composing of linear units can only represent a linear function. A sufficiently large network with nonlinear activation functions is capable to represent arbitrary functions.

- A deep neural network may suffer from *vanishing gradient* problem. This is because a gradient is calculated from a product of derivatives. When the gradient is very small, weights are not properly updated.

$$\frac{\partial \mathcal{L}}{\partial w_{3,5}} = \frac{\partial \mathcal{L}}{\partial a_5} \cdot \frac{\partial a_5}{\partial z_5} \cdot \frac{\partial z_5}{\partial w_{3,5}}$$

$$\frac{\partial \mathcal{L}}{\partial w_{1,3}} = \frac{\partial \mathcal{L}}{\partial a_5} \cdot \frac{\partial a_5}{\partial z_5} \cdot \frac{\partial z_5}{\partial a_3} \cdot \frac{\partial a_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_{1,3}}$$

  Compared to sigmoid and tanh, ReLU suffers less from the vanishing gradient problem. ReLUs are an default choice of hidden unit.

- A ReLU might *die* during training since the ReLU returns 0 when its input < 0.

$$a_k = g_{relu}(w_{1,k}x_1 + w_{2,k}x_2 + b_k)$$

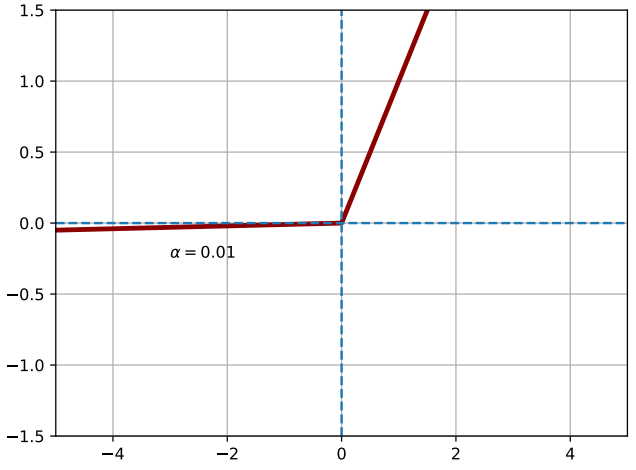
When $w_{1,k}$, $w_{2,k}$, and b_k are negative, it may cause the weighted sum be negative. Thus, the output of the ReLU function is 0. This unit may not get updated.

Dead ReLUs are not necessary bad if they don't happen all the time.
They sometimes help regularized the network.

Leaky ReLU is proposed to deal with the dying ReLU problem.

$$g_{leaky}(z) = \max(0, z) + \alpha \times \min(0, z)$$

$$g'_{leaky}(z) = \begin{cases} 1 & z > 0 \\ \alpha & z \leq 0 \end{cases}$$



3.3 Loss Functions

- Mean Squared Error (MSE) measures the squared difference between y and $\hat{y}$.
- Mean Absolute Error (MAE) measures the difference between y and $\hat{y}$.

$$\mathcal{L}(\mathbf{x}^{[i]}, y^{[i]}; \mathbf{w}, b) = |y^{[i]} - \hat{y}^{[i]}|$$

  Both MSE and MSE losses are used for regression when we believe that $y^{[i]}$ comes from a normal distribution.

- Binary Cross-Entropy Loss measures the negative log likelihood.

- Poisson Loss is for regression when we believe that $y^{[i]}$ comes from a Poisson distribution. For example, the number of COVID-19 cases, the number of emails, etc.

$$\mathcal{L}(\mathbf{x}^{[i]}, y^{[i]}; \mathbf{w}, b) = \hat{y}^{[i]} - y^{[i]} \log(\hat{y}^{[i]})$$

- Squared Hinge Loss is for constructing a binary classifier with *maximum margin*.

$$\mathcal{L}(\mathbf{x}^{[i]}, y^{[i]}; \mathbf{w}, b) = \max(0, 1 - y^{[i]} \cdot \hat{y}^{[i]})$$

where both $y^{[i]}$ and $\hat{y}^{[i]}$ are either -1 or 1.

3.4 Input and Output Encoding

- If an attribute is Boolean, usually $false = 0$ and $true = 1$.
- If an attribute is a real value, its value should be *normalized* or *standardized*:

Normalization

$$z = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Standardization

$$z = \frac{x - \mu}{\sigma}$$

If the values change enormously among examples, they should be mapped onto a *log scale*.

- If an attribute is a categorical value with more than two categories, it is common to use the *one-hot encoding*.

  An attribute with d possible values can be represented using d bits. For any given value, the corresponding bit is set to 1 and all the others are set to 0.

  For example, an attribute *weather* has four possible values: *sun*, *cloud*, *rain*, *snow*.

$$sun = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad cloud = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad rain = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad snow = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

- If an attribute is an ordinal value, it can be encoded using the *ordered encoding*.

  For example, an attribute *age* has 4 values. Each value represents an age group where $grp_0 < grp_1 < grp_2 < grp_3$.

$$grp_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad grp_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad grp_2 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad grp_3 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix},$$

Example 3.1. The objective of the *Auto MPG* dataset[1] is to predict the fuel efficiency of a car (in miles per gallon) from the car's attributes, e.g. horse power, cylinders, etc.

Attribute name	**Type**	**Description**
MPG	continuous	$[9.0, 46.6]$
cylinders	multi-valued discrete	$\{3, 4, 5, 6, 8\}$
displacements	continuous	$[68.0, 455.0]$
horsepower	continuous	$[46.0, 230.0]$
weight	continuous	$[1613.0, 5104.0]$
acceleration	continuous	$[8.0, 24.8]$
model year	multi-valued discrete	discrete values from 70 to 82
origin	multi-valued discrete	1=US, 2=Europe, 3=Japan

[1] `https://archive.ics.uci.edu/ml/datasets/auto+mpg`

Explain how to encode the following attributes:

1. *cylinders*

2. $$modelyear' = \begin{cases} 0 & \text{if } modelyear < 73 \\ 1 & \text{if } 73 \leq modelyear < 76 \\ 2 & \text{if } 76 \leq modelyear < 79 \\ 3 & \text{if } modelyear \geq 79 \end{cases}$$

3. *origin*

What should be the number of output units and the activation function of the output layer?

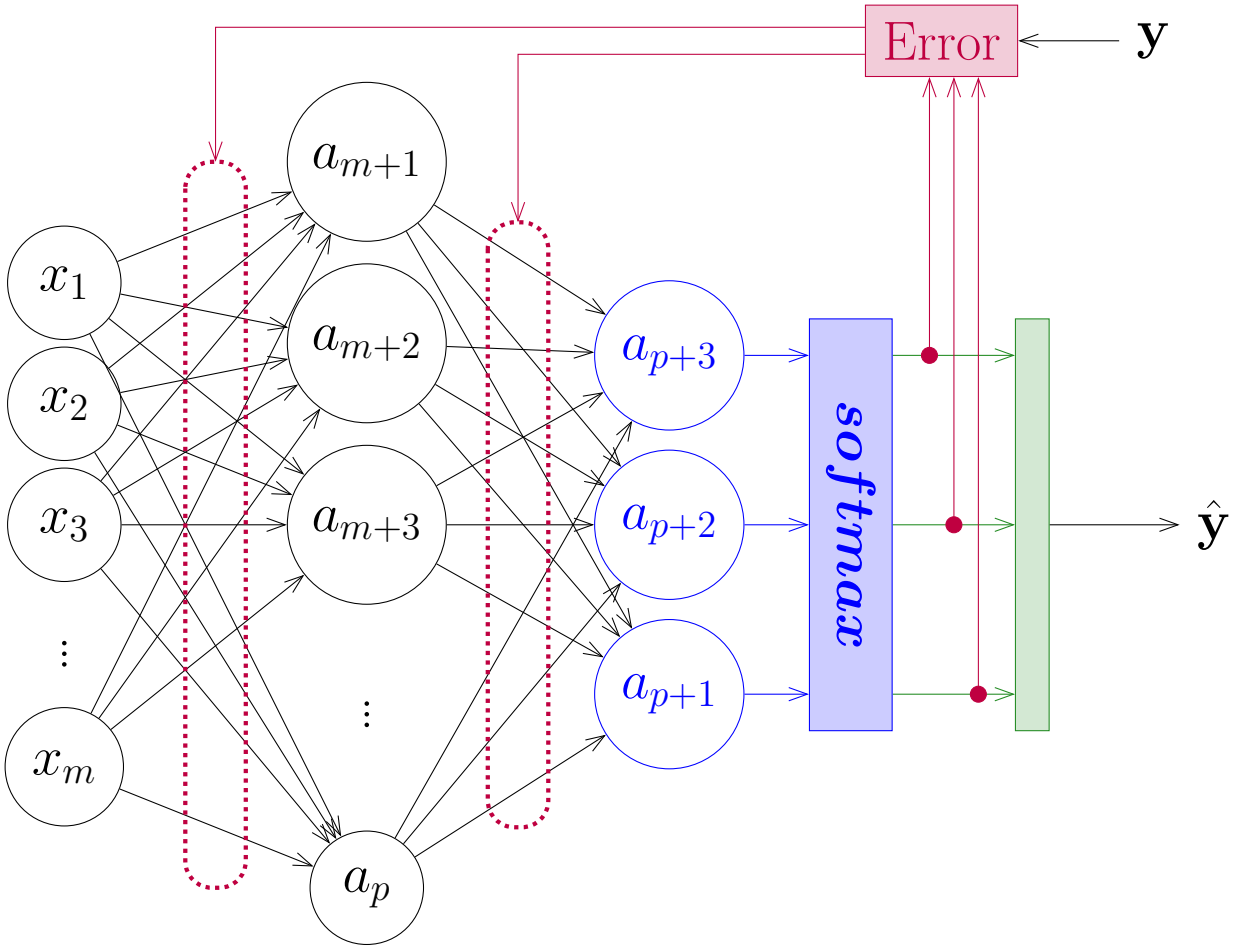
3.5 Multi-Class Classification

To handle a training set with more than 2 classes, i.e. $y^{[i]} \in \{0, 1, 2, \ldots, C-1\}$ and $C > 2$, we use the one-hot encoding to represent the output (y). Therefore, we construct an MLP with C units in the output layer. Since each unit predicts a probability value, each output unit is designated to each class.

To train this type of networks, the target output needs to be encoded in the *one-hot* encoding. Each target output is transformed into a vector with C members. When $y^{[i]} = k$, the one-hot vector has only the kth element set to 1, while the other elements are set to 0.

$$y^{[i]} = k$$
$$\downarrow$$
$$\mathbf{y}^{[i]} = [\underbrace{0, \ldots, 0, 1, 0, \ldots, 0}_{k\text{th entry is 1}}]^\top$$

Example 3.2. When we have a training set with 3 classes, i.e. 0, 1, and 2, we design an MLP to have 3 output units.



3.5.1 Softmax Activation Function

The softmax function is a vector function accepting a vector of weighted sums and returns a vector of probabilities. It is a generalization of the sigmoid function when there are more than two classes.

$$\boldsymbol{softmax}(\mathbf{z}) = [\sigma_0, \sigma_1, \ldots, \sigma_{C-1}]^\top \tag{3.1}$$

where each element of the output vector, $\sigma_i = \dfrac{e^{z_i}}{\sum_{j=0}^{C-1} e^{z_j}}$ for $i \in \{0, 1, \ldots, C-1\}$.

3.5.2 Categorical Cross-Entropy Loss

From the cross-entropy loss function,

$$\mathcal{L}_{log}(\mathbf{x}, y; \mathbf{w}, b) = -\big[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\big]$$

We can extend it to multi-class classification:

$$\begin{aligned}\mathcal{L}_{catlog}(\mathbf{x}, \mathbf{y}; \mathbf{w}, b) &= -\sum_{c=0}^{C-1} y_c \log(\hat{y}_c) \\ &= -\big[y_0 \log(\hat{y}_0) + y_1 \log(\hat{y}_1) + \cdots + y_{C-1} \log(\hat{y}_{C-1})\big]\end{aligned}$$

Example 3.3. Calculate $\mathcal{L}_{catlog}$ when $\mathbf{z} = [2, 3, 1]^\top$. Note that $e^1 = 2.718$, $e^2 = 7.389$, and $e^3 = 20.086$.

3.5.3 Argmax Function

To convert a predicted output from the one-hot encoding into a nominal label, we select the class with the highest probability.

The *argmax* function accepts a vector and returns the index of the vector element with the maximum value. For example,

$$\arg\max\left(\begin{bmatrix}0.15\\0.80\\0.05\end{bmatrix}\right) = 1$$

This is because the element at index 1 has the maximum value. Here, we assume that the index starts at 0.

Example 3.4. Calculate $\arg\max(\boldsymbol{softmax}(\mathbf{z}))$ when $\mathbf{z} = [2, 3, 1]^\top$.

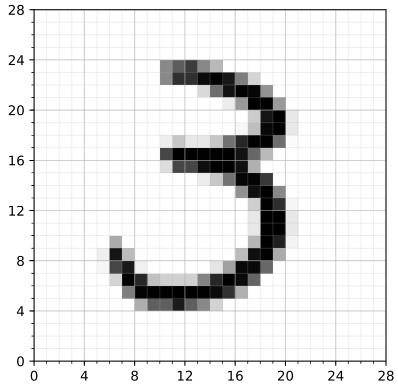
Note that $e^1 = 2.718$, $e^2 = 7.389$, and $e^3 = 20.086$.

Example 3.5. Recognize handwritten digits.[2]

This MNIST database contains 70,000 handwritten images. Each image is labeled with one of the digits, i.e. $\mathcal{T} = \{0, 1, 2, \ldots, 9\}$.

[2]The MNIST Database of Handwritten Digits (`http://yann.lecun.com/exdb/mnist/`); Y. LeCun, L. Bottou, Y. Bengio and P. Haffner: Gradient-Based Learning Applied to Document Recognition, Proceedings of the IEEE, 86(11):2278-2324, November 1998

An image is preprocessed and transformed into a 28×28 greyscale image. Therefore, each input vector $\mathbf{x}^{(i)}$ is composed of 784 features.



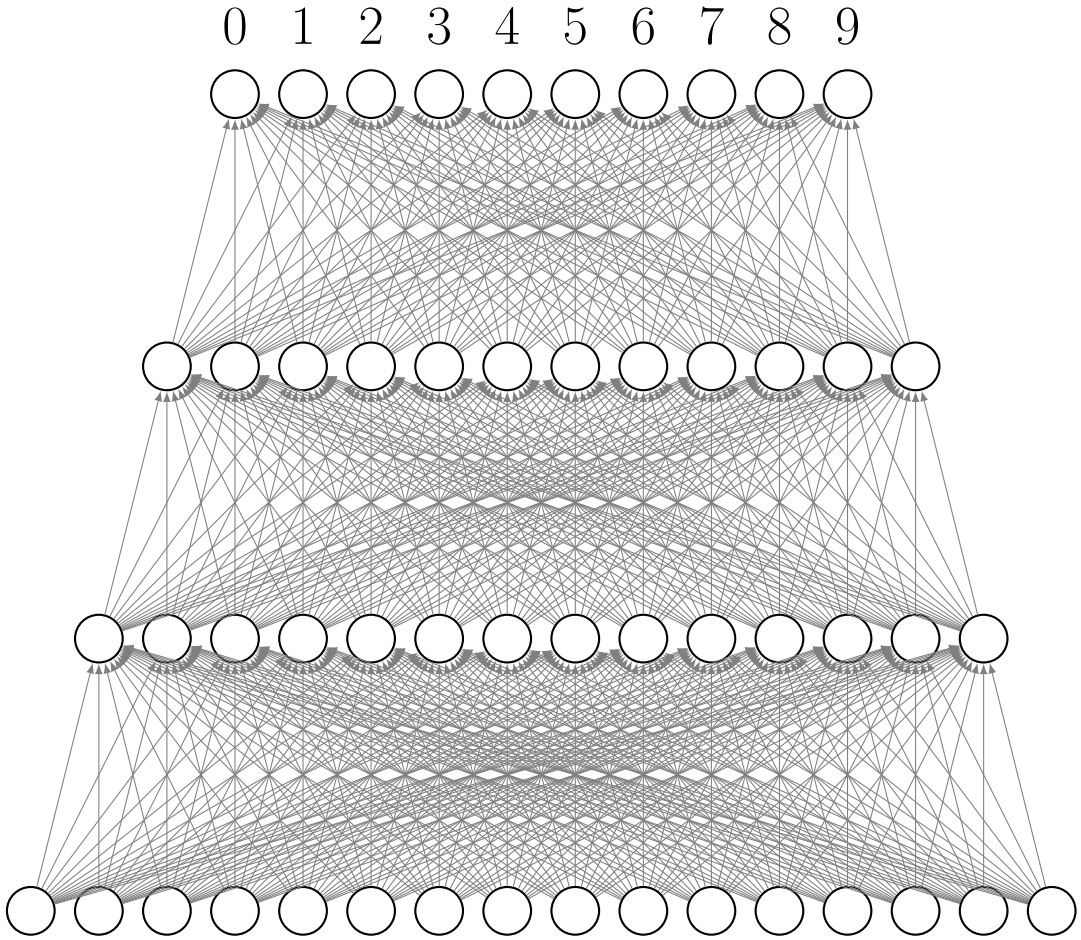
$=$

0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.46	0.63	0.76	0.47	0.27	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.46	0.81	0.8	0.95	0.99	0.9	0.5	0.16	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.15	0.57	0.93	0.99	0.97	0.42	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.07	0.4	0.99	0.99	0.4	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.2	0.92	1.	0.09	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.04	0.25	0.96	0.99	0.09	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.07	0.22	0.09	0.09	0.22	0.55	0.84	0.99	0.99	0.57	0.02	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.72	0.99	0.99	0.99	0.99	0.99	0.92	0.59	0.27	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.13	0.72	0.72	0.94	0.99	0.99	0.93	0.31	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.07	0.22	0.55	0.97	0.98	0.77	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.43	0.94	0.99	0.46	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.19	0.99	0.82	0.04	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.09	0.99	0.99	0.09	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.09	0.99	0.99	0.09	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.33	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.29	0.99	0.87	0.05	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.07	0.91	0.27	0.	0.	0.	0.	0.	0.	0.	0.	0.27	0.94	0.99	0.33	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.04	0.72	0.89	0.06	0.	0.	0.	0.	0.	0.1	0.37	0.96	0.96	0.59	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.16	0.99	0.84	0.25	0.18	0.18	0.18	0.48	0.84	0.99	0.95	0.6	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.5	0.98	0.99	0.99	0.99	0.99	0.99	0.95	0.8	0.31	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.31	0.62	0.62	0.89	0.62	0.46	0.17	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.

$$= \left[0.0, 0.0, \ldots, 0.46, \ldots, 0.0\right]_{1 \times 784}$$

$$= \mathbf{x}^{(19000)}$$

Construct a neural network to cope with the MNIST dataset.

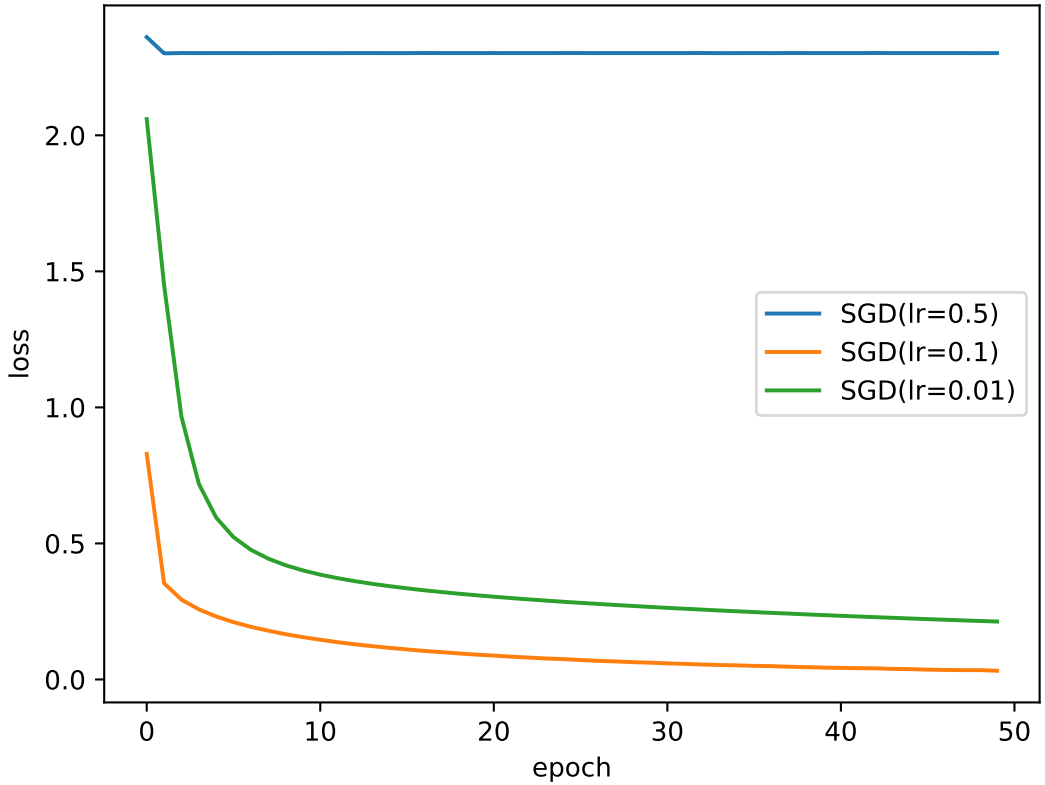


See `css324_deep_mnist.ipynb` for source code.

3.6 Learning Rate and Optimizers

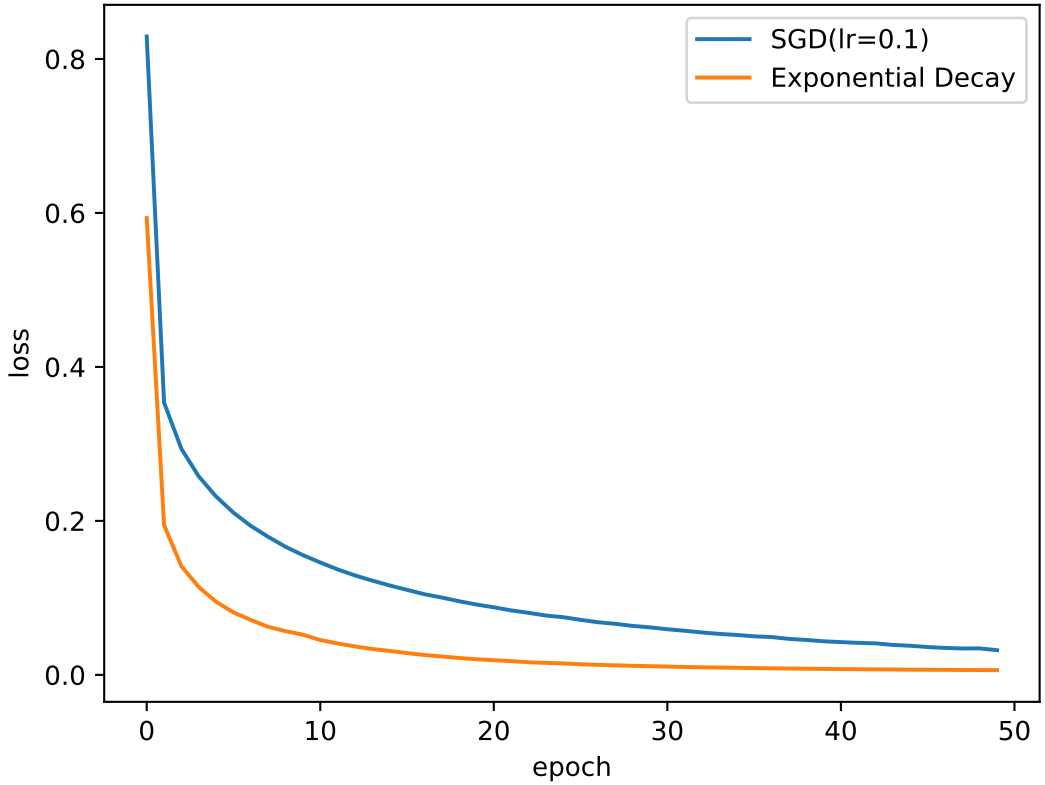
The learning rate or step size should be appropriately set.

- If the learning rate is too small, it causes slow convergence.
- If the learning rate is too high, it causes plateau or even divergence.



Generally, decreasing the learning rate over time improves convergence. For example, we can schedule the learning rate to exponentially decayed.

$$\eta_t = 0.5 \times 0.97^{\frac{t}{100}}$$



3.6.1 Momentum

When the weights and bias are updated to be near a local minimum, the gradient calculated from a small minibatch may have *high variance*. It may lead to a wrong direction.
Possible solutions for the problem are:

- Increase batch size as the training goes on,
- Keep the running average of the gradients of past minibatches to compensate for small batch sizes.

$$\mathbf{m}_t = \gamma \mathbf{m}_{t-1} + \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}_{t-1})$$
$$\mathbf{w}_t = \mathbf{w}_{t-1} - \mathbf{m}_t$$

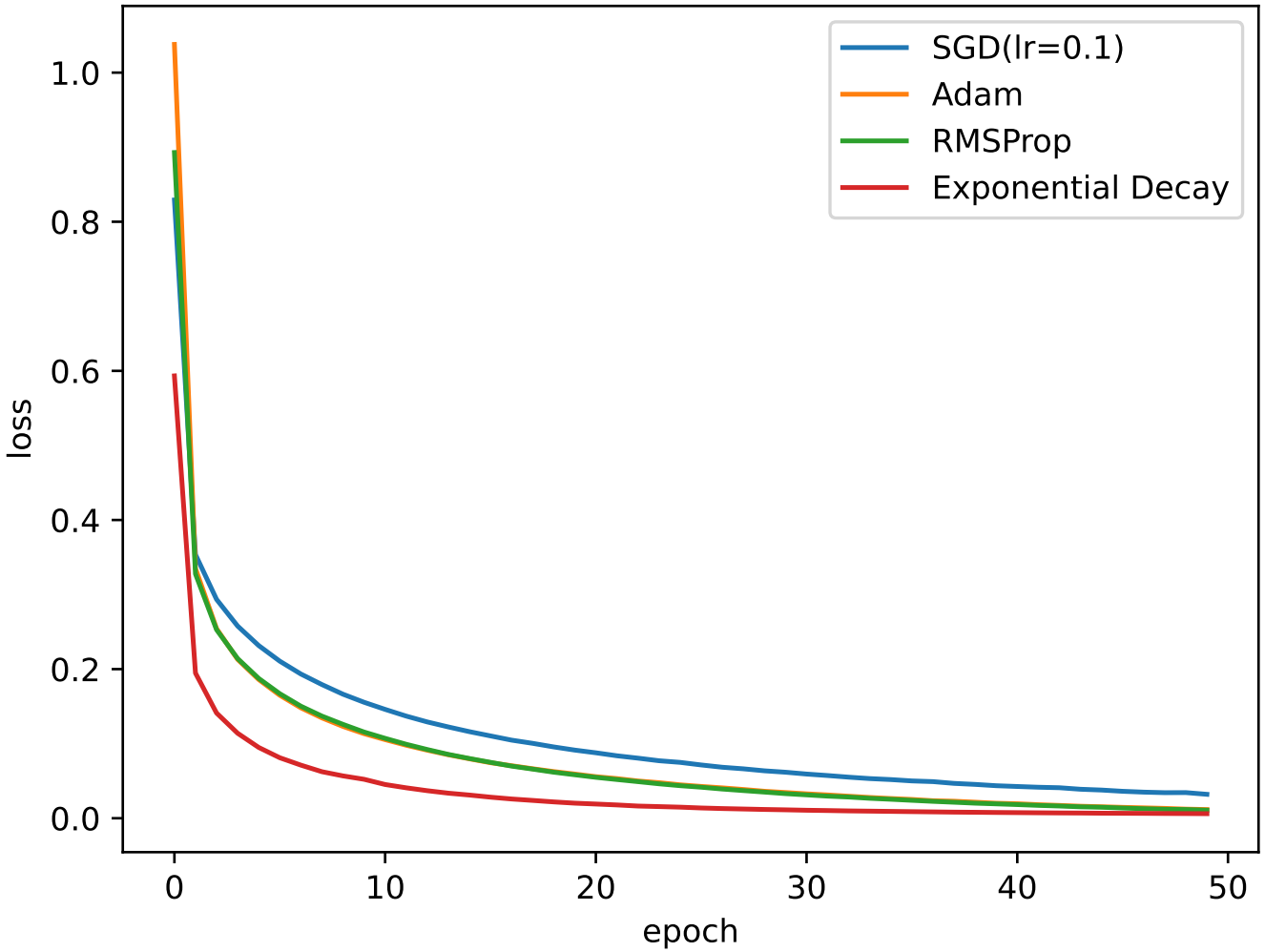
where η is the learning rate, and γ is typically set to 0.90.

3.6.2 Optimizers

Various optimization techniques have been proposed to automatically adapt the learning rate.

- RMSProp (Root Mean Square Propagation)
  - maintains a moving average of the squared gradients, and
  - divide the gradient by the square root of the moving average.
- Adam (Adaptive Moment Estimation) improves RMSProp by using the running averages of both the first-order moments and the second-order moments of the gradients.

The Keras API also implements these two methods. We can choose them by specifying as a value for the `optimizer` parameter of `model.compile`.


SGD(lr=0.1)
Adam
RMSProp
Exponential Decay
loss
epoch
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50

3.7 Batch Normalization

Batch normalization is a technique to improve the convergence rate by normalized the outputs of the hidden layers from the examples within each batch.

$$\hat{a}_i = \gamma \frac{a_i - \mu}{\sqrt{\epsilon + \sigma^2}} + \beta$$

where a_i $(i = 1, \ldots, b)$ is the output of a hidden unit, i is the index within a batch, and b is the batch size. The parameters γ and β are trainable parameters.

```
# Define a model
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.BatchNormalization(),

    tf.keras.layers.Dense(128, activation='relu',
        kernel_regularizer=tf.keras.regularizers.L2(1e-4)),
    tf.keras.layers.BatchNormalization(),

    tf.keras.layers.Dense(64, activation='relu',
        kernel_regularizer=tf.keras.regularizers.L2(1e-4)),
    tf.keras.layers.BatchNormalization(),

    tf.keras.layers.Dense(10, activation='softmax')
])
```

3.8 Underfitting and Overfitting

Underfitting is a situation when a model is not trained enough to capture the characteritisc of a dataset.

Overfitting is a situation when a model is trained to fit too much with a training set. Therefore, it is not good enough to work with unknown examples. Overfitting occurs easily in a neural network with many hidden layers and parameters. Overfitting may cause from

- the model is too complex for the examples,
- too many features with a small number of examples.

3.8.1 Three Sets

Given a set of examples, the entire set should not be used to train and evaluate a model. This is because evaluating a model by the same set of examples used for training is unreliable.
In practice, a set of examples (or, dataset) should be randomly separated into three sets for different purposes, i.e.

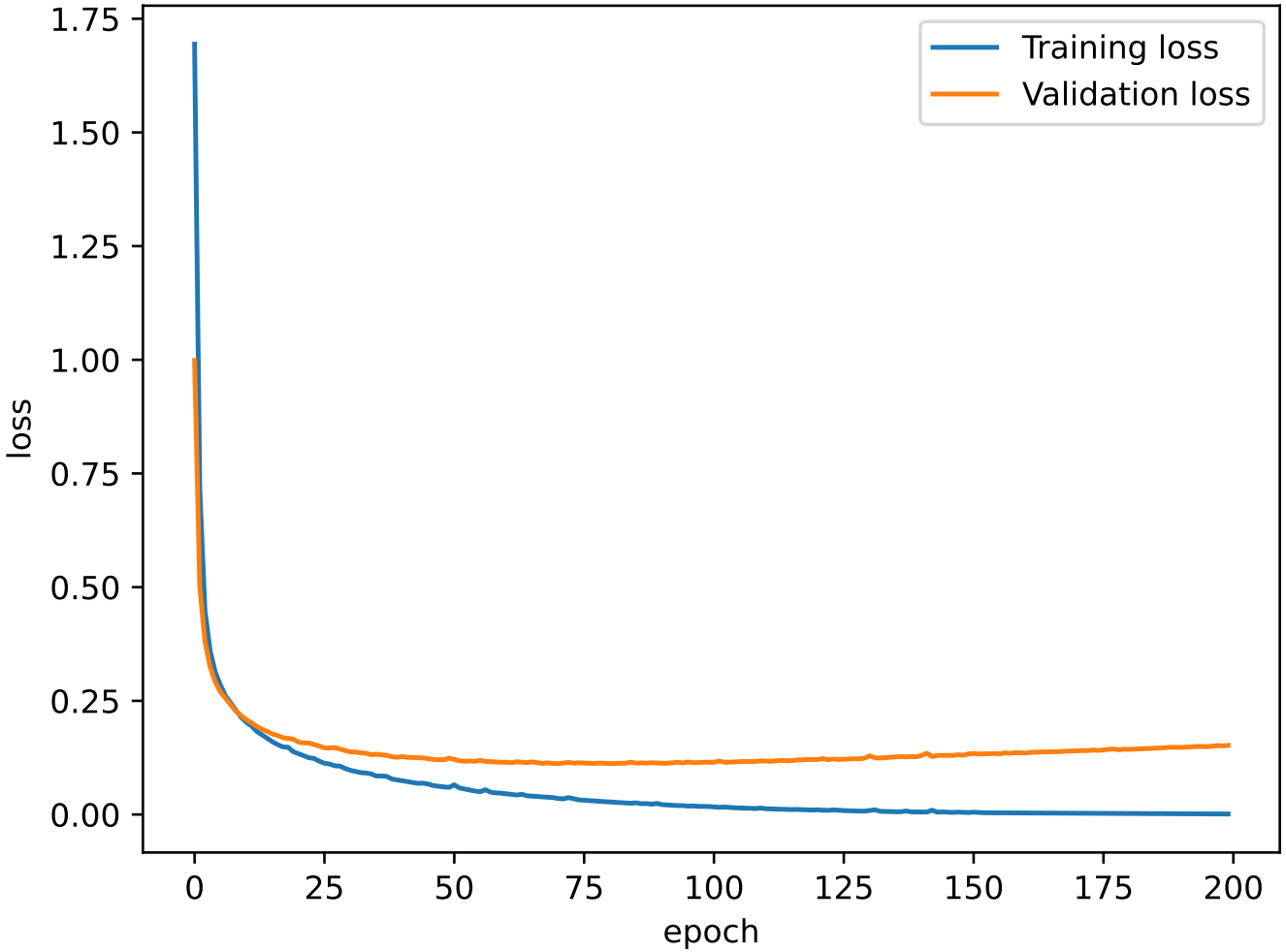
- *training set* for building the model
- *validation set* for finding the best algorithm and the best values of hyperparameters.
- *test set* for evaluating the final model

Typically, the training set should be the biggest set of examples since it is used to build the model. The validation and test sets are much smaller than the training set and roughly the same size.

- When the number of the examples is not huge, the rule of thumb is 70% for the training set, 15% for the validation set, and 15% for the test set.
- If we have millions of examples, the set of examples can be split into 95% for training, 2.5% for validation, and 2.5% for testing.

In the `model.fit` function, we can use a named parameter, `validation_split`, to construct a validation set from a set of examples. This validation set is used to evaluate the model during the training process.

```
# Define a model
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=3e-4),
              loss=tf.keras.losses.CategoricalCrossentropy(),
              metrics=['accuracy'])

# Train the model
batch_size = 1024
history = model.fit(X_train, Y_train, epochs=200,
                    batch_size=batch_size,
                    shuffle=True, validation_split=0.3)
```




```
Training accuracy = 0.9912
Test accuracy     = 0.9737
```

3.8.2 Regularization

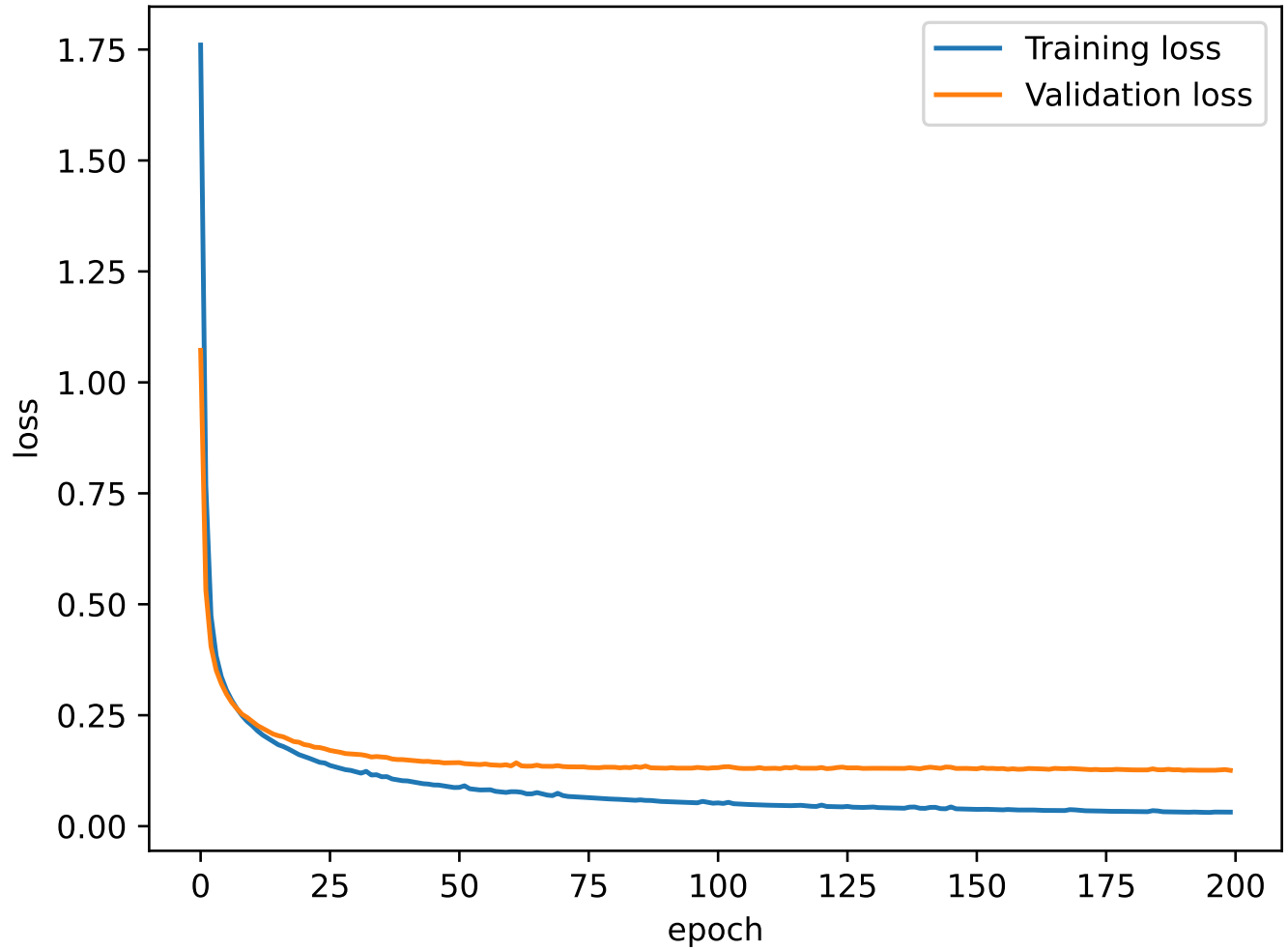
Regularization is a technique to modify the objective function in order to penalize a too complex model. It therefore forces the algorithm to build a less complex model and avoid overfitting.

A neural network is less complex when many weights are zeros. The *L2 regularization* tries to minimize $\|\mathbf{w}\|^2$ as well as the errors:

$$\mathcal{J}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_{\mathbf{w},b}(\mathbf{x}^{[i]}, y^{[i]}) + C\|\mathbf{w}\|^2 \tag{3.2}$$

where C is a hyperparameter controlling the importance of regularization.

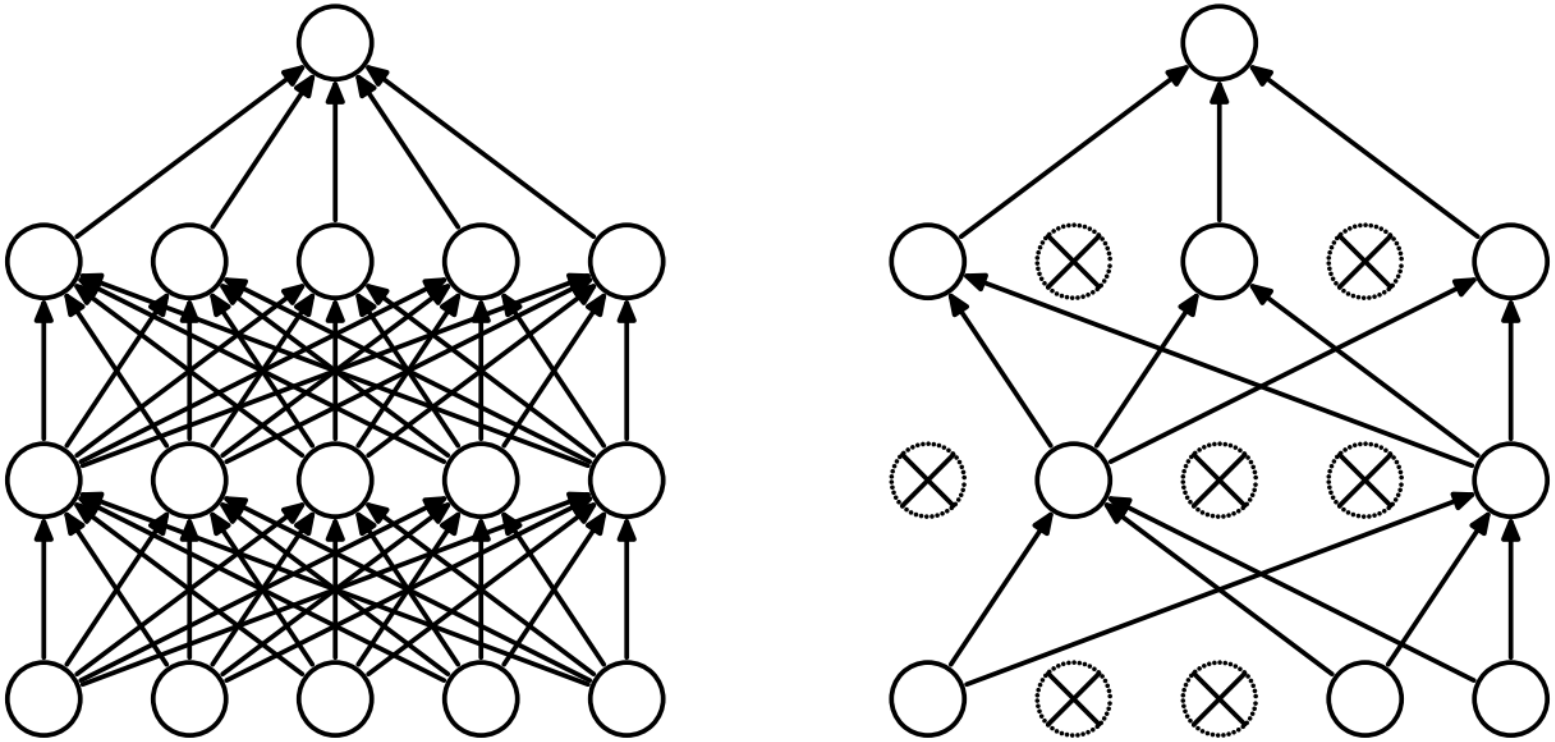
```
# Define a model
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu',
        kernel_regularizer=tf.keras.regularizers.L2(1e-4)),
    tf.keras.layers.Dense(64, activation='relu',
        kernel_regularizer=tf.keras.regularizers.L2(1e-4)),
    tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=3e-4),
              loss=tf.keras.losses.CategoricalCrossentropy(),
              metrics=['accuracy'])
```




```
Training accuracy = 0.9919
Test accuracy     = 0.9767
```

3.8.3 Dropout

Dropout is a technique that temporarily excludes a number of random units during the training process.

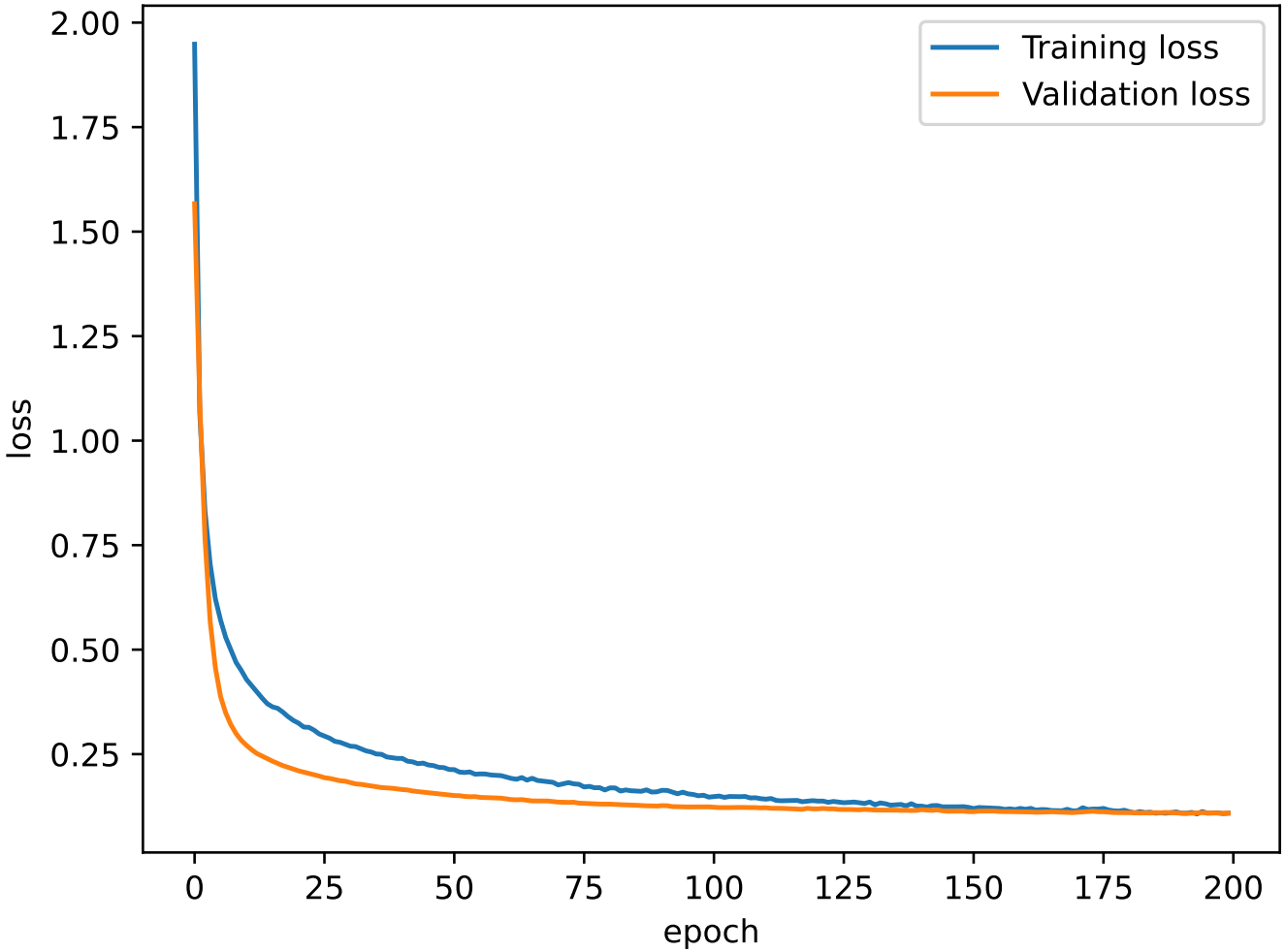


(a) Standard Neural Net

(b) After applying dropout.

Source: `https://wiki.tum.de/display/lfdv/Dropout`

```
# Define a model
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dropout(0.3),

    tf.keras.layers.Dense(128, activation='relu',
        kernel_regularizer=tf.keras.regularizers.L2(1e-4)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dropout(0.3),

    tf.keras.layers.Dense(64, activation='relu',
        kernel_regularizer=tf.keras.regularizers.L2(1e-4)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dropout(0.3),

    tf.keras.layers.Dense(10, activation='softmax')
])
```




```
Training accuracy = 0.9884
Test accuracy     = 0.9778
```

Performance Comparison

Method	Training Acc.	Test Acc.
Without regularization	99.12%	97.37%
L2 regularization	99.19%	97.67%
L2 regularization + Dropout + Batch normalization	98.84%	97.78%

It is therefore recommended that regularizers, dropout and batch normalization are applied in a neural network. The plot of training and validation losses can be used to select the most appropriate values of hyperparameters.

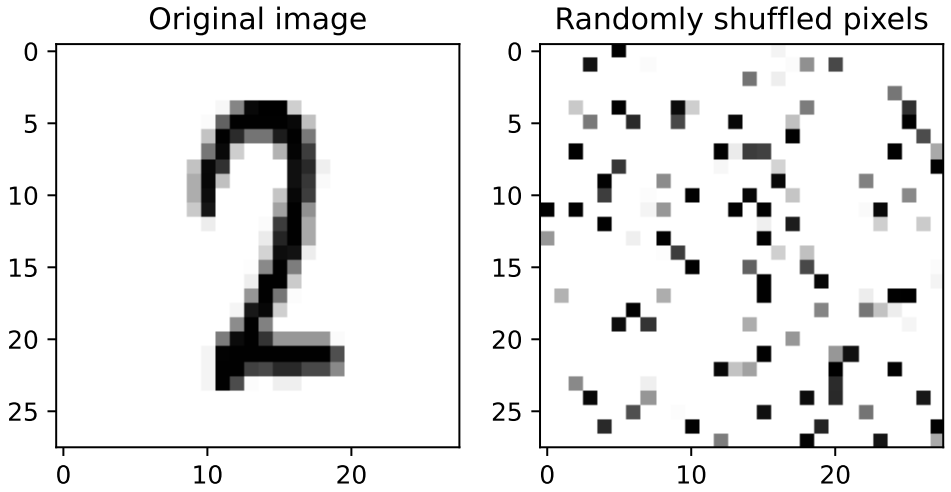
Chapter 4

Convolutional Networks

4.1 Neural Networks and Images

When *an image is flattened into a vector* before feeding it into a neural network with fully connected layers,

(1) adjacency of pixels is not taken into account,



Training the network with either the original image or the image with randomly shuffled pixels would yield the same result.

(2) the same pattern occuring in different locations may not be recognized because weights are not shared across locations,

0	1	0	0
1	1	1	0
0	1	0	0
0	0	0	0

0	0	0	0
0	0	1	0
0	1	1	1
0	0	1	0

(3) high-resolution images require very large numbers of weights.

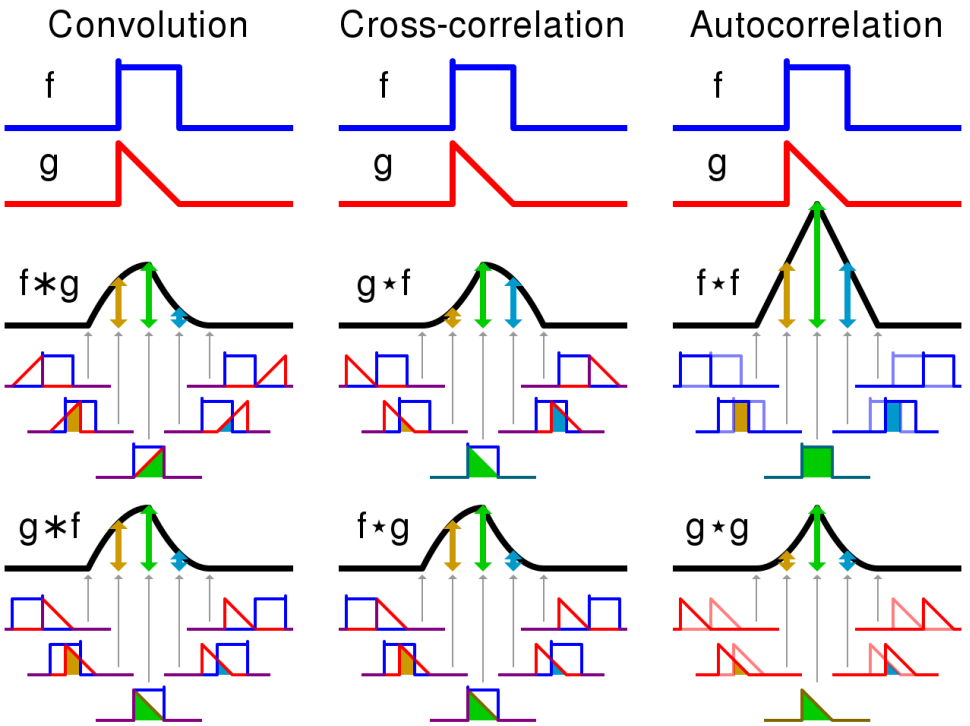
$$\textit{an image with } n \textit{ pixels} \rightarrow \textit{a layer with } \frac{n}{2} \textit{ units} = \frac{(n)(n)}{2} + \frac{n}{2} \textit{ parameters}$$

4.2 Convolution Operations

Convolution is defined as a operation between two functions:

$$[f * g](t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau \tag{4.1}$$

i.e. the integral of the product of f and g after g is flipped and shifted.



https://commons.wikimedia.org/wiki/File:Comparison_convolution_correlation.svg

The convolution operation is similar to *cross-correlation* when either f or g is flipped.

4.2.1 Cross-Correlation Operations

Given an input vector $\mathbf{x}$ and a vector kernel $\mathbf{k}$ of size l, the cross-correlation operation, $\mathbf{x} \star \mathbf{k}$, is defined as

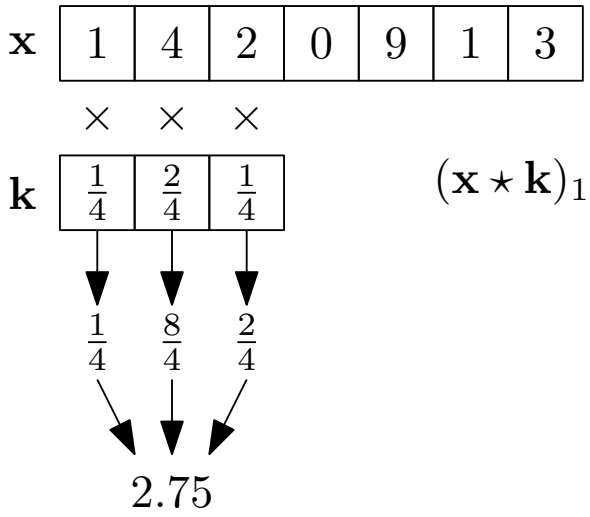
$$(\mathbf{x} \star \mathbf{k})_i = \sum_{u=1}^{l} k_u x_{i+u-1} \tag{4.2}$$

For example, when $l = 3$,

$$(\mathbf{x} \star \mathbf{k})_i = \sum_{u=1}^{3} k_u x_{i+u-1} = k_1 x_i + k_2 x_{i+1} + k_3 x_{i+2}$$

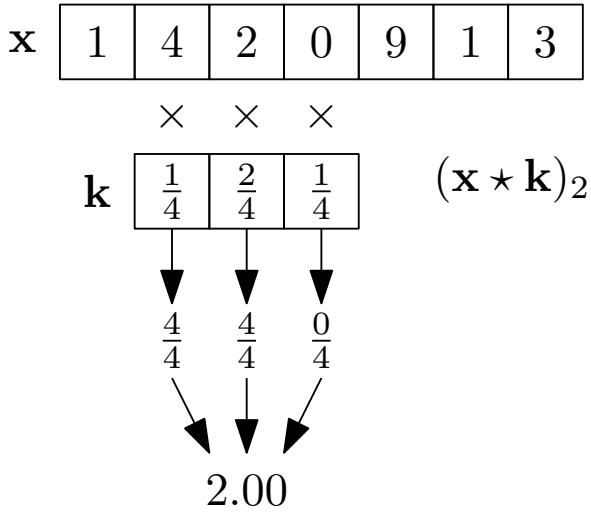
Therefore, the output at position i is the dot product between the kernel $\mathbf{k}$ and a portion of $\mathbf{x}$ of size l starting from x_i.

Example 4.1. Let $\mathbf{x} = \begin{bmatrix} 1 & 4 & 2 & 0 & 9 & 1 & 3 \end{bmatrix}$ and $\mathbf{k} = \begin{bmatrix} \frac{1}{4} & \frac{2}{4} & \frac{1}{4} \end{bmatrix}$, calculate $\mathbf{z} = \mathbf{x} \star \mathbf{k}$.



$$(\mathbf{x} \star \mathbf{k})_1 = \sum_{u=1}^{3} k_j x_{1+u-1}$$

$$= k_1 x_1 + k_2 x_2 + k_3 x_3$$



$$(\mathbf{x} \star \mathbf{k})_2 = \sum_{u=1}^{2} k_j x_{2+u-1}$$

$$= k_1 x_2 + k_2 x_3 + k_3 x_4$$

The cross-correlation operation can be applied to 2D arrays:

$$(\mathbf{x} \star \mathbf{k})_{i,j} = \sum_{u=1}^{k_h} \sum_{v=1}^{k_w} k_{u,v} x_{i+u-1,j+v-1}$$

where an array kernel $\mathbf{k}$ is a 2D array with size $k_h \times k_w$.

Example 4.2. Write an expression for $(\mathbf{x} \star \mathbf{k})_{i,j}$ when $\mathbf{k} = \begin{bmatrix} k_{1,1} & k_{1,2} \\ k_{2,1} & k_{2,2} \end{bmatrix}$

Example 4.3. Conduct a cross-correlation operation on the following $\mathbf{x}$ and $\mathbf{k}$.

1	4	1	1	2
4	3	1	2	1
5	2	2	2	5
1	3	2	4	3

$\star$

−1	−1	−1
−1	8	−1
−1	−1	−1

$=$

4	−9	1
−5	−3	−4

$$
\begin{aligned}
(\mathbf{x} * \mathbf{k})_{1,1} &= (-1)(1) + (-1)(4) + (-1)(1) + (-1)(4) + (8)(3) + (-1)(1) \\
&\quad + (-1)(5) + (-1)(2) + (-1)(2) \\
&= 4
\end{aligned}
$$

4.2.2 Boundary conditions

Valid convolution/cross-correlation

Given a 2D array $\mathbf{x}$ with n_h rows and n_w columns, and a kernel $\mathbf{k}$ with k_h rows and k_w columns, the convolution operation $\mathbf{x} \star \mathbf{k}$ returns a 2D array with the shape of

$$(n_h - k_h + 1) \times (n_w - k_w + 1).$$

Same convolution/cross-correlation

To output an array of the *same size* as the input, zeros are padded around the border of the input array. The number of padded zeros p can be calculated from the size of kernel l,

$$p = l - 1$$

Note that $\star_v$ and $\star_s$ are used to denote valid and same convolution/cross correlation operations, respectively.

Example 4.4. Conduct the same cross-correlation using $\mathbf{x}$ and $\mathbf{k}$ from the previous example.

$$p_h = 3 - 1 = 2$$

$$p_w = 3 - 1 = 2$$

$$\begin{array}{|ccccccc|} \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 4 & 1 & 1 & 2 & 0 \\ 0 & 4 & 3 & 1 & 2 & 1 & 0 \\ 0 & 5 & 2 & 2 & 2 & 5 & 0 \\ 0 & 1 & 3 & 2 & 4 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array} \star_S \begin{array}{|ccc|} \hline -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \\ \hline \end{array} = \begin{array}{|ccccc|} \hline -3 & 22 & -3 & 1 & 12 \\ 17 & 4 & -9 & 1 & -4 \\ 27 & -5 & -3 & -4 & 28 \\ -2 & 12 & 3 & 18 & 13 \\ \hline \end{array}$$

4.2.3 Strides

By default, the kernel is slided to the right and down one element at a time. *Stride* specifies the number of rows and columns moved per slide.

When the stride is set to $s_h \times s_w$, the shape of the output array is

$$\left\lfloor \frac{n_h - k_h + p_h + s_h}{s_h} \right\rfloor \times \left\lfloor \frac{n_w - k_w + p_w + s_w}{s_w} \right\rfloor$$

Example 4.5. Conduct the cross-correlation using $\mathbf{x}$ and $\mathbf{k}$ from the previous example using the stride of (2×2).

The shape of the output is

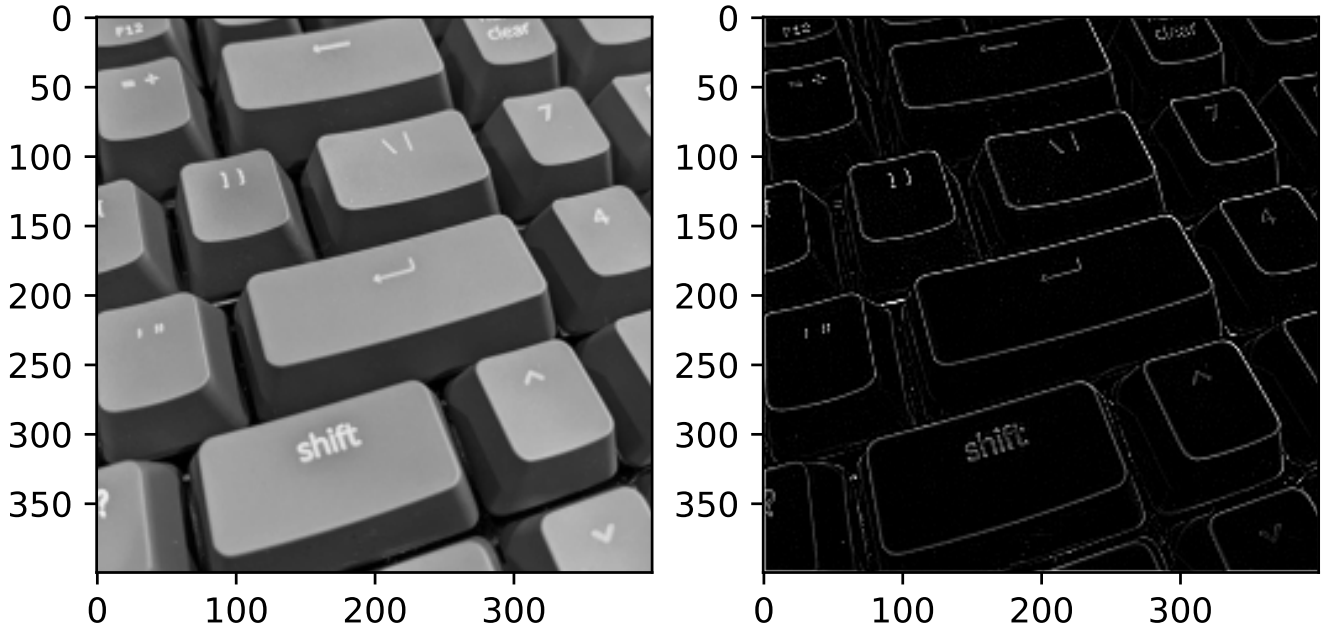
$$\left\lfloor \frac{4-3+2+2}{2} \right\rfloor \times \left\lfloor \frac{5-3+2+2}{2} \right\rfloor = 2 \times 3$$

$$\begin{array}{|ccccccc|} \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 4 & 1 & 1 & 2 & 0 \\ 0 & 4 & 3 & 1 & 2 & 1 & 0 \\ 0 & 5 & 2 & 2 & 2 & 5 & 0 \\ 0 & 1 & 3 & 2 & 4 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array} \star_s \begin{array}{|ccc|} \hline -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \\ \hline \end{array} = \begin{array}{|ccc|} \hline -3 & -3 & 12 \\ 27 & -3 & 28 \\ \hline \end{array}$$

4.2.4 Applications of Convolution/Cross-correlation

In the field of image processing, convolution operations are applied to process images. Various type of kernels have been designed to blur, sharpen, or detect edges of images.

Example 4.6. When $\mathbf{x}$ is an image and $\mathbf{k} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$, the convolution of $\mathbf{x}$ and $\mathbf{k}$ returns



```
1  import numpy as np
2  from scipy.signal import convolve2d
3  import matplotlib.pyplot as plt
4  import matplotlib.image as img
5
6  x = img.imread("kb.jpg")/255
7
8  k = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
9  z = convolve2d(x, k, "same")
10
11 fig, axs = plt.subplots(1, 2)
12 axs[0].imshow(x, vmin=0, vmax=1, cmap="gray")
13 axs[1].imshow(z, vmin=0, vmax=1, cmap="gray")
14 plt.show()
```

With a proper kernel, the convolution/cross-correlation operations can be used to extract *features* from local areas of input arrays.

4.3 Convolutional Neural Networks

4.3.1 Convolutional Layer

A Convolutional Neural Network, proposed by LeCun et al. in 1998, consists of *convolutional layers* using the gradient descent algorithm to find appropriate kernel values.
The learned kernels can be used to extract meaningful features from local areas of an input array.

$$\mathbf{z} = g\left((\mathbf{x} \star \mathbf{w}) + bJ_{h,w}\right) \tag{4.3}$$

where the shape of the output of the cross-correlation is $h \times w$, and $J_{h,w}$ is an all-ones matrix with h rows and w columns.

Example 4.7. A convolutional layer conducting a valid convolution operation accepts an input array of shape 3×3. It uses 1 weighting kernel of shape 2×2. What is the output of this layer?

$$\mathbf{z} = g\left(\left(\begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{2,3} \end{bmatrix} \star_v \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix}\right) + \begin{bmatrix} b & b \\ b & b \end{bmatrix}\right) = \begin{bmatrix} z_{1,1} & z_{1,2} \\ z_{2,1} & z_{2,2} \end{bmatrix}$$

$z_{1,1} = g(w_{1,1}x_{1,1} + w_{1,2}x_{1,2} + w_{2,1}x_{2,1} + w_{2,2}x_{2,2} + b)$

$z_{1,2} =$

$z_{2,1} =$

$z_{2,2} =$

4.3.2 Receptive Field

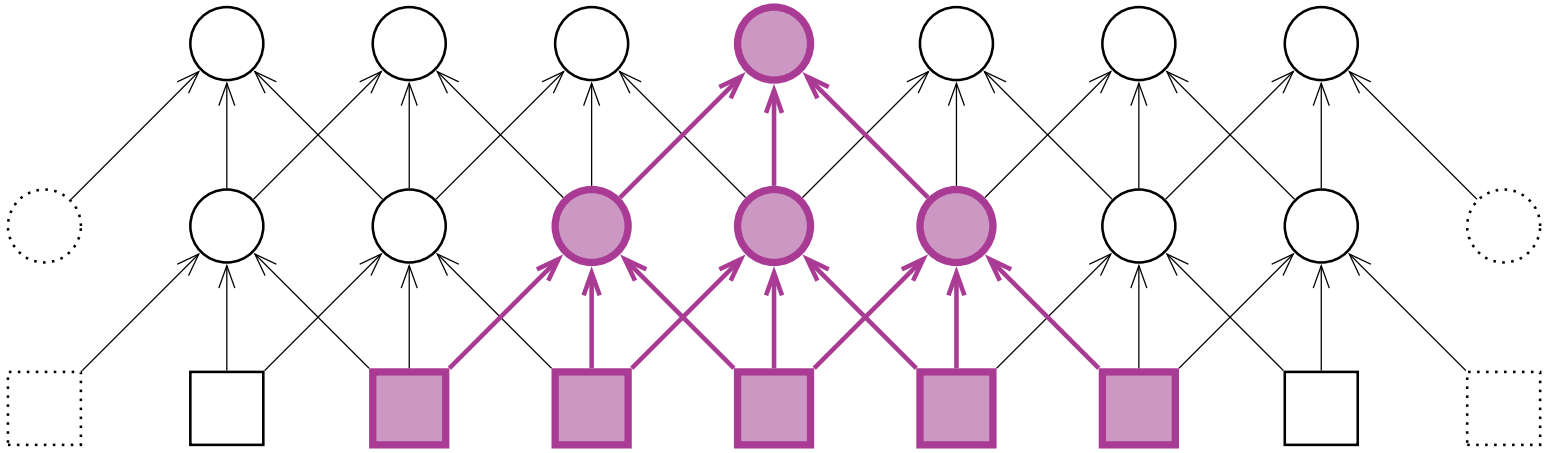
Receptive field of a neuron is the part of the input that affects the output of that neuron.

The receptive field of a neuron in the first hidden layer is equal to the size of the kernel.

The receptive field of a neuron in the next layer can be much larger.

When the stride is 1, the receptive field grows linearly i.e. a node in the mth hidden layer has the field of size $(l-1)m+1$ where l is the kernel size and s is the stride.

When the stride is larger than 1, the receptive field grows exponentially. The size is $O(ls^m)$ at the mth hidden layer.



4.3.3 Pooling Layer

A pooling layer works in a similar manner as a convolutional layer in order to extend the receptive field without trainable parameters.

- **Average pooling** finds the average value within a moving window. It works exactly the same as conducting a convolution operation with a uniform kernel array.

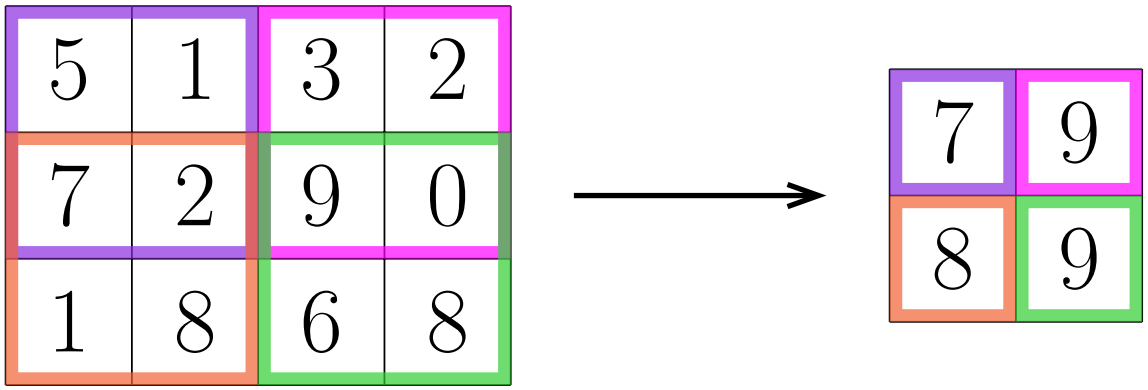
  For example, $\mathbf{k} = \begin{bmatrix} 1/l & 1/l & \dots & 1/l \end{bmatrix}^\top$ when the size of $\mathbf{k}$ is l.

  The average-pooling layer makes *multiscale recognition* possible.

- **Max pooling** finds the maximum value within a moving window. It works as a kind of logical disjunction.

  For example, when a trained kernel returns a black dot if a pattern is found in an area. A max-pooling layer summarizes that there exists a pattern in a greater area.

Example 4.8. Given an array with shape 3×4, conduct the max pooling operation using a window of shape 2×2 and stride 1×2.



4.3.4 Implementation of Convolutional Neural Networks

```
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, AvgPool2D, \
        Flatten, Dense

# Loading the dataset
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocessing the dataset
x_train = tf.reshape(x_train, (-1, 28, 28, 1))
x_test = tf.reshape(x_test, (-1, 28, 28, 1))

X_train = x_train/255
X_test = x_test/255
Y_train = tf.one_hot(y_train, 10)
Y_test = tf.one_hot(y_test, 10)

```

```
18 # Defining a model
19 model = tf.keras.Sequential([
20     Conv2D(6,
21            kernel_size=(5, 5), padding='same',
22            activation='sigmoid',
23            input_shape=(28, 28, 1)),
24     AvgPool2D(pool_size=(2, 2), padding='same',
25               strides=(2, 2)),
26     Conv2D(16,
27            kernel_size=(5, 5), padding='valid',
28            activation='sigmoid'),
29     AvgPool2D(pool_size=(2, 2), padding='same',
30               strides=(2, 2)),
31     Flatten(),
32     Dense(120, activation='sigmoid'),
33     Dense(84, activation='sigmoid'),
34     Dense(10, activation='softmax')
35     ])
36 model.summary()
```

```
# Specifying a training algorithm and loss function
model.compile(optimizer="adam",
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Training the model
model.fit(X_train, Y_train,
          epochs=100,
          batch_size=512,
          shuffle=True,
          validation_split=0.2)

# Save trained weights
model.save_weights("mnist_lenet.h5")

# Evaluating the model
r = model.evaluate(X_test, Y_test)
print(f"Test accuracy = {r[1]:.4f}")
```

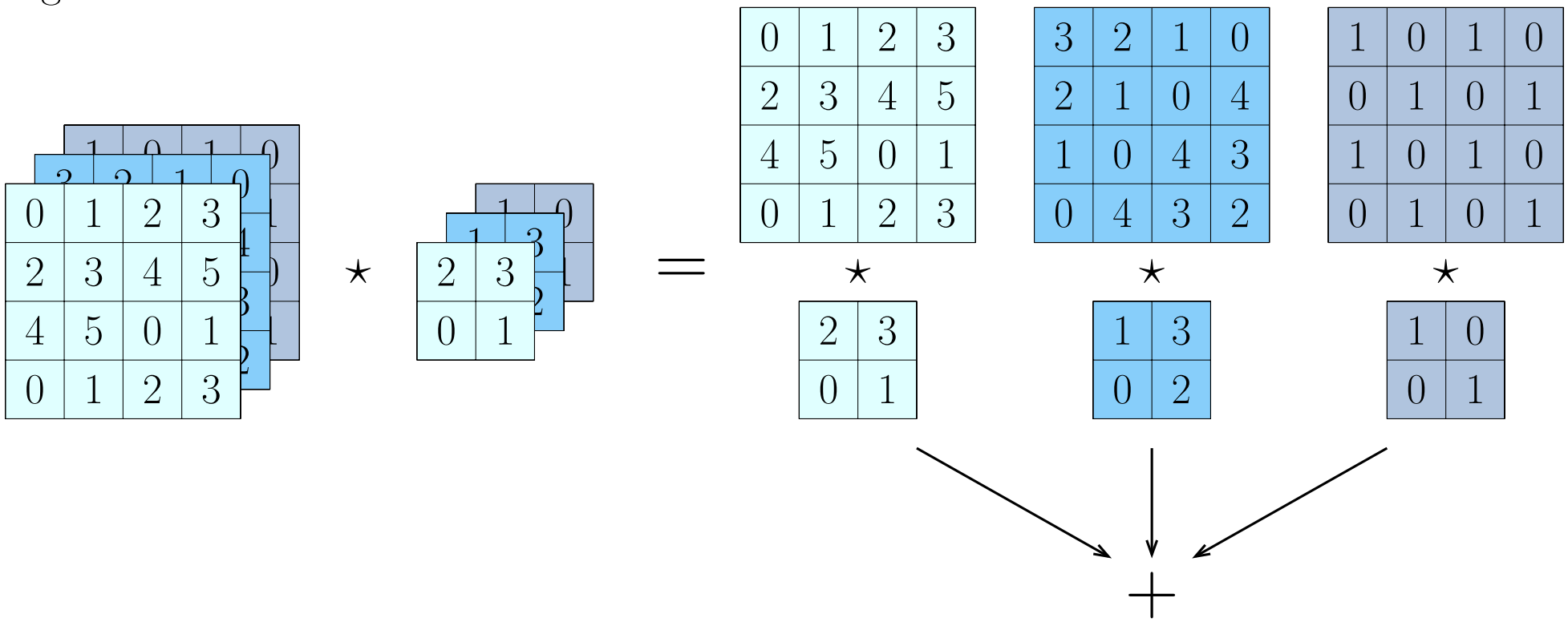
```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d (Conv2D)              (None, 28, 28, 6)         156
 average_pooling2d            (None, 14, 14, 6)         0
 conv2d_1 (Conv2D)            (None, 10, 10, 16)        2416
 average_pooling2d_1          (None, 5, 5, 16)          0
 flatten (Flatten)            (None, 400)               0
 dense (Dense)                (None, 120)               48120
 dense_1 (Dense)              (None, 84)                10164
 dense_2 (Dense)              (None, 10)                850
=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
_________________________________________________________________
...
Test accuracy = 0.9872
```

This is the first convolutional neural network proposed by LeCun et al. It is called '*LeNet*'.

4.3.5 Multiple Input Channels

When the input arrays contains multiple channels, or the shape is $h \times w \times c$ with $c > 1$, a kernel of shape $k_h \times k_w$ is created for each channel.

The cross-correlation operation is conducted for every channel before adding all the results together.

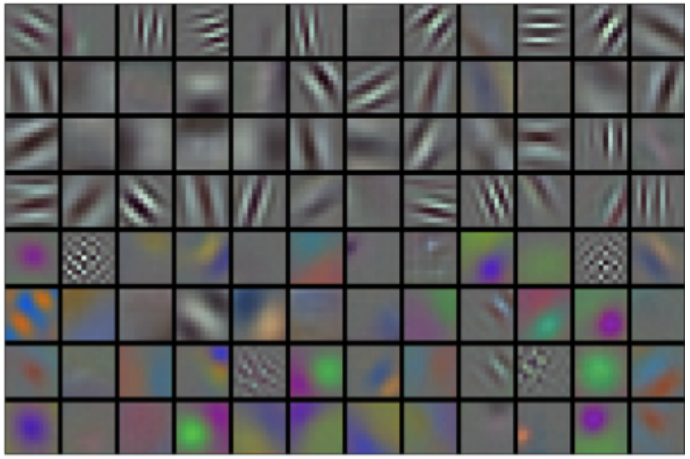


4.4 Deep Convolutional Neural Networks

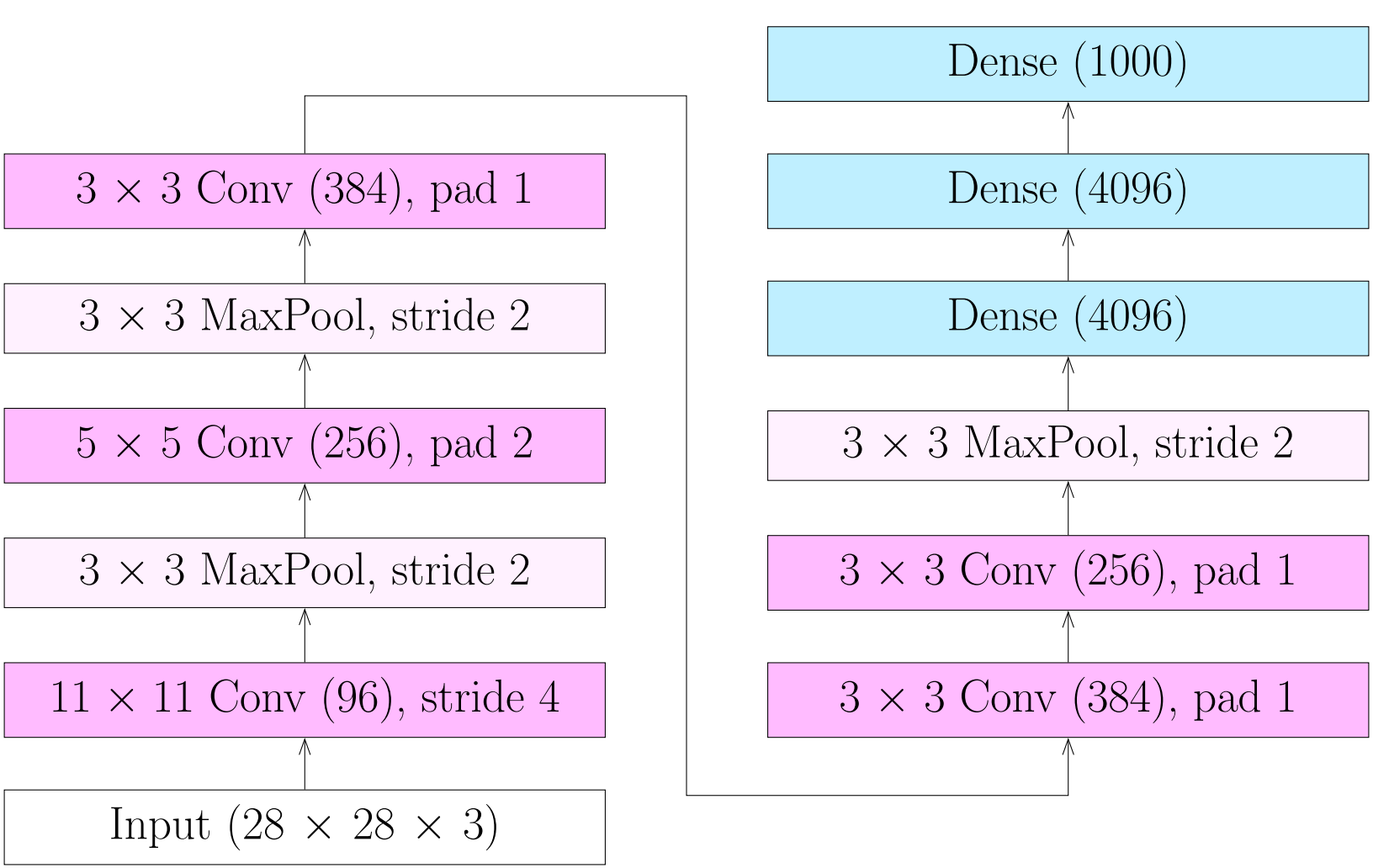
4.4.1 AlexNet

AlexNet is one of the first deep convolutional neural networks designed for the ImageNet challenge. It contains 5 convolutional layers with 3 pooling layers.

The idea of *representation learning* is introduced with AlexNet. The lowest layers are learned to extract features. Then, higher layers are learned to represent larger structures from those features.



Filters learned by the first layer of AlexNet


Input (28 × 28 × 3)
11 × 11 Conv (96), stride 4
3 × 3 MaxPool, stride 2
5 × 5 Conv (256), pad 2
3 × 3 MaxPool, stride 2
3 × 3 Conv (384), pad 1
3 × 3 Conv (384), pad 1
3 × 3 Conv (256), pad 1
3 × 3 MaxPool, stride 2
Dense (4096)
Dense (4096)
Dense (1000)

- AlexNet uses the kernel of shape 11×11 in the first layer to cope with a larger image in the ImageNet dataset.
- LeNet uses the sigmoid activation function. AlexNet changes to use the ReLU activation function.
- AlexNet uses dropout in the fully-connected (dense) layers to handle overfitting.

Example 4.9. Complete the following part of program to define AlexNet using Keras Sequential API.

```
model = tf.keras.Sequential([
  Cov2D(96, kernel_size=11, activation='relu'),
  MaxPool2D(pool_size=3, strides=2),
```

4.4.2 Visual Geometry Group (VGG) Network

From the structures of LeNet and AlexNet, CNNs are composed of a sequence of the followings:

(1) a convolution layer with padding and a nonlinear activiation,

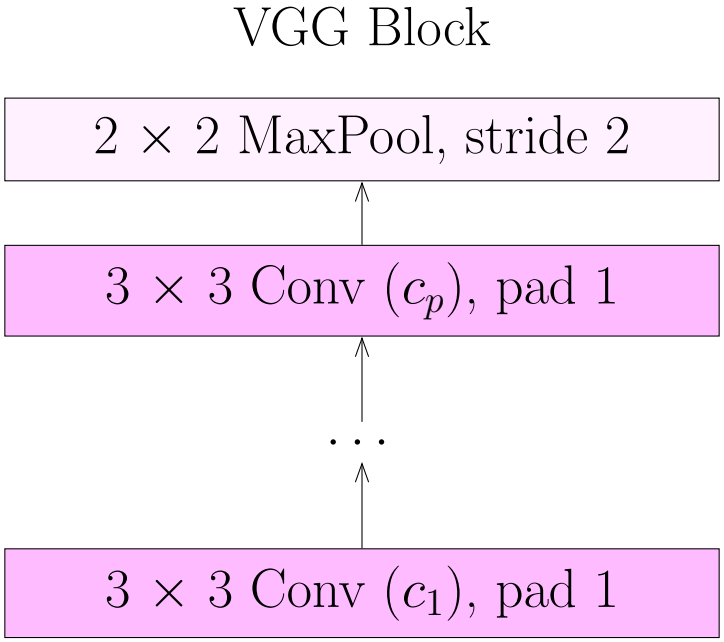
(2) a pooling layer with stride of 2.

Since each pooling layer reduces the resolution by half, the maximum number of these convolution+pooling layers is $\log_2 d$ where d is the dimension of the input array.
However, deeper networks significantly performs better than shallow networks.
Simonyan and Zisserman from the Visual Geomtry Group therefore propose *VGG block*.

VGG Block is defined as

(1) a sequence of convolution layers with 3×3 kernels and padding of 1,

(2) a max pooling layer with 2×2 window and stride of 2.



VGG Network is defined as a sequence of VGG blocks with increasing number of channels followed by 3 dense layers.
VGG-16 is composed of 5 VGG blocks followed by 3 dense layers. Therefore, it contains the total of 16 weight layers.

- *Block 1*: 2 convolutional layers with 64 channels
- *Block 2*: 2 convolutional layers with 128 channels
- *Block 3*: 3 convolutional layers with 256 channels
- *Block 4*: 3 convolutional layers with 512 channels
- *Block 5*: 3 convolutional layers with 512 channels

4.5 Residual Networks

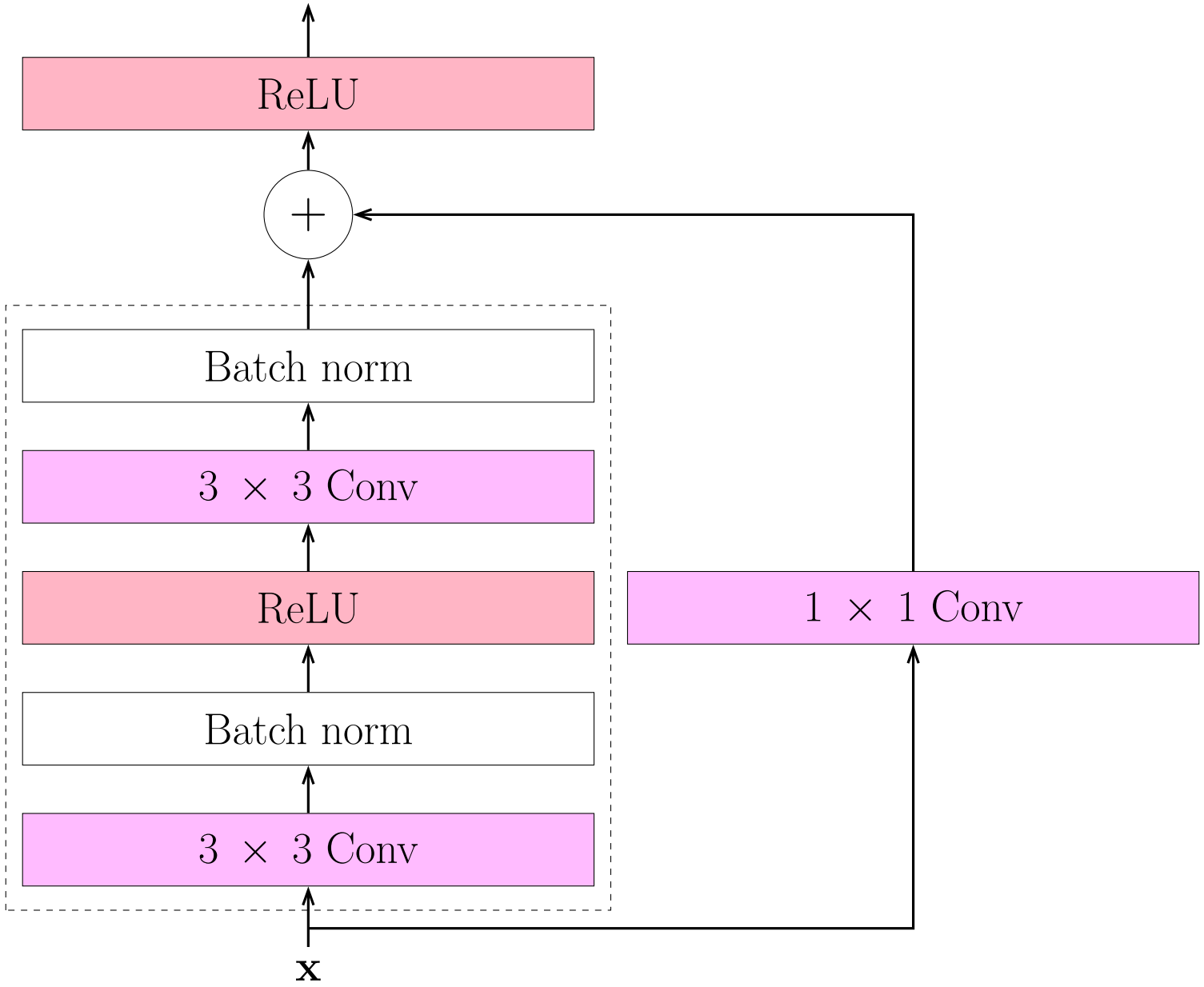
Typically, the output at layer i completely replaces the output at layer $i-1$.

$$\mathbf{a}_i = \mathbf{g}_i(\mathbf{W}_i\mathbf{a}_{i-1})$$

Therefore, all of the layers must learn to keep the meaningful information from the preceding layer while introducing the new information at the same time.
Residual network introduces an idea that a layer should *perturb* the representation from the previous layer instead of replacing it completely:

$$\mathbf{a}_i = \mathbf{g}_i^r(\mathbf{a}_{i-1} + \mathbf{g}_i(\mathbf{W}_i\mathbf{a}_{i-1}))$$

where $\mathbf{g}^r$ is the activation function for the residual layer.


ReLU
+
Batch norm
3 × 3 Conv
ReLU
Batch norm
3 × 3 Conv
1 × 1 Conv
x

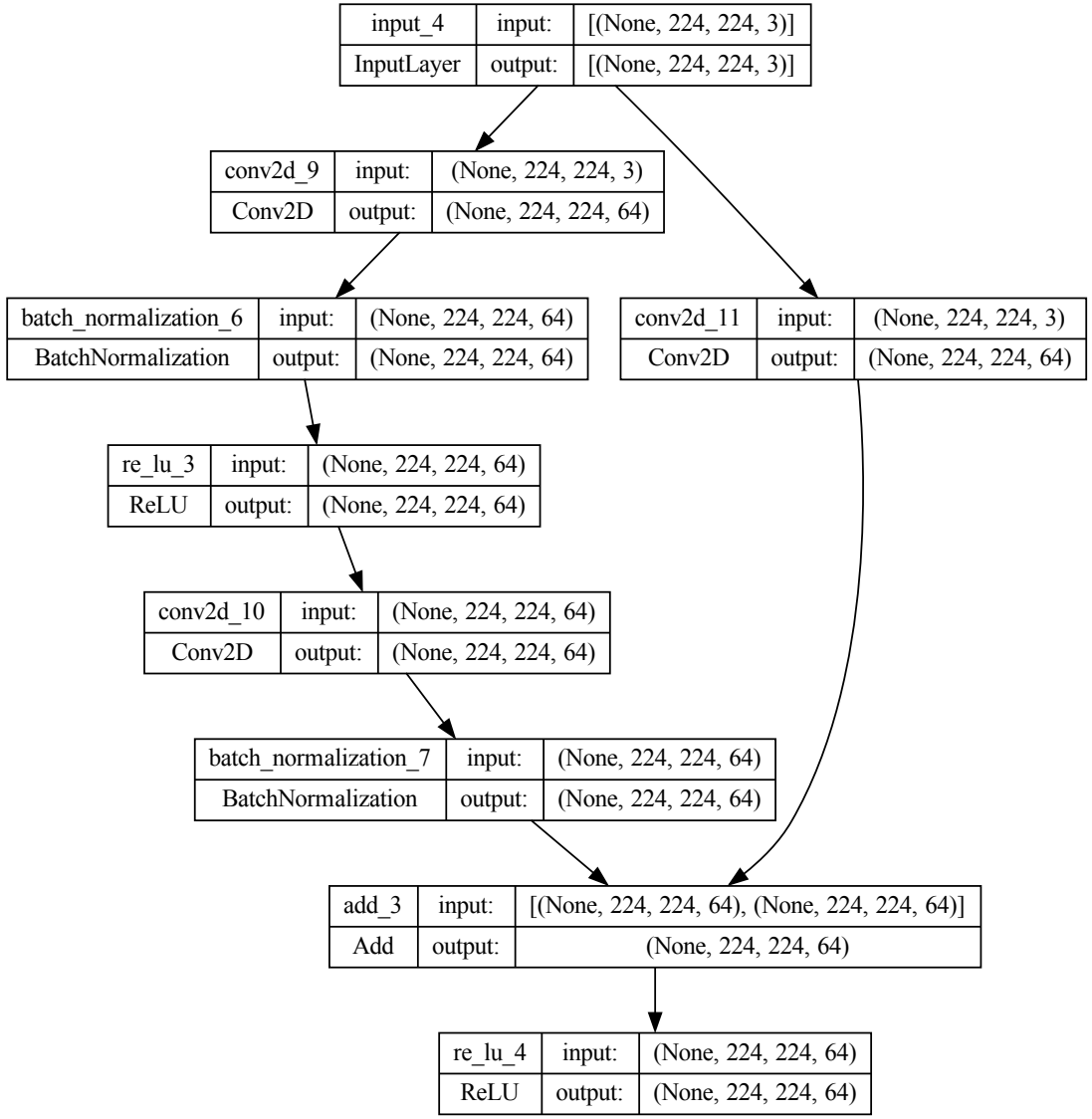
4.6 Keras Functional API

The Keras Functional API is a more flexible way to define a neural network in Keras. Each layer is considered a function accepting another layer as an argument.

Example 4.10. Define the residual block using Keras Functional API

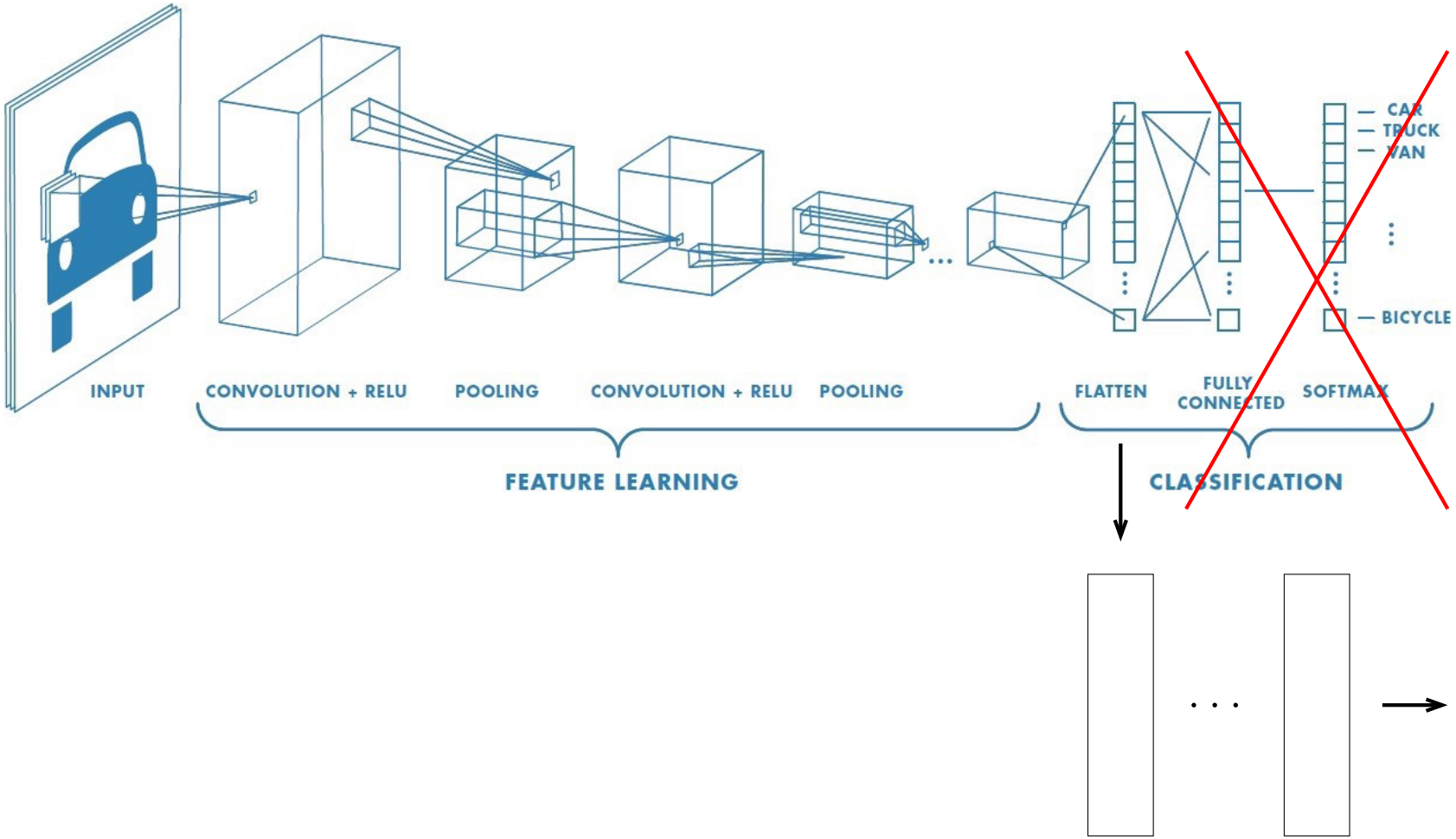
```
1  import tensorflow as tf
2  from tensorflow.keras.layers import Input, Dense, \
3      Conv2D, BatchNormalization, ReLU, Add
4  from tensorflow.keras.models import Model
5
6  input = Input(shape=(224, 224, 3))
7
8  x1 = Conv2D(64, kernel_size=3, strides=1, padding='same')(input)
9  x1 = BatchNormalization()(x1)
10 x1 = ReLU()(x1)
11 x1 = Conv2D(64, kernel_size=3, strides=1, padding='same')(x1)
12 x1 = BatchNormalization()(x1)
13
```

```
14 x2 = Conv2D(64, kernel_size=1, strides=1, padding='same')(input)
15
16 x = Add()([x1, x2])
17 output = ReLU()(x)
18
19 model = Model(inputs=input, outputs=output)
20
21 # model.summary()
22 tf.keras.utils.plot_model(model, "residual_block.pdf",
23     show_shapes=True)
```


input_4 | input: | [(None, 224, 224, 3)]
InputLayer | output: | [(None, 224, 224, 3)]
conv2d_9 | input: | (None, 224, 224, 3)
Conv2D | output: | (None, 224, 224, 64)
batch_normalization_6 | input: | (None, 224, 224, 64)
BatchNormalization | output: | (None, 224, 224, 64)
conv2d_11 | input: | (None, 224, 224, 3)
Conv2D | output: | (None, 224, 224, 64)
re_lu_3 | input: | (None, 224, 224, 64)
ReLU | output: | (None, 224, 224, 64)
conv2d_10 | input: | (None, 224, 224, 64)
Conv2D | output: | (None, 224, 224, 64)
batch_normalization_7 | input: | (None, 224, 224, 64)
BatchNormalization | output: | (None, 224, 224, 64)
add_3 | input: | [(None, 224, 224, 64), (None, 224, 224, 64)]
Add | output: | (None, 224, 224, 64)
re_lu_4 | input: | (None, 224, 224, 64)
ReLU | output: | (None, 224, 224, 64)

4.7 Transfer Learning

A deep neural network can be separated into two parts: feature learning and classification parts. The feature learning part of a model trained with a huge dataset can then be reused on a slightly different application with a smaller dataset. The idea of reusing an existing neural network for a similar application is called *transfer learning*.


CAR
TRUCK
VAN
BICYCLE
INPUT
CONVOLUTION + RELU
POOLING
CONVOLUTION + RELU
POOLING
FLATTEN
FULLY CONNECTED
SOFTMAX
FEATURE LEARNING
CLASSIFICATION
$\hat{y}$

Here are the steps for transfer learning:

1. Select a deep neural network built from a huge dataset for a similar application.
2. Remove the classification part of the selected network.
3. Add new layers for our new problem.
4. Freeze the parameters (weights and biases) of the first network.
5. Train the newly added layers using our dataset.
6. Fine tune the network by unfreezing the parameters and training the entire network with a very small learning rate.

```
import tensorflow as tf
import tensorflow_datasets as tfds

# Load the "cats vs dogs" dataset using tensorflow_datasets
# Split the dataset into 3 subsets:
#    70% for training,
#    15% for validation,
#    15% for testing
d_train, d_valid, d_test = tfds.load("cats_vs_dogs",
    split=["train[:70%]", "train[70%:85%]", "train[85%:100%]"],
    as_supervised=True)

# Display the number of examples in each subset
print("n_train = %d" % tf.data.experimental.cardinality(d_train))
print("n_valid = %d" % tf.data.experimental.cardinality(d_valid))
print("n_test  = %d" % tf.data.experimental.cardinality(d_test))


```

```
19 # Preprocess
20 #
21 # Resize each image into 64x64
22 size = (64, 64)
23 d_train = d_train.map(lambda x, y: (tf.image.resize(x, size), y))
24 d_valid = d_valid.map(lambda x, y: (tf.image.resize(x, size), y))
25 d_test = d_test.map(lambda x, y: (tf.image.resize(x, size), y))
26
27 # Normalize the color values
28 d_train = d_train.map(lambda x, y: (x / 255.0, y))
29 d_valid = d_valid.map(lambda x, y: (x / 255.0, y))
30 d_test = d_test.map(lambda x, y: (x / 255.0, y))
31
32 # Set up batches
33 batch_size = 256
34 d_train = d_train.cache().batch(batch_size).prefetch(buffer_size=10)
35 d_valid = d_valid.cache().batch(batch_size).prefetch(buffer_size=10)
36 d_test = d_test.cache().batch(batch_size).prefetch(buffer_size=10)
37
```

```
38 # Load the VGG16 model and use it as the base model
39 # Use the weights trained by the imagenet dataset
40 base_model = tf.keras.applications.VGG16(weights="imagenet",
41                                          input_shape=(64, 64, 3),
42                                          include_top=False)
43 # Freeze the weights of VGG16
44 base_model.trainable = False
45
46 # Set up the classification part of the model
47 inputs = tf.keras.Input(shape=(64, 64, 3))
48 x = base_model(inputs, training=False)
49 x = tf.keras.layers.GlobalMaxPooling2D()(x)
50 x = tf.keras.layers.Dropout(0.3)(x)
51 x = tf.keras.layers.Dense(128, activation='relu',
52         kernel_regularizer=tf.keras.regularizers.l2(1e-5))(x)
53 outputs = tf.keras.layers.Dense(1, activation='sigmoid')(x)
54
55 model = tf.keras.Model(inputs=inputs, outputs=outputs)
56
```

```
model.compile(optimizer=tf.keras.optimizers.Adam(3e-4),
    loss=tf.keras.losses.BinaryCrossentropy(),
    metrics=['accuracy'])

model.summary()

# Train the model
model.fit(d_train, epochs=20, batch_size=batch_size,
          shuffle=True, validation_data=d_valid)

# Evaluate the model using the test set
model.evaluate(d_test, verbose=0)

r = model.evaluate(d_test, verbose=0)
print(f"Test accuracy 1 = {r[1]:.4f}")

# Fine tune the model
base_model.trainable = True
model.summary()
```

```
76
77 # The learning rate is set to a very small value
78 model.compile(optimizer=tf.keras.optimizers.Adam(1e-6),
79     loss=tf.keras.losses.BinaryCrossentropy(),
80     metrics=['accuracy'])
81
82 # Training the entire model
83 model.fit(d_train, epochs=10, batch_size=batch_size,
84           shuffle=True, validation_data=d_valid)
85
86 # Evaluate the model using the test set
87 r = model.evaluate(d_test, verbose=0)
88 print(f"Test accuracy 2 = {r[1]:.4f}")
```

Chapter 5

Unsupervised Learning

5.1 Unsupervised Learning

Unsupervised learning learns from a set of examples without labels/classes.

Tasks of an unsupervised learning are:

- to learn a function $h : \mathcal{X} \rightarrow \mathcal{Z}$ that maps an unlabeled input $\mathbf{x}$ into new representations.
- to learn a generative model such as a probability distribution to generate new examples.

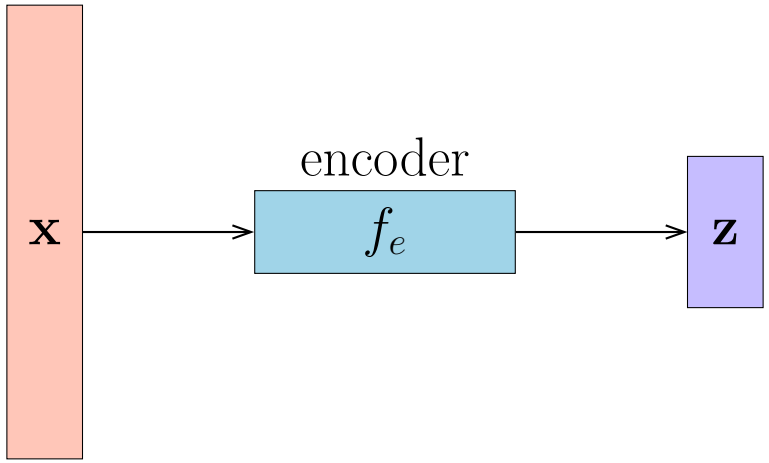
5.2 Autoencoder

Autoencoder is an unsupervised learning technique that learns *new representations* of data. For example,

- generate features of fingerprint images to make it easier to check if two fingerprints are from the same person,
- perform *dimension reduction* by transforming an input vector into a space with fewer dimensions.

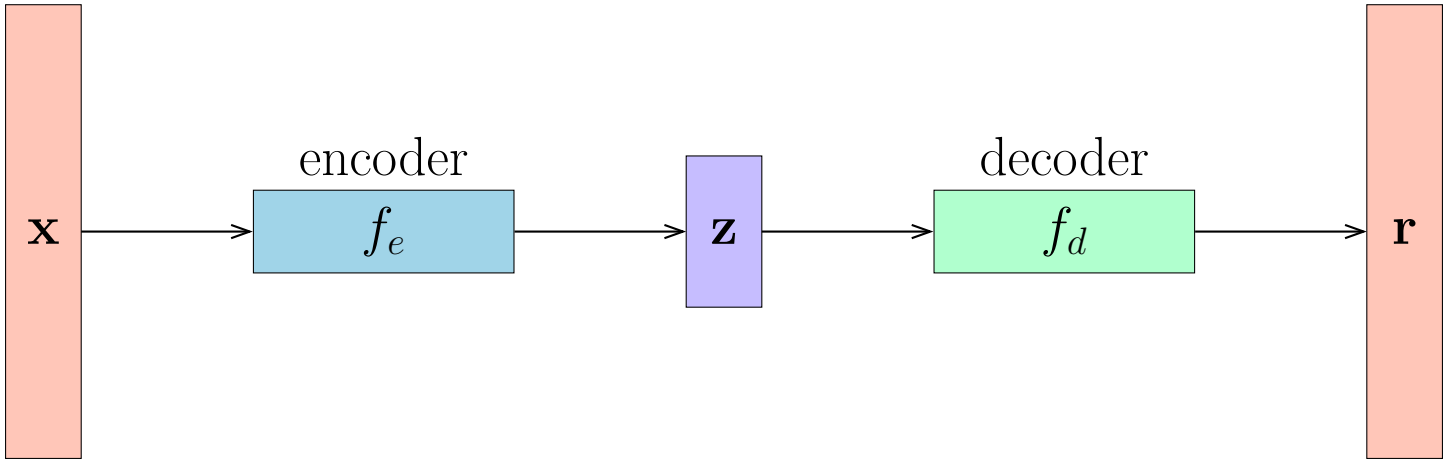
Autoencoder creates an encoder that maps an input array $\mathbf{x}$ into a new representation $\mathbf{z}$:

$$\mathbf{z} = f_e(\mathbf{x})$$



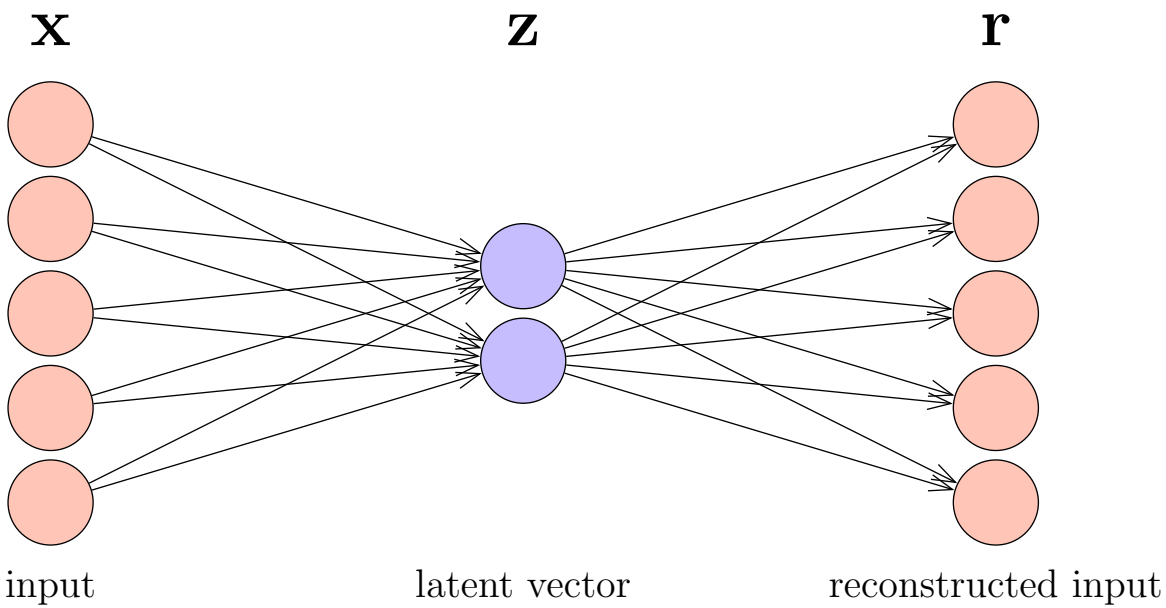
Since target outputs are not provided, we can train this model by creating another function that accepts the encoded array $\mathbf{z}$ and reconstructs the input array. $\mathbf{x}$:

$$\mathbf{r} = f_d(\mathbf{z}) = f_d(f_e(\mathbf{x}))$$



The encoder and decoder are connected and trained to reconstruct the input values. This is called *autoencoder*.

5.2.1 Fully-connected Autoencoder



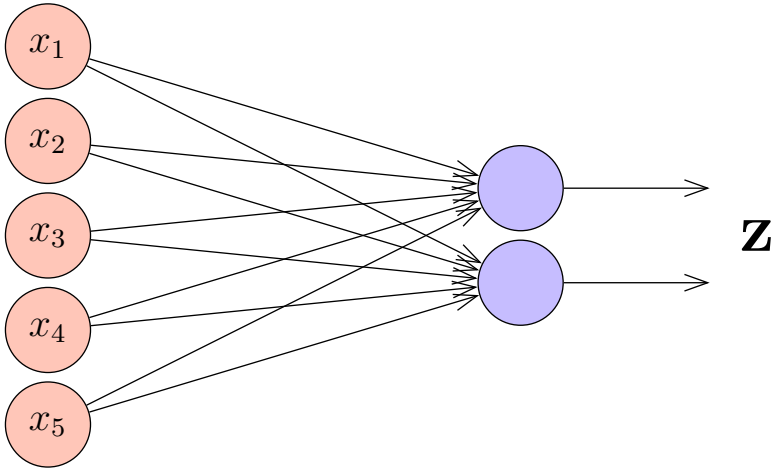
$$\mathbf{r} = g(\mathbf{W}_2\mathbf{z} + \mathbf{b}_2) = g(\mathbf{W}_2\left(g(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)\right) + \mathbf{b}_2)$$

This autoencoder can be trained using the mean squared error loss:

$$\begin{aligned}\mathcal{L}(\mathbf{x}, \mathbf{r}) &= \frac{1}{m} \|\mathbf{x} - \mathbf{r}\|_2^2 \\ &= \frac{1}{m} \sum_{i=1}^{m} (x_i - r_i)^2\end{aligned}$$

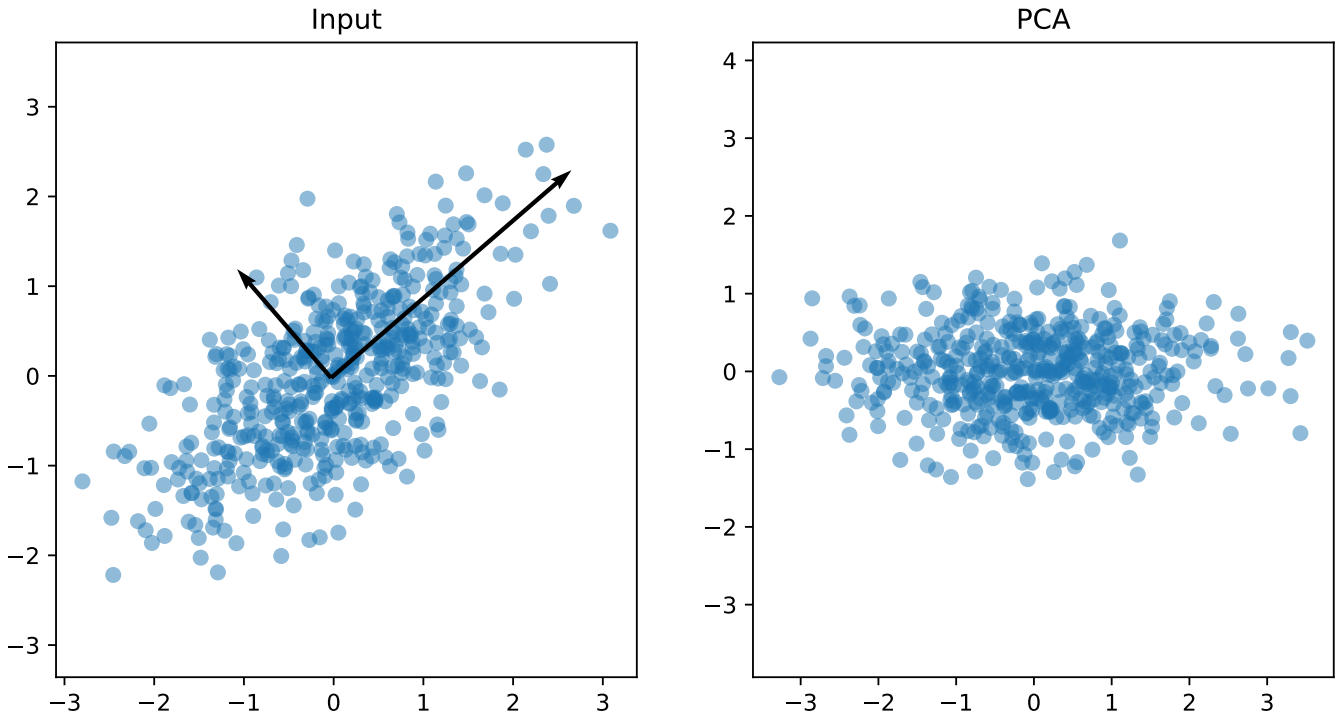
When $0 \leq x_i \leq 1$ for all $i = 1, \ldots, m$, the binary cross entropy can also be used as the loss function.

After complete the training step, the decoder can be removed.



When the activation function is set to the linear function, this autoencoder works almost identically to *Principal Component Analysis* (PCA) technique.

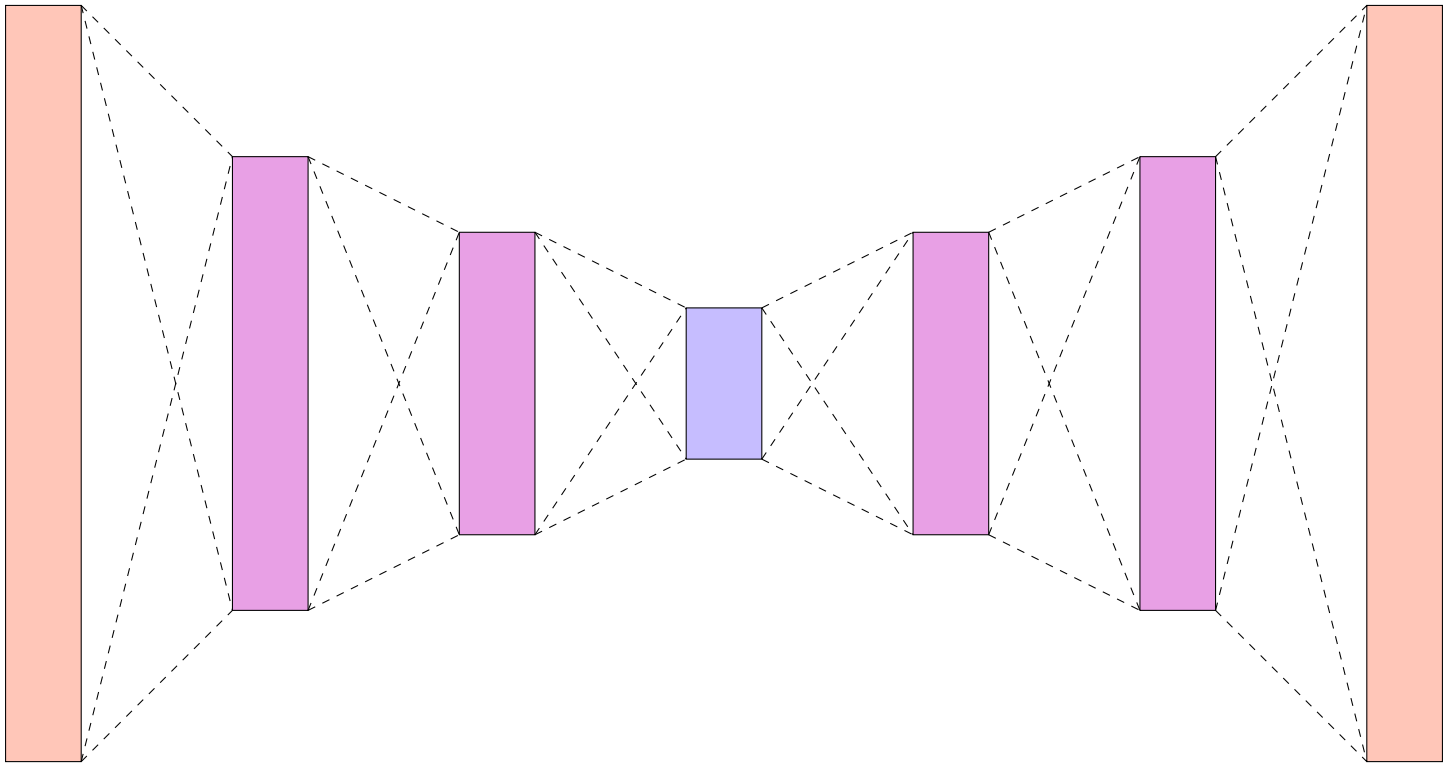
However, the encoder extracted from an autoencoder does not guarantee orthoginal components.



Implementation of Autoencoder

```
from tensorflow.keras.layers import Layer, Input, Dense
from tensorflow.keras.models import Model

inputs = Input(shape=(5,))
encoded = Dense(2, activation='linear')(inputs)
decoded = Dense(5, activation='linear')(encoded)

encoder = Model(inputs=inputs, outputs=encoded)
autoencoder = Model(inputs=inputs, outputs=decoded)
autoencoder.compile(optimizer='adam', loss='mse')

autoencoder.summary()
```

We can introduce multiple hidden layers with nonlinear activation functions into an autoencoder.



Example 5.1. From the MNIST dataset, we construct a fully-connected autoencoder. The encoder is composed of 4 layers with 256, 128, 64, and 32 units, respectively. Therefore, the latent vector $\mathbf{z}$ has 32 elements.

Fill in the following table with the number of units and activation functions of the decoder.

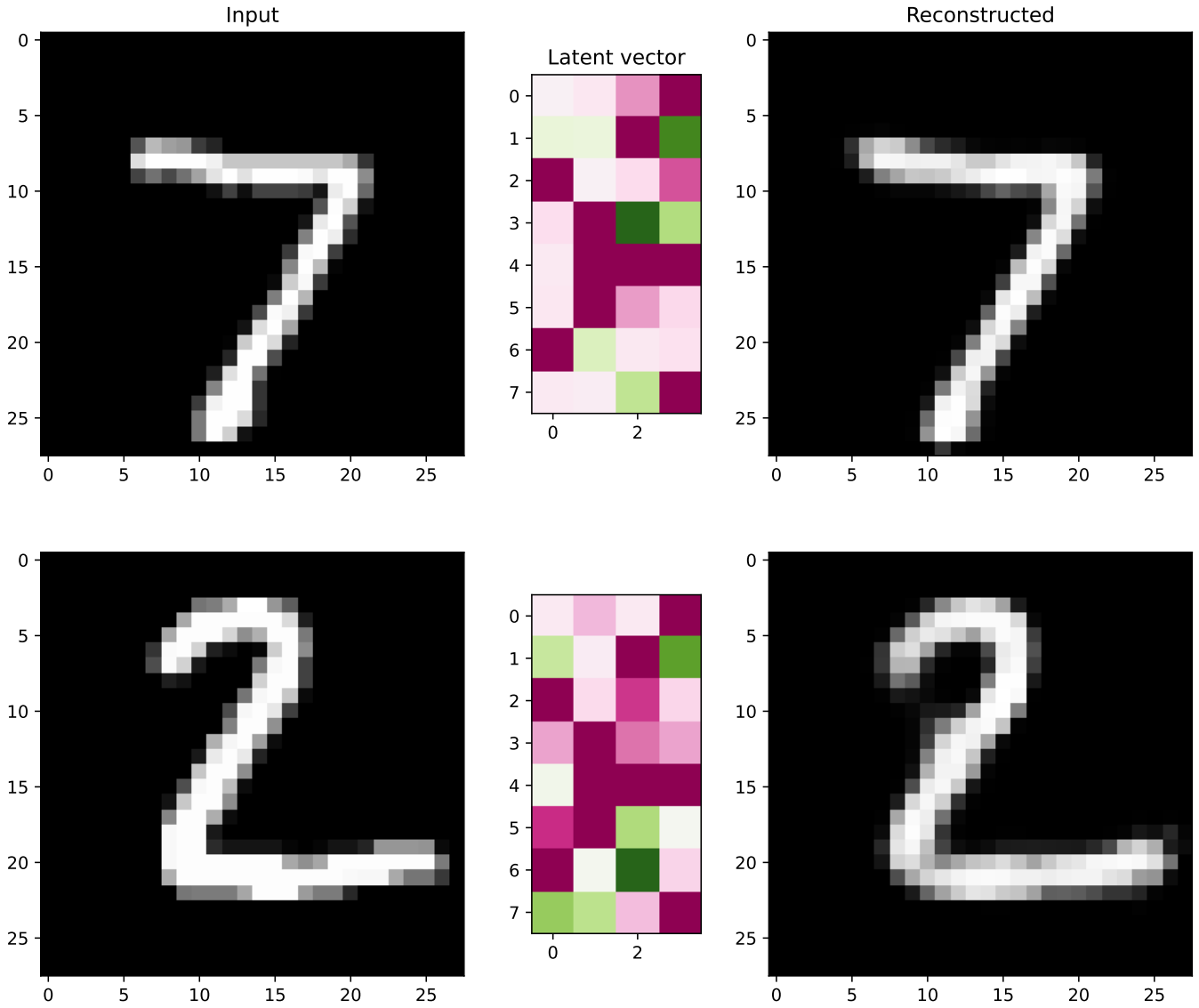
Layer Type	Units	Activation Function
Input	784	-
Dense	256	relu
Dense	128	relu
Dense	64	relu
Dense	32	relu
Dense		
Dense		
Dense		
Dense		

```
1  from tensorflow.keras.layers import Layer, Input, Dense
2  from tensorflow.keras.models import Model
3
4  inputs = Input(shape=(784,))
5  encoded = Dense(256, activation='relu')(inputs)
6  encoded = Dense(128, activation='relu')(encoded)
7  encoded = Dense(64, activation='relu')(encoded)
8  encoded = Dense(32, activation='relu')(encoded)
9
10 decoded = Dense(64, activation='relu')(encoded)
11 decoded = Dense(128, activation='relu')(decoded)
12 decoded = Dense(256, activation='relu')(decoded)
13 decoded = Dense(784, activation='sigmoid')(decoded)
14
15 encoder = Model(inputs=inputs, outputs=encoded)
16 autoencoder = Model(inputs=inputs, outputs=decoded)
17 autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
18
```

```
19 autoencoder.summary()
20 autoencoder.fit(X_train, X_train,
21                 epochs=100, batch_size=512,
22                 shuffle=True)
```

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 input_2 (InputLayer)         [(None, 784)]             0
 dense_8 (Dense)              (None, 256)               200960
 dense_9 (Dense)              (None, 128)               32896
 dense_10 (Dense)             (None, 64)                8256
 dense_11 (Dense)             (None, 32)                2080
 dense_12 (Dense)             (None, 64)                2112
 dense_13 (Dense)             (None, 128)               8320
 dense_14 (Dense)             (None, 256)               33024
 dense_15 (Dense)             (None, 784)               201488
=================================================================
```

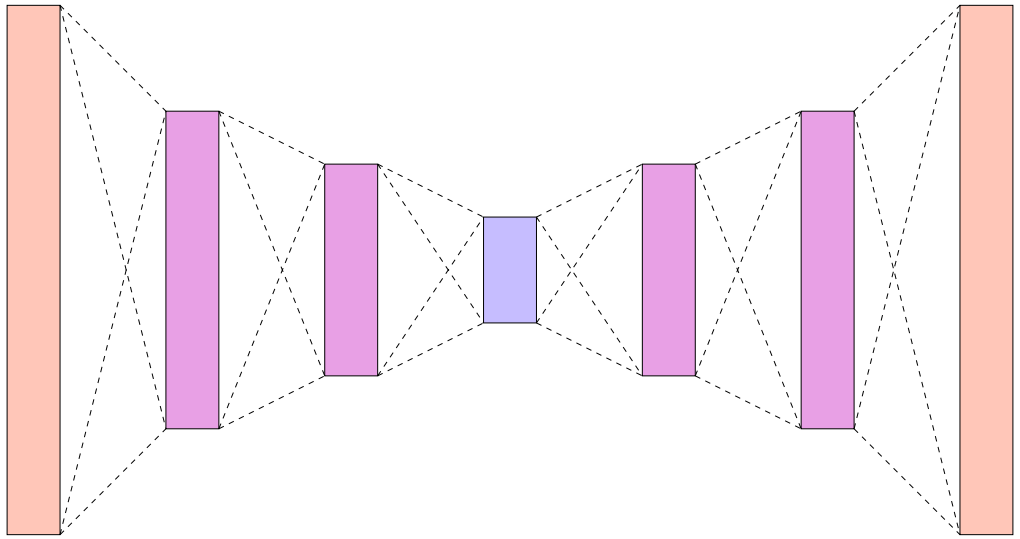
```
X1 = X_test[0:1, :]
Z1 = encoder.predict(X1)
R1 = autoencoder.predict(X1)
X2 = X_test[1:2, :]
Z2 = encoder.predict(X2)
R2 = autoencoder.predict(X2)
fig, ax = plt.subplots(ncols=3, nrows=2, figsize=(12, 10),
                       width_ratios=(5,2,5))
ax[0, 0].imshow(X1.reshape((28, 28)), cmap='gray')
ax[0, 1].imshow(Z1.reshape((8, 4)), cmap='PiYG')
ax[0, 2].imshow(R1.reshape((28, 28)), cmap='gray')
ax[1, 0].imshow(X2.reshape((28, 28)), cmap='gray')
ax[1, 1].imshow(Z2.reshape((8, 4)), cmap='PiYG')
ax[1, 2].imshow(R2.reshape((28, 28)), cmap='gray')
ax[0, 0].set_title('Input')
ax[0, 1].set_title('Latent vector')
ax[0, 2].set_title('Reconstructed')
plt.savefig("mnist_fc_autoencoder_output.pdf", bbox_inches='tight')
```


Input
Latent vector
Reconstructed

5.2.2 Convolutional Autoencoder

Convolutional autoencoder includes at least one convolutional layer.

Each layer in the decoding part needs to increase size of its input array in order to reconstruct the input.



Transposed Convolution

Given an input array with the shape of $n_h \times n_w$ and a kernel with the shape of $k_h \times k_w$, a transposed convolution operation is performed as follows:

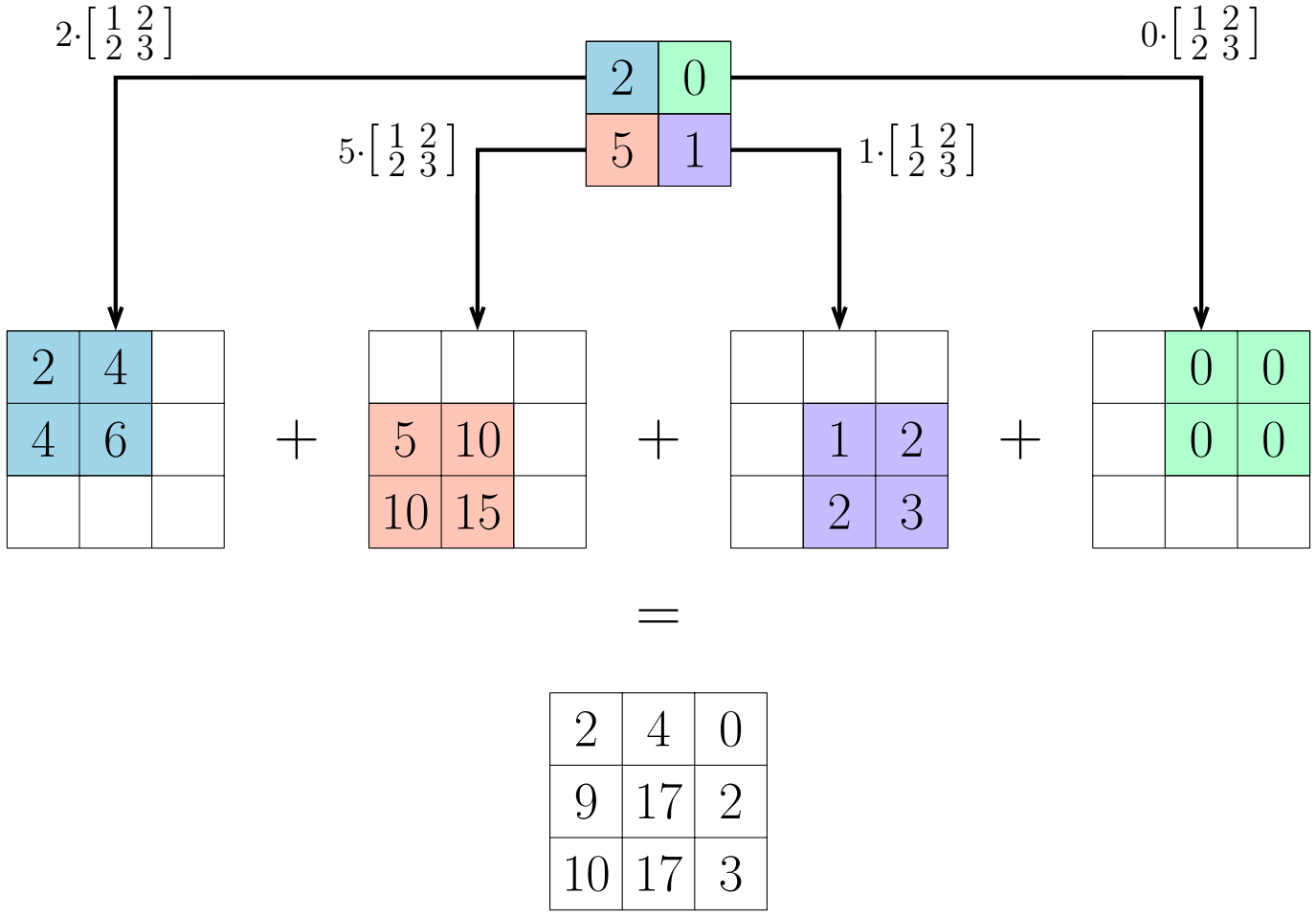
1. Create $n_h n_w$ intermediate array with the shape of $(n_h + k_h - 1) \times (n_w + k_w - 1)$, and initialize all of the elements to zeros.

2. Multiply each element of the input array with the kernel.

3. Replace a portion of an intermediate array with the multiplication result.

4. Sum all the intermediate arrays to create the output array.

Example 5.2. Given an input array $\left[\begin{smallmatrix} 2 & 0 \\ 5 & 1 \end{smallmatrix}\right]$ and a kernel $\left[\begin{smallmatrix} 1 & 2 \\ 2 & 3 \end{smallmatrix}\right]$, conduct a transposed convolution.



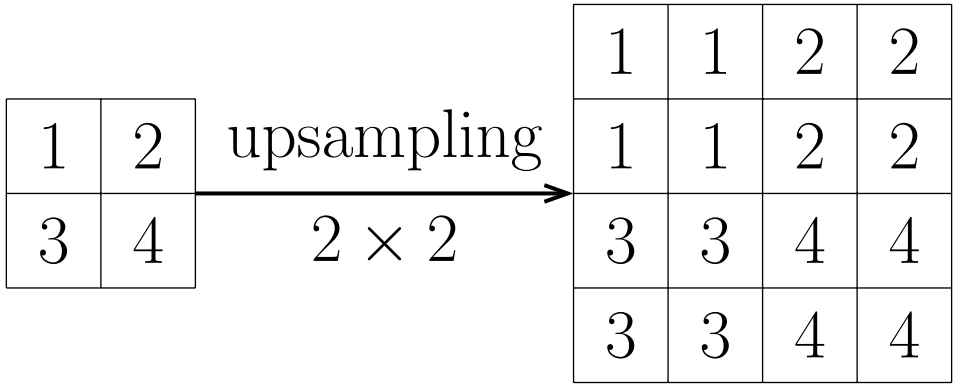
2	4	0
9	17	2
10	17	3

Example 5.3. Compute the output of a transposed convolution operation given the following input array $\mathbf{x}$ and kernel $\mathbf{k}$.

$$\mathbf{x} = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \qquad \mathbf{k} = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 1 & 3 \end{bmatrix}$$

Upsampling

Upsampling is an operation that conducts the reverse of a pooling operation.
When we performs an upsampling operation on a 2D array with size of $s_h \times s_w$,
the rows and columns of the input array are repeated by s_h and s_w, respectively.



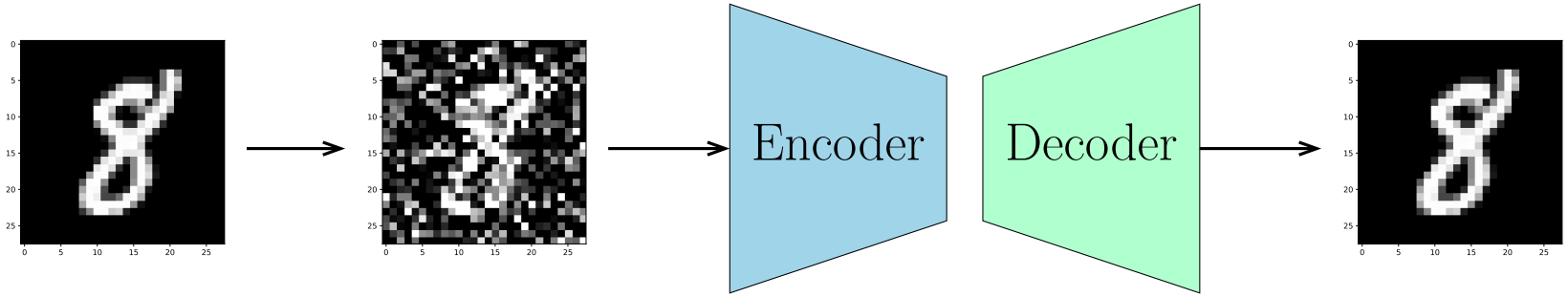
A combination of an upsampling and a convolutional layers can be used in place of a transposed convolutional layer.

5.2.3 Denoise Autoencoder

We can train an autoencoder to remove noise from the input by

1. adding noises into the input,
2. training the autoencoder using the original input as the target.

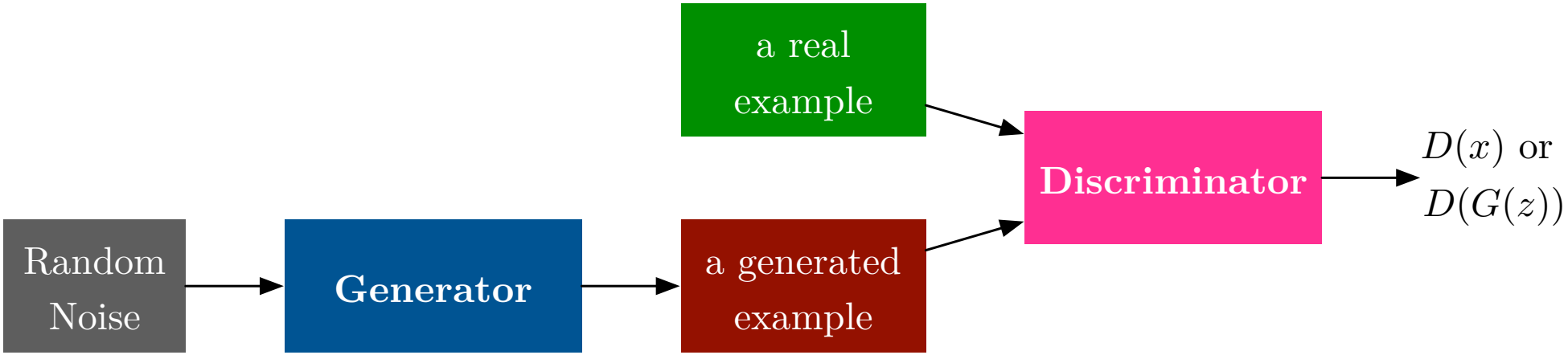
This type of autoencoders is called *denoise autoencoder*.



5.3 Generative Adversarial Network

A *Generative Adversarial Network (GAN)*, proposed by GoodFellow et al. in 2014, is a *generative model*. It is originally designed to generate new data.
A GAN is composed of two parts:

1. **generator** learns to generate fake data,
2. **discriminator** learns to determine whether an example is real or fake.



Minimax Loss

The following minimax loss function is used to train a GAN:

$$V(\theta_d, \theta_g) = \mathbb{E}_x\big[\log D(x)\big] + \mathbb{E}_z\big[\log(1 - D(G(z)))\big]$$

where

- x is a real example, z is a noise,
- $D(x)$ is the probability that x is real estimated by discriminator,
- $G(z)$ is an example generated from z,
- $D(G(z))$ is the probability that the generated example is real,
- $\mathbb{E}_x$ is the expected value over all the real examples,
- $\mathbb{E}_z$ is the expected value all the random noises.

The discriminator is trained to maximize the loss, while the generator is trained to minimize the loss, i.e. $\min_G \max_D V(\theta_d, \theta_g)$

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

end for

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

GAN Training Steps

1. Freeze the parameters of the generator,

    - Create a minibatch of real data.
    - Create a minibatch of noises and use the generator to synthesize fake data.
    - Use *gradient ascent* algorithm to update the parameters of the discriminator to maximize $V(\theta_d, \theta_g)$, or

    $$\text{maximize } \left[\frac{1}{n}\sum_{i=1}^{n}\left[\log\left(D(\mathbf{x}^{[i]})\right) + \log\left(1 - D(G(\mathbf{z}^{[i]}))\right)\right]\right]$$

This is equivalent to using the *gradient descent* algorithm to minimize the negative log likelihood, or

$$\text{minimize} \left[-\frac{1}{n} \sum_{i=1}^{n} \left[\log \left(D(\mathbf{x}^{[i]}) \right) + \log \left(1 - D(G(\mathbf{z}^{[i]})) \right) \right] \right]$$

Comparing to the binary cross entropy function:

$$\text{minimize} \left[-\frac{1}{n} \sum_{i=1}^{n} \left[y^{[i]} \log(\hat{y}) + (1 - y) \log(1 - \hat{y}) \right] \right]$$

Training the discriminator to maximize $V(\theta_d, \theta_g)$ is equivalent to training a ordinary neural network using the binary cross entropy loss function and setting the target output to

- 1 when feeding a real data,
- 0 when feeding a generated/fake data.

2. Freeze the parameters of the discriminator,

   - Create a minibatch of noises and use the generator to synthesize fake data.
   - Use *gradient descent* algorithm to update the parameters of the generator to minimize $V(\theta_d, \theta_g)$, or

     $$\text{minimize } \frac{1}{n}\sum_{i=1}^{n}\left[\log\left(1 - D(G(\mathbf{z}^{[i]}))\right)\right]$$

     At early training stage, the generator synthesizes data that does not look like real data. The discriminator can easily predict the generated data as a fake one with high confidence.

     This causes a vanishing gradient problem since the error needs to be propagated backward from the discriminator to the generator.

To avoid the saturation phenomenon, we can adjust the training process to:

$$\text{maximize} \quad \left[\frac{1}{n}\sum_{i=1}^{n}\left[\log\left(D(G(\mathbf{z}^{[i]}))\right)\right]\right]$$

This is equivalent to:

$$\text{minimize} \quad \left[-\frac{1}{n}\sum_{i=1}^{n}\left[\log\left(D(G(\mathbf{z}^{[i]}))\right)\right]\right]$$

This is also equivalent to using the binary cross entropy loss function when we set the target output to 1. Therefore, the generator is trained to maximize the probability of data being predicted as real.

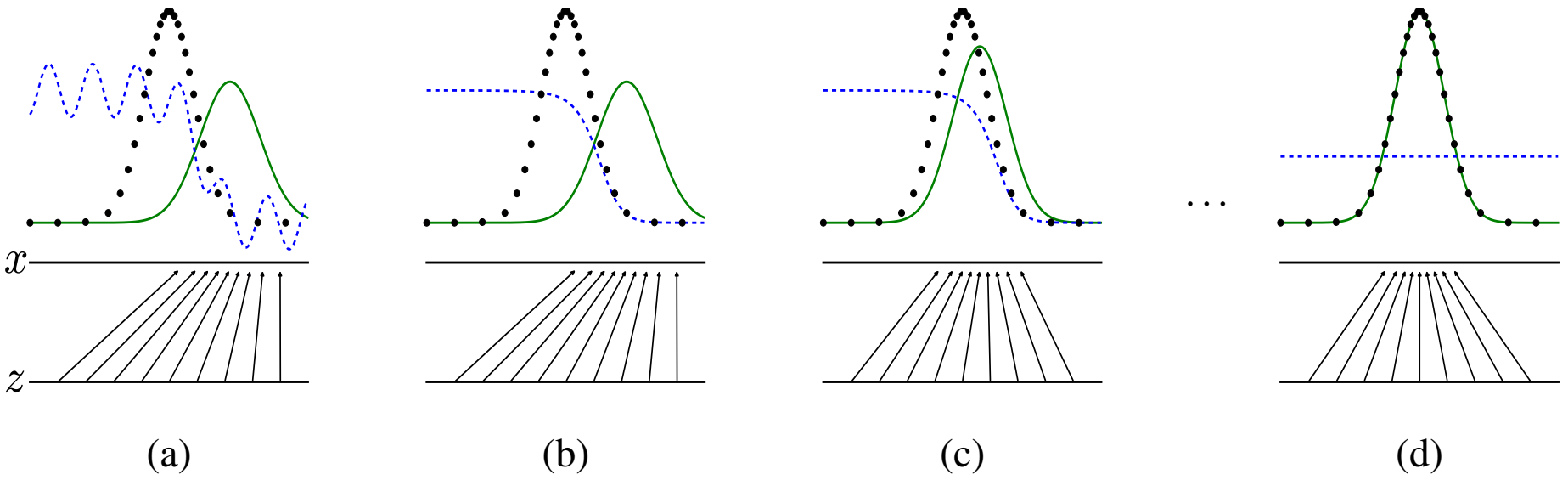
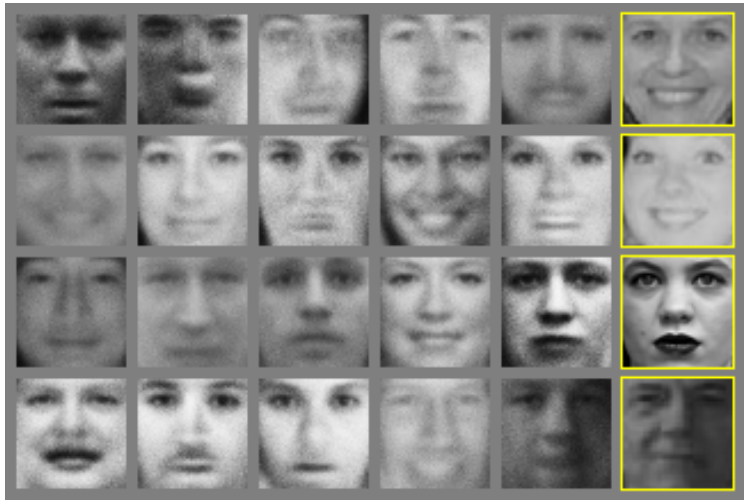
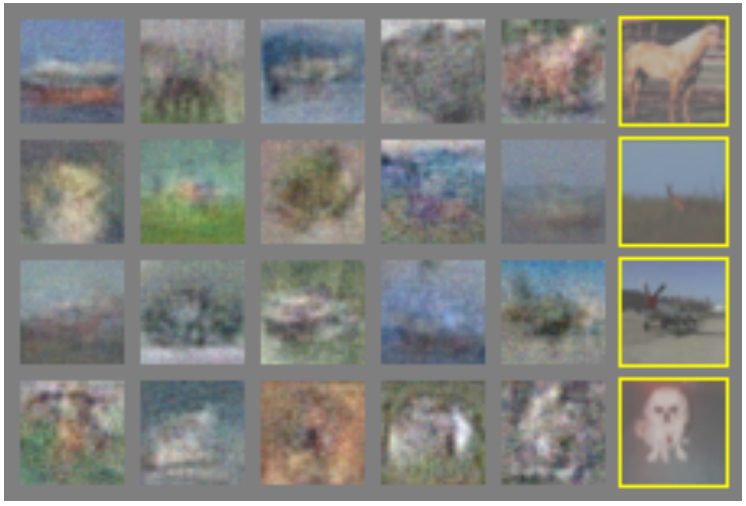


Figure 1: Generative adversarial nets are trained by simultaneously updating the **d**iscriminative distribution (D, blue, dashed line) so that it discriminates between samples from the data generating distribution (black, dotted line) $p_{\boldsymbol{x}}$ from those of the **g**enerative distribution p_g (G) (green, solid line). The lower horizontal line is the domain from which $\boldsymbol{z}$ is sampled, in this case uniformly. The horizontal line above is part of the domain of $\boldsymbol{x}$. The upward arrows show how the mapping $\boldsymbol{x} = G(\boldsymbol{z})$ imposes the non-uniform distribution p_g on transformed samples. G contracts in regions of high density and expands in regions of low density of p_g. (a) Consider an adversarial pair near convergence: p_g is similar to p_{data} and D is a partially accurate classifier. (b) In the inner loop of the algorithm D is trained to discriminate samples from data, converging to $D^*(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})}$. (c) After an update to G, gradient of D has guided $G(\boldsymbol{z})$ to flow to regions that are more likely to be classified as data. (d) After several steps of training, if G and D have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{\text{data}}$. The discriminator is unable to differentiate between the two distributions, i.e. $D(\boldsymbol{x}) = \frac{1}{2}$.

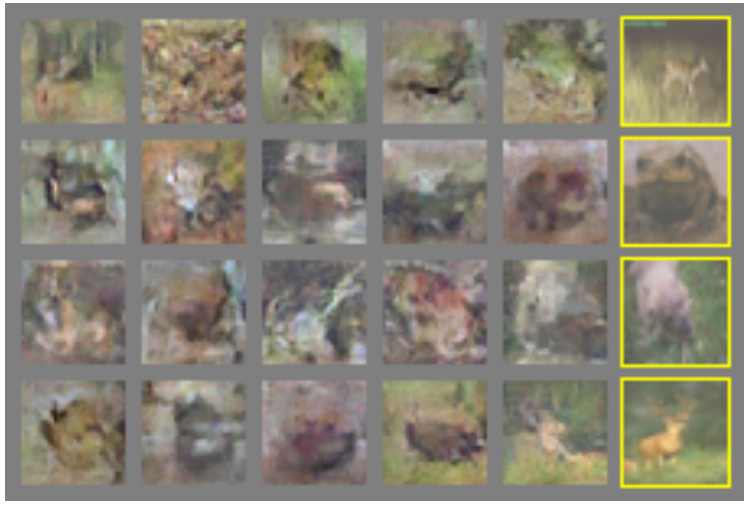
a)



b)

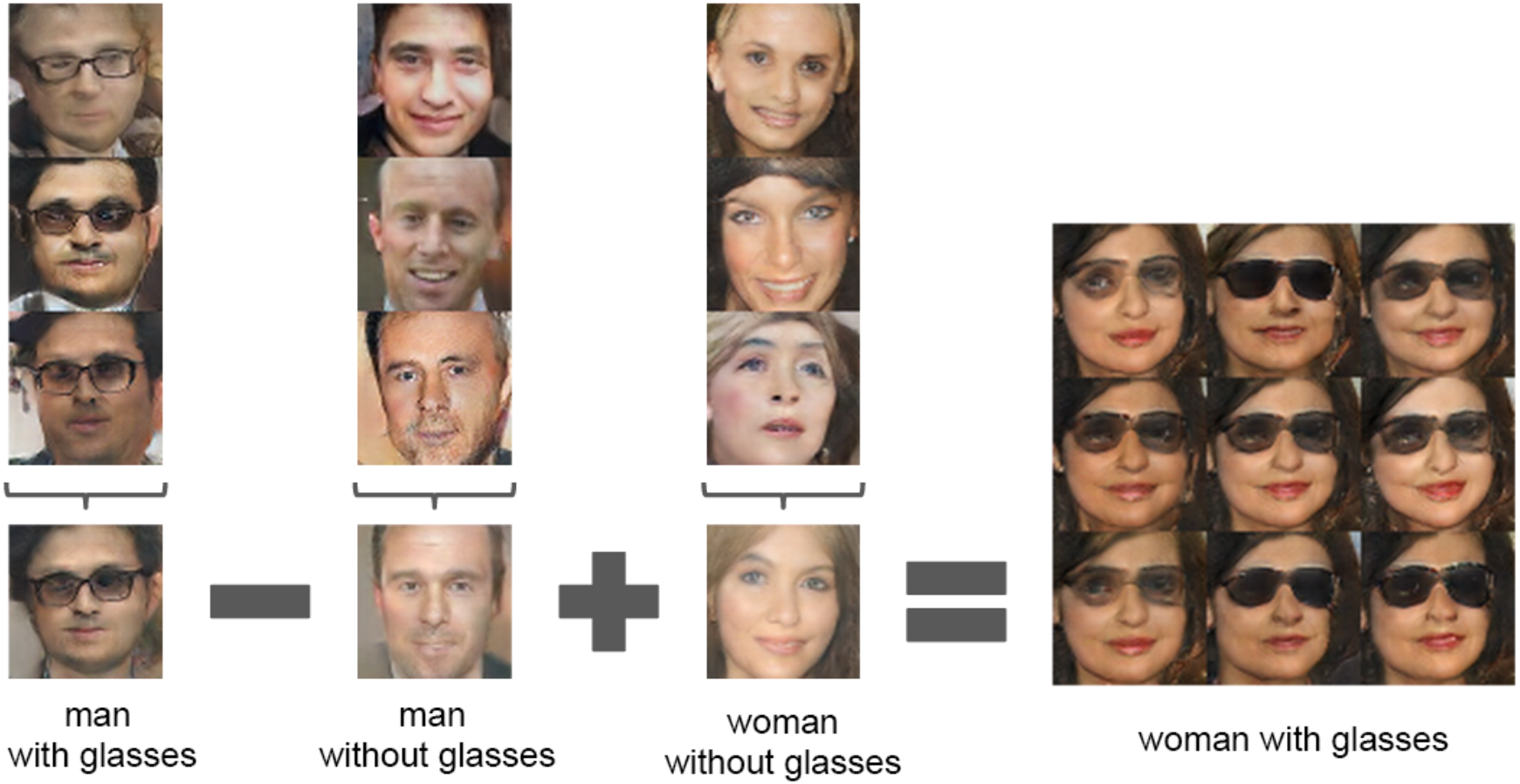


c)



d)

Vector Arithmetic



References

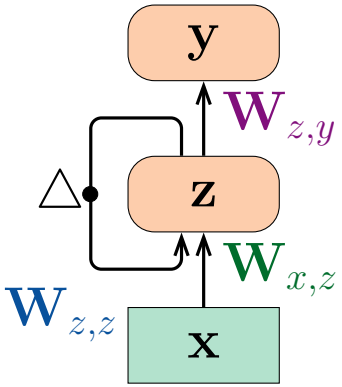
- Goodfellow et al. "Generative Adversarial Nets." Advances in Neural Information Processing Systems. 2014.
- Radford et al. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks." ICLR 2016.

Chapter 6

Recurrent Neural Networks

6.1 Recurrent Neural Networks

A **Recurrent Neural Network (RNN)** is a type of neural networks that allows cycles in the computation graph. This is different from a *feedforward neural network*.



- $\mathbf{x}$ is the input layer, $\mathbf{y}$ is the output layer.
- $\mathbf{z}$ is the hidden layer with recurrent connections. $\triangle$ indicates a delay.
- $\mathbf{W}_{x,z}$, $\mathbf{W}_{z,y}$ and $\mathbf{W}_{z,z}$ denote the weight matrices.

Assume that $\mathbf{x}$ and $\mathbf{y}$ are observed at each time step indexed by t,

$$\mathbf{z}_t = f_{\mathbf{W}}(\mathbf{z}_{t-1}, \mathbf{x}_t) = \mathbf{g}_z(\mathbf{W}_{z,z}\mathbf{z}_{t-1} + \mathbf{W}_{x,z}\mathbf{x}_t) \tag{6.1}$$

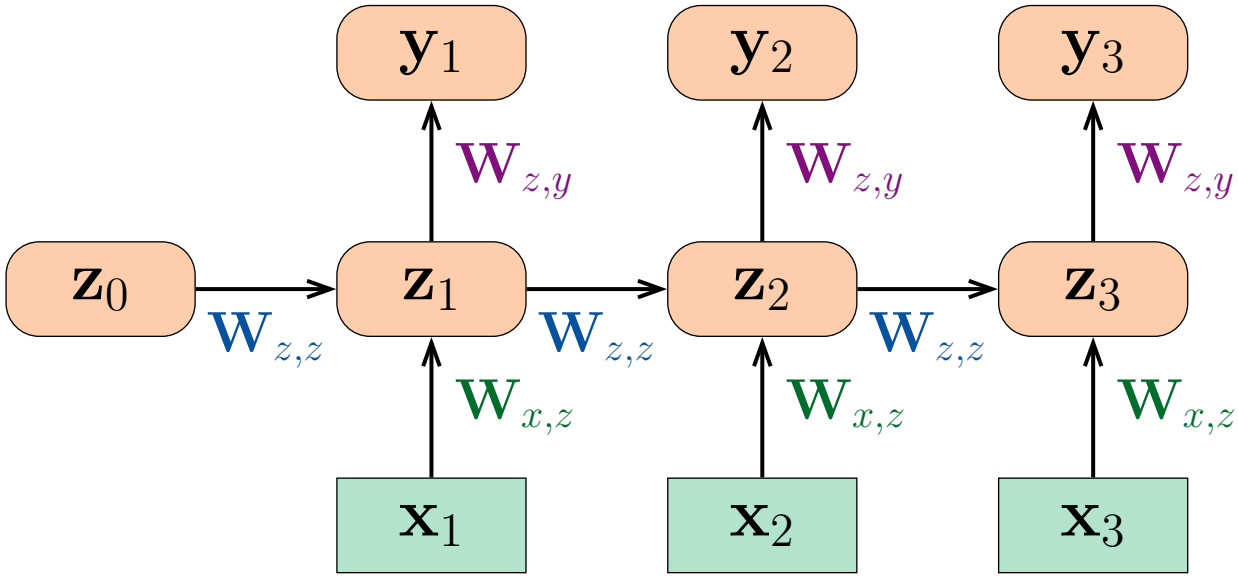
$$\hat{\mathbf{y}}_t = \mathbf{g}_y(\mathbf{W}_{z,y}\mathbf{z}_t) \tag{6.2}$$

From the equations,

- The hidden state at time t or $\mathbf{z}_t$ is computed from the current input $\mathbf{x}_t$ and the previous hidden state $\mathbf{z}_{t-1}$.
- The output at time t or $\hat{\mathbf{y}}_t$ is computed from the current hidden state $\mathbf{z}_t$.

RNNs make a **Markov assumption** i.e. the current state depends on a *finite fixed number* of previous states.

The unrolled version of the network can be displayed as:



Example 6.1. Given an RNN with one input unit, one hidden unit, and one output unit, the hidden state and output can be calculated by

$$z_t = g_z(w_{z,z}z_{t-1} + w_{x,z}x_t + w_{0,z}) \qquad w_{z,z} = 0.5, \quad w_{x,z} = 0.1, \; w_{0,z} = -1$$
$$\hat{y}_t = g_y(w_{z,y}z_t + w_{0,y}) \qquad w_{z,y} = -0.5, \; w_{0,y} = 0.3, \; z_0 = 0$$

where both g_z is sigmoid function and g_y is linear function. When we feed an input sequence $\langle 1, 4, 3 \rangle$ into the network, compute the output.

$$z_1 = \sigma((0.5)(z_0) + (0.1)(x_1) - 1) = \sigma((0.5)(0) + (0.1)(1) - 1) = \sigma(-0.9) = 0.2891$$
$$\hat{y}_1 = (-0.5)(z_1) + 0.3 = (-0.5)(0.2891) + 0.3 = 0.1554$$

Backpropagation through time

To train an RNN, the gradient descent algorithm is used. Assume that a squared-error loss summed over the time steps is used, $L = \sum_{t=1}^{T}(y_t - \hat{y}_t)^2$

From the RNN in Example 6.1, we can compute the gradient as

$$\begin{aligned}
\frac{\partial L}{\partial w_{z,z}} &= \frac{\partial}{\partial w_{z,z}} \sum_{t=1}^{T}(y_t - \hat{y}_t)^2 = \sum_{t=1}^{T} -2(y_t - \hat{y}_t)\frac{\partial \hat{y}_t}{\partial w_{z,z}} \\
&= \sum_{t=1}^{T} -2(y_t - \hat{y}_t)\frac{\partial}{\partial w_{z,z}}\Big[g_y(w_{z,y}z_t + w_{0,y})\Big] \\
&= \sum_{t=1}^{T} -2(y_t - \hat{y}_t)g'_{z,z}(w_{z,y}z_t + w_{0,y})\frac{\partial}{\partial w_{z,z}}\Big[w_{z,y}z_t + w_{0,y}\Big] \\
&= \sum_{t=1}^{T} -2(y_t - \hat{y}_t)g'_{z,z}(w_{z,y}z_t + w_{0,y})w_{z,y}\frac{\partial z_t}{\partial w_{z,z}}
\end{aligned}$$

$$
\begin{aligned}
\frac{\partial z_t}{\partial w_{z,z}} &= \frac{\partial}{\partial w_{z,z}}\Big[g_z(w_{z,z}z_{t-1} + w_{x,z}x_t + w_{0,z})\Big] \\
&= g_z'(w_{z,z}z_{t-1} + w_{x,z}x_t + w_{0,z})\frac{\partial}{\partial w_{z,z}}\Big[w_{z,z}z_{t-1} + w_{x,z}x_t + w_{0,z}\Big] \\
&= g_z'(w_{z,z}z_{t-1} + w_{x,z}x_t + w_{0,z})\Big[\frac{\partial}{\partial w_{z,z}}\big(w_{z,z}z_{t-1}\big) + \frac{\partial}{\partial w_{z,z}}\big(w_{x,z}x_t\big) + \frac{\partial w_{0,z}}{\partial w_{z,z}}\Big] \\
&= g_z'(w_{z,z}z_{t-1} + w_{x,z}x_t + w_{0,z})\Big[z_{t-1}\frac{\partial w_{z,z}}{\partial w_{z,z}} + w_{z,z}\frac{\partial z_{t-1}}{\partial w_{z,z}}\Big] \\
&= g_z'(w_{z,z}z_{t-1} + w_{x,z}x_t + w_{0,z})\Big[z_{t-1} + w_{z,z}\frac{\partial z_{t-1}}{\partial w_{z,z}}\Big]
\end{aligned}
$$

The gradient from time step t is calculated using the gradient from time step $t-1$. The gradient calculation requires the recursive calculation.

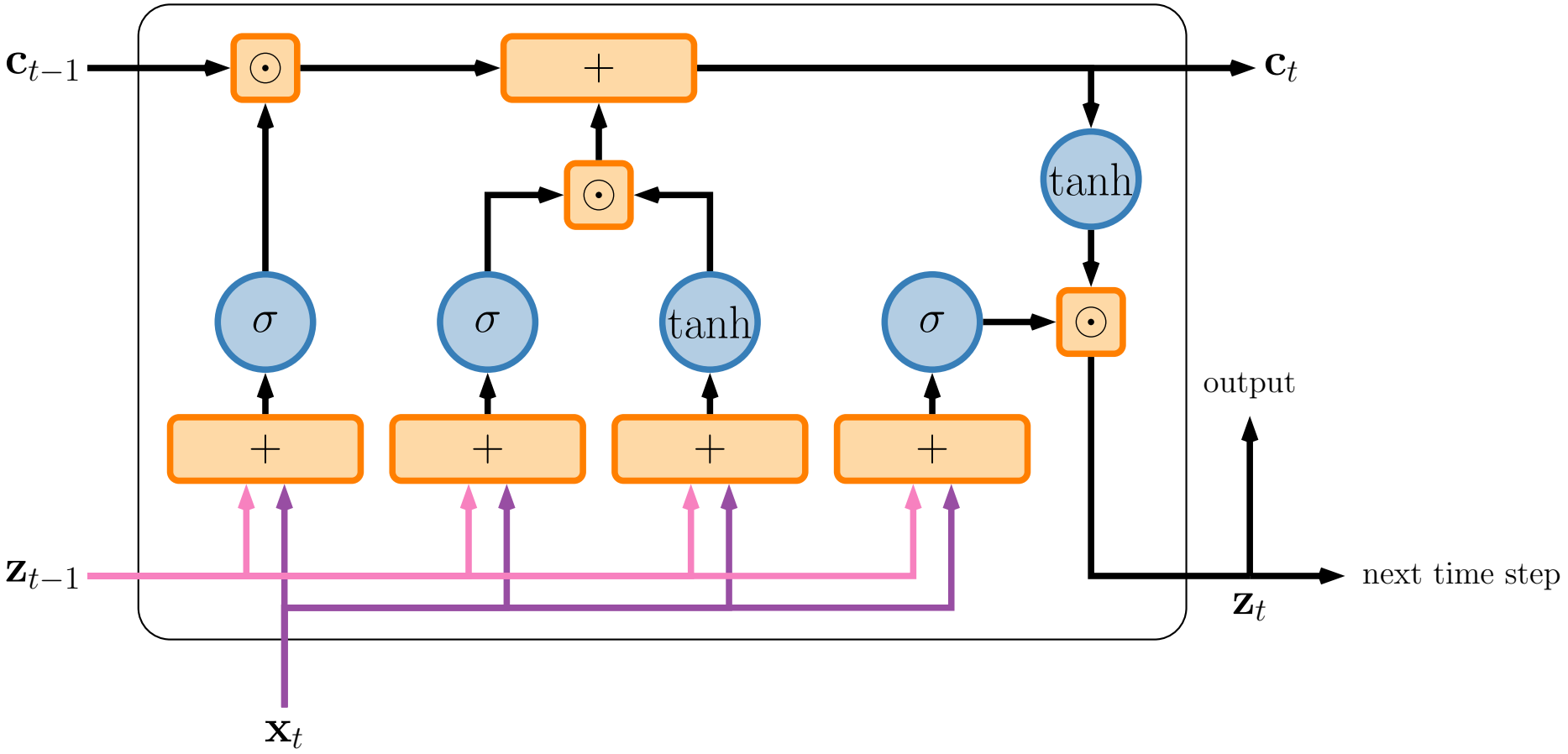
The gradients at T will include terms proportional to $w_{z,z} \prod_{t=1}^{T} g_z'(w_{z,z} z_{t-1} + w_{z,x} x_t + w_{0,z})$

- If $w_{z,z} < 1$, the RNN may suffer from the *vanishing gradient* problem.
- If $w_{z,z} > 1$, the RNN may suffer from the *exploding gradient* problem.

especially when T is large.

6.2 Long Short-Term Memory (LSTM)

The LSTM is an RNN architecture designed to overcome the vanishing gradient problem by preserving information over many time steps.



LSTM introduces a component called the **memory cell** or **cell state** denoted by $\mathbf{c}$. It is a piece of information copied from time step to time step.
The memory cell is updated wihtout being directly multiplied by any weight. New information enters the memory cells by *adding* updates.
LSTM uses several **gates** to control the flow of information. The value of each gating unit is in the range $[0, 1]$.

- The **forget gate** controls if each element of the memory cell is copied to the next time step (remembered) or reset to 0 (forgotten).

$$\mathbf{f}_t = \sigma(W_{x,f}\mathbf{x}_t + W_{z,f}\mathbf{z}_{t-1})$$

- The **input gate** controls if each element of the memory cell is updated additively by the **candiate value** from the input at the current time step.

$$\mathbf{i}_t = \sigma(W_{x,i}\mathbf{x}_t + W_{z,i}\mathbf{z}_{t-1})$$
$$\tilde{\mathbf{c}}_t = \tanh(W_{x,c}\mathbf{x}_t + W_{z,c}\mathbf{z}_{t-1})$$

- The **output gate** controls if each element of the memory cell is transferred to the hidden state.

$$\mathbf{o}_t = \sigma(W_{x,o}\mathbf{x}_t + W_{z,o}\mathbf{z}_{t-1})$$

Finally, the memory cell and the hidden state (also, the output) are calculated from

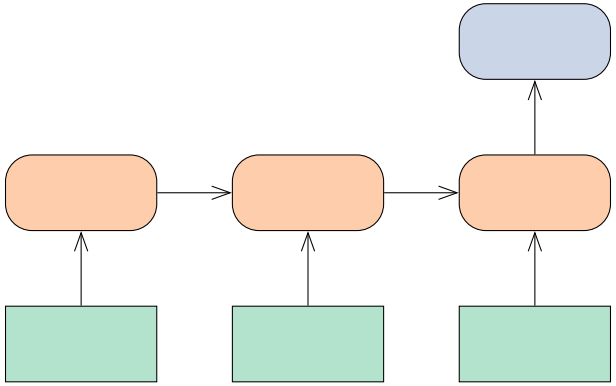
$$\mathbf{c}_t = (\mathbf{c}_{t-1} \odot \mathbf{f}_t) + (\tilde{\mathbf{c}}_t \odot \mathbf{i}_t)$$
$$\mathbf{z}_t = \tanh(\mathbf{c}_t) \odot \mathbf{o}_t$$

where $\odot$ is the elementwise multiplication operator.

6.3 Categories of Sequence Modeling

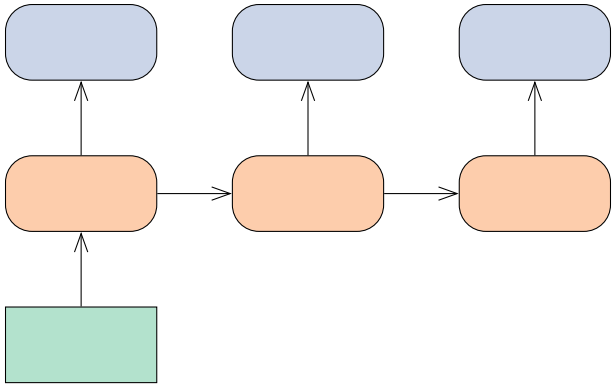
Many-to-one

- Input: a sequence of arrays i.e. $T_x > 1$.
- Output: a fixed-size vector or a scalar value i.e. $T_y = 1$.
- Example: sentiment classification, video classification, gait recognition



One-to-many

- Input: a fixed-size array i.e. $T_x = 1$.
- Output: a sequence of arrays i.e. $T_y > 1$.
- Example: image captioning, music generation

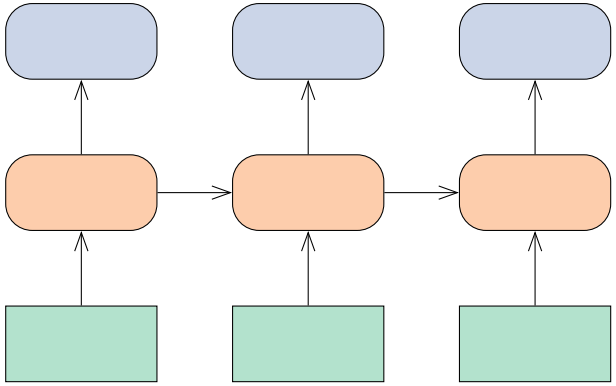


Many-to-many

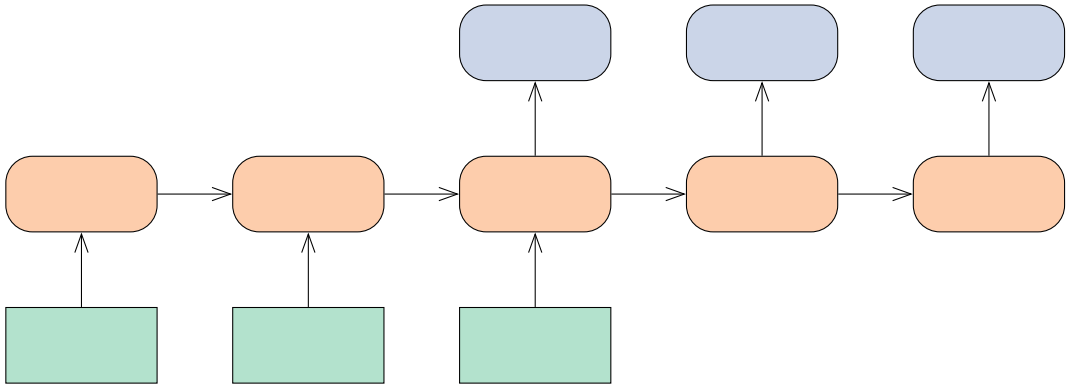
- Input: a sequence of arrays i.e. $T_x > 1$.
- Output: a sequence of arrays i.e. $T_y > 1$.

This category can be divided into two subcategories:

- When $T_x = T_y$: part-of-speech tagging, name-entity recognition

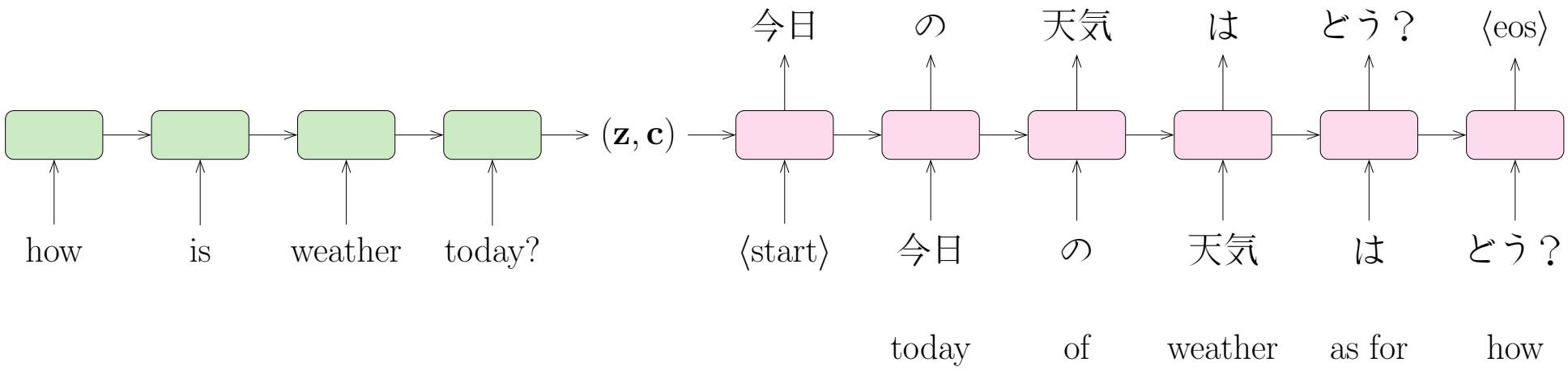


- When $T_x \neq T_y$: machine translation, cursive handwriting recognition

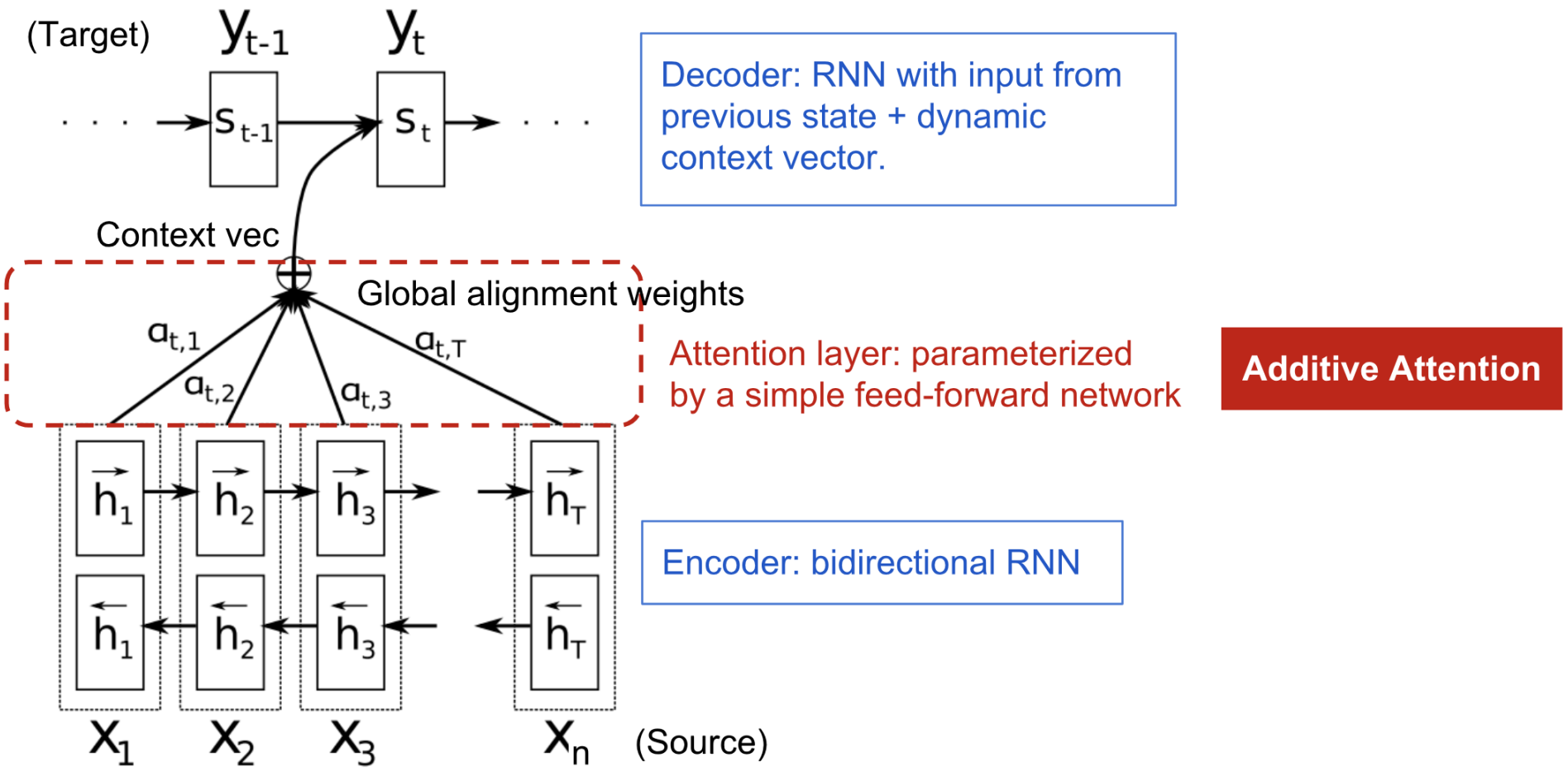


6.4 Seq2seq Model and Attention Mechanism

Sequence-to-sequence model is an approach designed to cope with Natual Language Processing (NLP) applications, e.g. machine translation, image captioning, etc.

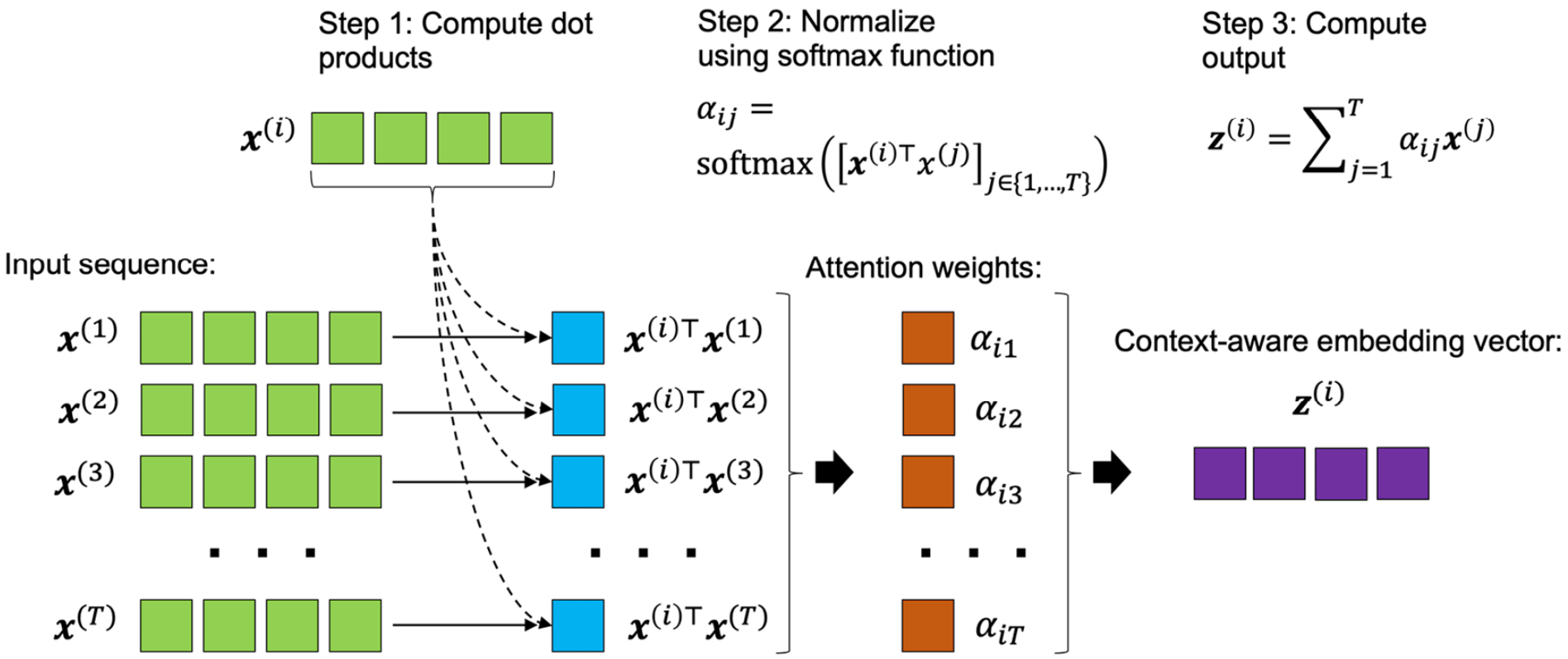


Attention Mechanism



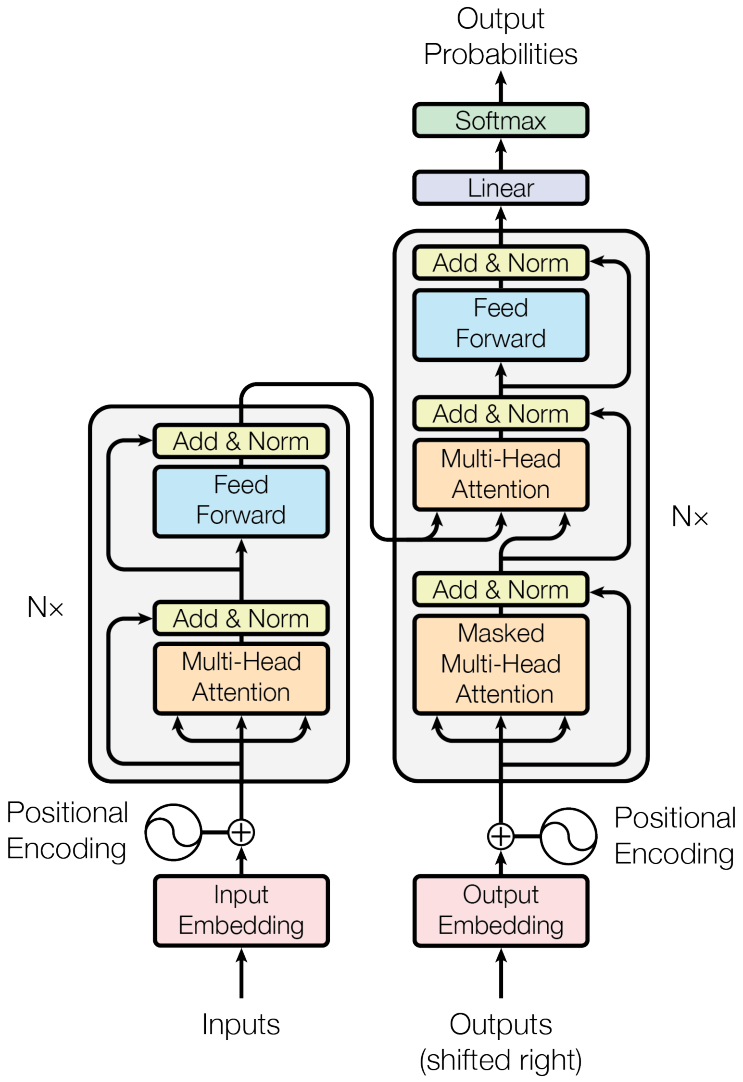
https://lilianweng.github.io/lil-log/assets/images/encoder-decoder-attention.png

Self-Attention



Machine Learning with PyTorch and Scikit-Learn by Raschka et al.

6.5 Transformer: Attention Is All You Need



$$(x_1, x_2, \ldots, x_n)$$
$$\downarrow$$
$$(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$$
$$\downarrow$$
$$(\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n)$$
$$\downarrow$$
$$(y_1, y_2, \ldots, y_m)$$

6.5.1 Positional encoding

Since the transformer is not a recurrent neural network, all calculations are conducted in parallel. Position information needs to be included in each input vector.

$$\begin{array}{ccccccc} \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3 & \dots & \mathbf{x}_n \\ \downarrow & \downarrow & \downarrow & \dots & \downarrow \\ 0 & 1 & 2 & \dots & n-1 \end{array}$$

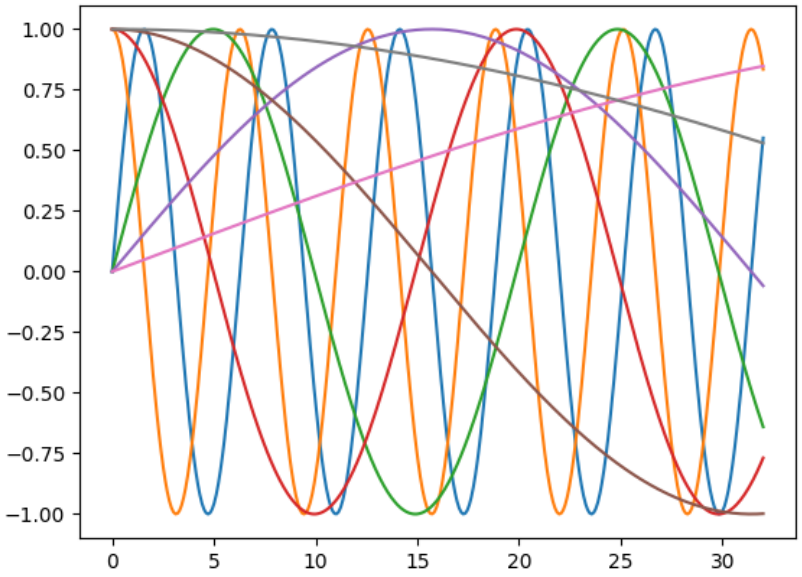
The problem of this encoding is the scale of numbers. This causes problems like exploding gradients.

$$\begin{array}{cccccccc} \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3 & \mathbf{x}_4 & \mathbf{x}_5 & \mathbf{x}_6 & \mathbf{x}_7 & \mathbf{x}_8 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}$$

Sine and cosine functions of different frequencies are used for the positional encoding:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right) \tag{6.3}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right) \tag{6.4}$$



6.5.2 Self attention

Given a sequence of vectors, $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}$ where each $\mathbf{x}^{(i)} \in \mathbb{R}^d$, find a sequence of context-aware embedding vectors, $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(n)}$ based on similarities between input vectors.

$$X_{n\times d} = \begin{bmatrix} \cdots\cdots \mathbf{x}^{(1)\top} \cdots\cdots \\ \cdots\cdots \mathbf{x}^{(2)\top} \cdots\cdots \\ \vdots \\ \cdots\cdots \mathbf{x}^{(n)\top} \cdots\cdots \end{bmatrix} \quad \rightarrow \quad Z_{n\times d} = \begin{bmatrix} \cdots\cdots \mathbf{z}^{(1)\top} \cdots\cdots \\ \cdots\cdots \mathbf{z}^{(2)\top} \cdots\cdots \\ \vdots \\ \cdots\cdots \mathbf{z}^{(n)\top} \cdots\cdots \end{bmatrix}$$

For each $\mathbf{x}^{(i)}$,

$$\mathbf{z}^{(i)} = \sum_{j=1}^{n} a_{ij}\mathbf{x}^{(j)} \tag{6.5}$$

where

$$a_{ij} = \text{softmax}\left(\left[\omega_{ij}\right]_{j=1,\ldots,n}\right) \tag{6.6}$$

$$= \frac{\exp(\omega_{ij})}{\sum_{j=1}^{T} \exp(\omega_{ij})} \tag{6.7}$$

$$\omega_{ij} = \mathbf{x}^{(i)\top}\mathbf{x}^{(j)} \tag{6.8}$$

6.5.3 Scaled dot-product attention

Introduce three parameter matrices: $W^Q \in \mathbb{R}^{d \times d_k}$, $W^K \in \mathbb{R}^{d \times d_k}$ and $W^V \in \mathbb{R}^{d \times d_v}$.

$$Q = XW^Q \tag{6.9}$$

$$K = XW^K \tag{6.10}$$

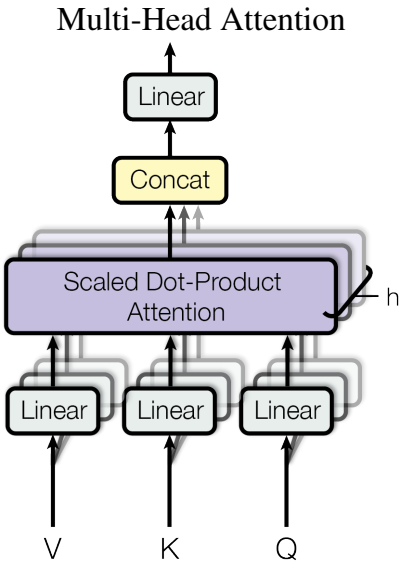
$$V = XW^V \tag{6.11}$$

Scaled dot-product attention can be computed from

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V \tag{6.12}$$

6.5.4 Multi-Head Attention

Multi-head attention is a concatenation of h attentions.

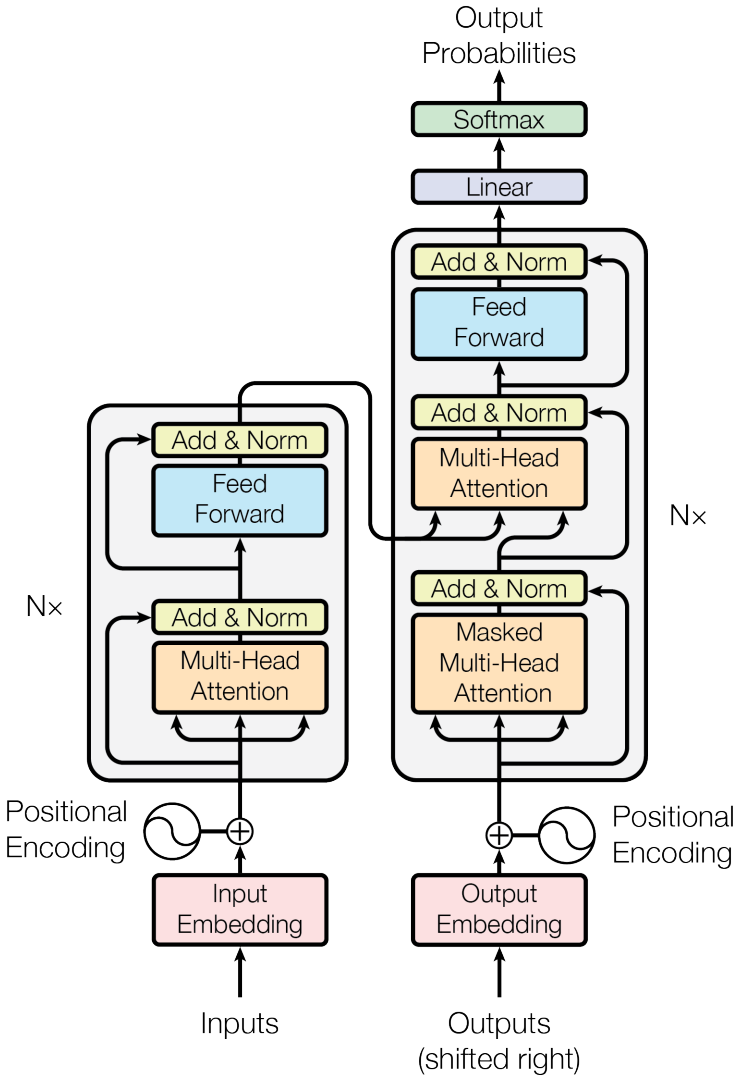


6.5.5 Position-wise Feed-Forward Network

An MLP is applied to each vector $\mathbf{z}_i$ separately,

$$\text{FFN}(\mathbf{z}) = \text{ReLU}(\mathbf{z}W_1 + b_1)W_2 + b_2 \tag{6.13}$$

6.5.6 Classification



Further Study

- Transformers from Scratch,

  `https://peterbloem.nl/blog/transformers`

- Summary by Dmitry Kobak,

  `https://twitter.com/hippopedoid/status/1641432291149848576`