

MATHEMATICS FOR MACHINE LEARNING

Special Mathematics Lecture, Nagoya University (Fall 2023)

Lecturer: Henrik Bachmann (Math. Building Room 457, henrik.bachmann@math.nagoya-u.ac.jp)

Teaching Assistant: Risan

Course information are available at: <https://www.henrikbachmann.com/mml2023.html>

If you are a student of this course, you are welcome to help us out!

🔔 These notes are under construction and therefore may contain mistakes and change without notice. If you find any typos/errors or have any suggestion, please contact us.

Contents

1	Introduction	2
2	Basics	3
2.1	Python	3
2.2	Tic-Tac-Toe & Minimax	3
2.3	Recall some Linear Algebra	7
3	Basics of supervised learning	9
3.1	Linear regression	9
3.2	Logistic regression	17
3.3	Naive Bayes	21
3.4	Gaussian Discriminant Analysis	26
4	Neural Networks	28
4.1	Multi-layer fully connected feedforward neural networks	28
4.2	Training the neural network: Backpropagation	30
4.3	Convolutional neural networks	34
5	Reinforcement Learning	35
	References	39

1 Introduction

Machine learning is commonly defined as the study of computer algorithms that improve automatically through experience. This is in contrast with traditional programming, in which the programmer implements an explicit algorithm to solve the problem.

Machine learning is used in a myriad of fields in the modern world, such as (but not limited to) AI in computer games, image processing, speech recognition, translation, weather forecast and email spam filtering. The goal of this course is to get an overview of the algorithms used in machine learning, understand the mathematics behind these algorithms, and learn how to implement them in Python.

Machine learning is typically divided into three big paradigms:

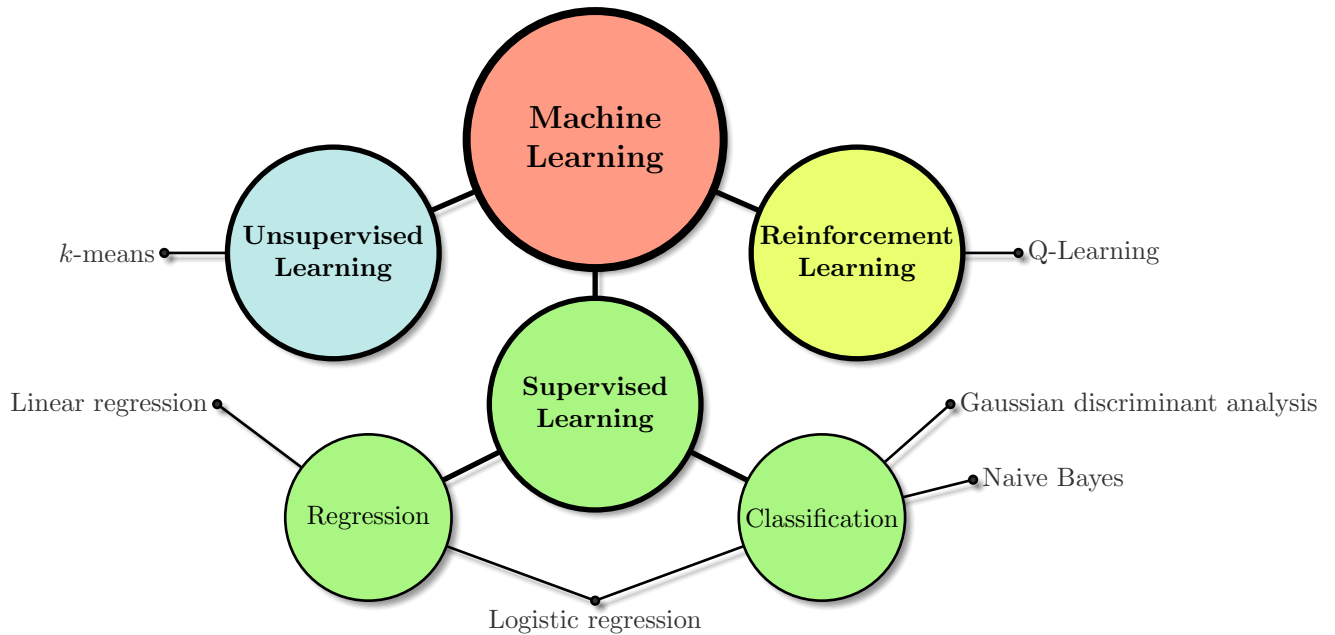


Figure 1: An illustration of the branches of Machine Learning.

- 1) **Supervised learning:** Supervised learning is the branch of machine learning where labelled input-output data (training data) is provided to the machine. The training data is a pair consisting of an input (a vector) and a desired output value (also called the supervisory signal). The machine's task is to analyze the training data and infer a function from said training data to be able to map new examples. Supervised learning is further divided based on the type of output: **classification** if the output is discrete (e.g. categorization) and **regression** if the output is continuous.
- 2) **Unsupervised learning:** Unsupervised learning is the branch of machine learning that deals with machines that take in untagged data and discerns patterns from said untagged data. By doing this, one hopes that the machine learns by mimicry, which leads it to build a representation of its world and then generate imaginative content from it. In contrast to supervised learning, where the training data has been tagged beforehand, unsupervised learning forces the machine to self-

organize the data and represent patterns as probability densities or a combination of neural feature preferences.

- 3) **Reinforcement Learning:** Reinforcement learning is an area of machine learning concerned with the programming of intelligent machines such that they take the optimal actions in an environment to maximize cumulative reward. In contrast to supervised learning, reinforcement learning needs neither labelled input/output pairs nor explicit correction of sub-optimal actions. Instead, the focus is finding a balance between exploration (of the unknown) and exploitation (of current knowledge).

In this course, we will consider at least one algorithm for each of these areas. Of course, we will not be able to cover everything in detail and student should see this course just as an entry guide to the big world of machine learning.

2 Basics

In this section, we recall some basic in Python and Linear Algebra.

2.1 Python

In this course, we will do several examples in Python using Google Colab. An overview (or review) of some basic Python commands is given in [this Colab notebook](#) (courtesy of Risan).

2.2 Tic-Tac-Toe & Minimax

In this section, we will discuss an implementation of the game Tic-Tac-Toe using Python, alongside an algorithm that can evaluate the state of a game and find the best possible move given that state.

It is important to note that the code given below is simply one example. There are other ways to implement these functions and algorithms.

Tic-Tac-Toe is a well-known game where players (on alternating turns) put down marks X and O on a spot in the board (in this case, a 3×3 grid). The winner is the player who gets three of their marks in a row, column, or diagonal of the grid. We let always the player X start.

Minimax is an algorithm that can be used to look several steps ahead in perfect zero-sum games (games where if a player win, other loses, hence "zero-sum"), such as Chess or Tic-Tac-Toe. The goal of the algorithm is to assign a value ("evaluation") to a game state. The goal of one of the players is to maximize this evaluation, while the goal of the other player is to minimize this evaluation. Knowing this, each player looks ahead several steps ahead, and determines which step the opponent will take, and by considering that, takes the best possible move in any given game state.

An example of this algorithm applied to Tic-Tac-Toe is illustrated in Figure 2. We start on the turn of player X (maximizing), and then it alternates between X (maximizing player) and O (minimizing player). In each row, the maximum value (for player X) or the minimum value (for player O) is chosen.

2.2.1 Helping Functions

Before we discuss the implementation of the algorithm in Tic-Tac-Toe, it is useful to first describe how one can implement Tic-Tac-Toe in Python.

We can represent the board of Tic-Tac-Toe in Python by using a nested list. For example,

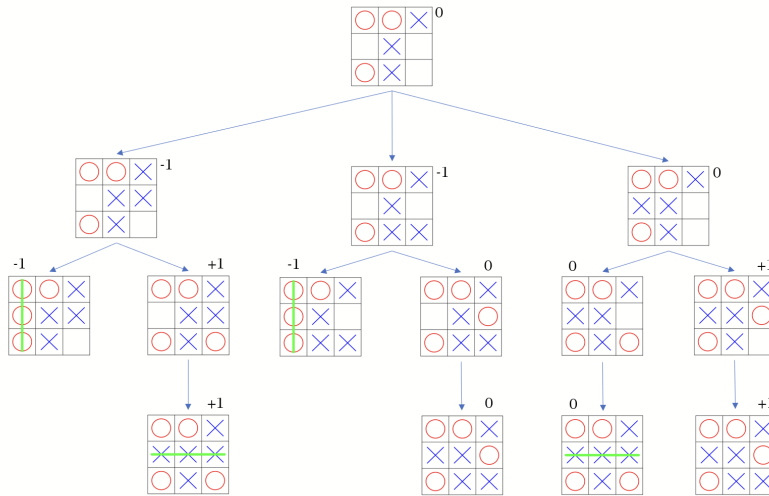


Figure 2: Illustration of Minimax algorithm in Tic-Tac-Toe

```

1 #Example Board
2 grid = [
3     [" ", " ", "0"],
4     [" ", "X", "X"],
5     [" ", " ", " "]]
6 #In reality, the line breaks in this list means nothing.
7 #This is only to represent the different rows of the grid.
    
```

From here, we define some helping functions (three, to be exact) to help in writing our implementation of Minimax in Python.

The first function takes a grid as the parameter and determines whether or not a game is over and if there is a winner. An example implementation is given below:

```

1 def get_winner(grid):
2     # Examine the rows:
3     for row in grid:
4         if (row[0] == row[1] == row[2]) and (row[0] != " "):
5             return row[0]
6
7     # Examine the columns:
8     for c in range(3):
9         if (grid[0][c] == grid[1][c] == grid[2][c]) and (grid[0][c] != " "):
10            return grid[0][c]
11
12    # Examine the diagonals (d = -1 : leading diagonal, while d = 1 : antidiagonal):
13    for d in (-1,1):
14        if (grid[0][1+d] == grid[1][1] == grid[2][1-d]) and (grid[1][1] != " "):
15            return grid[1][1]
16
17    # If there is no winner:
18    return None
    
```

The second function determines the next player given a board state. An example is given below:

```

1 # Make a function that counts the number of 0 or X in a given grid
    
```

```

2 def counter(grid, player):
3     cnt = 0
4     #We iterate over each row in the grid.
5     for row in grid:
6         #This function counts the number of times "X" or "O" appears in one given row
7         cnt += row.count(player)
8     return cnt
9
10 def next_player(grid):
11     # First we check if the game is over (full board or player win):
12     if counter(grid,"X") + counter(grid,"O") == 9 or (get_winner(grid) != None):
13         return None
14     # If game is not over we check if there are less X than O. Since X goes first, if
15     # the number of marks are equal, it is player X's turn.
16     elif (counter(grid,"X") <= counter(grid, "O")):
17         return "X"
18     # If game is not over, and it is not player X's turn, it is Player O's turn.
19     return "O"

```

The last function updates the grid when a move is made. We assume that all moves are valid, and no illegal moves can be made (moves are only made on empty spaces)

```

1 import copy
2 def make_move(grid,move):
3     #We make a copy of the grid, and then apply the move.
4     moved_grid = copy.deepcopy(grid)
5     moved_grid[move[0]][move[1]] = next_player(grid)
6     return moved_grid
7     #next_player(grid) automatically gives the next player's mark

```

Here, it is very important to use deepcopy instead of copy, since otherwise the original "grid" variable will be affected by each move (which might be undesirable in some cases).

2.2.2 Implementation of Minimax

Now, we want to implement the Minimax algorithm for Tic-Tac-Toe in Python. First, we need a code that can assign a value to a given board. An example is given below:

```

1 def get_possible_moves(grid):
2     possible_moves = []
3     for i in range(3):
4         for j in range(3):
5             if (grid[i][j] == " "):
6                 possible_moves.append((i,j))
7     return possible_moves
8
9 def minimax(grid, is_maximizing):
10     if (get_winner(grid) == next_player(grid) == None):
11         return 0
12     elif (get_winner(grid) == "X"):
13         return 1
14     elif (get_winner(grid) == "O"):
15         return -1
16
17     scores = []
18
19     for move in get_possible_moves(grid):
20         moved_grid = make_move(grid, move)
21         scores.append(minimax(moved_grid, not is_maximizing))

```

```

22
23     if (is_maximizing == True):
24         return max(scores)
25     else:
26         return min(scores)

```

The function "get_possible_moves" gives all possible moves for a given grid (the spaces which are completely empty). The algorithm takes two parameters, the grid and "is_maximizing". We assign the "X" player to be the maximizing player, so we set "is_maximizing == True" for "X" and "False" for "O".

The minimax function scans all possible game boards after one move from the player (lines 19-21) and runs the same minimax function on that new state, but for the opposite player (hence the "not is_maximizing" in line 21). It then appends these values to the list called "scores". The function then returns the maximum value of the elements in the list (if player is X) or the minimum value of the elements (if player is O).

Next, we implement a function to find the best possible move. Here, we define the 'best possible move(s)' as the move(s) that maximizes (or minimizes) the value of the minimax function. One implementation of this function is given below:

```

1 def find_move(grid):
2     if (next_player(grid) == None):
3         return None
4
5     possible_moves = get_possible_moves(grid)
6
7     if (next_player(grid) == "X"):
8         evaluation = -1
9         for move in possible_moves:
10            moved_grid = make_move(grid, move)
11            new_evaluation = minimax(moved_grid, False)
12            if (new_evaluation >= evaluation):
13                evaluation = new_evaluation
14                best_move = move
15        return best_move
16    elif (next_player(grid) == "O"):
17        evaluation = 1
18        for move in possible_moves:
19            moved_grid = make_move(grid, move)
20            new_evaluation = minimax(moved_grid, True)
21            if (new_evaluation <= evaluation):
22                evaluation = new_evaluation
23                best_move = move
24    return best_move

```

For each player, this function considers all possible moves, evaluates the new grid by the minimax function (for the opposite player). If a more favorable evaluation is obtained, the evaluation is then updated, and then the move is determined to be the best move.

2.2.3 Displaying the Grid

The Tic-Tac-Toe grid can be drawn in various representations, ranging from ASCII art (basic) to nice graphics in matplotlib. The implementation below is the code used to draw the grids in Figure 2 (though not the green lines there).

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 col3 = [0.0, 1.0, 0.0]
5 def draw_grid(grid):
6     plt.figure(figsize=(7, 7)) #Makes the figure
7
8     for a in range(4):
9         plt.plot([a,a], [0,-3], color = "black") #Draws the horizontal lines in the grid
10        plt.plot([0,3], [-a,-a], color = "black") #Draws the vertical lines in the grid
11        plt.axis('off')
12
13    for i in range(3):
14        for j in range(3):
15            if (grid[i][j] == "X"): #Draws X in the appropriate slots
16                plt.plot([j+0.2, j+0.8], [-i-0.2, -i-0.8], color="blue", linewidth= 3.0)
17                plt.plot([j+0.2, j+0.8], [-i-0.8, -i-0.2], color="blue", linewidth= 3.0)
18            if (grid[i][j] == "0"): #Draws 0 in the appropriate slots
19                circle1=plt.Circle((j+0.5, -i-0.5), 0.35, color='red', linewidth=3.0, fill=
                False)
20                plt.gca().add_patch(circle1)
21        plt.show()
22    return None
    
```

2.3 Recall some Linear Algebra

In this section, we recall some notations and concepts from Linear Algebra which we will use in this course.

Definition 2.1. (i) A $m \times n$ -matrix is given by an array (m rows, n columns) of numbers $a_{ij} \in \mathbb{R}$

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} = (a_{ij})_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}}.$$

Notation: We often just write $A = (a_{ij})$ if the size of A , i.e. m and n , are known from context. By $\mathbb{R}^{m \times n}$ we denote the set all of all $m \times n$ -matrices.

(ii) A (column-) **vector** of size n is a $n \times 1$ -matrix

$$v = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \tag{2.1}$$

and the set of all vectors of size n is denoted by $\mathbb{R}^n = \mathbb{R}^{n \times 1}$.

Definition 2.2. For matrices $A = (a_{ij}), B = (b_{ij}) \in \mathbb{R}^{m \times n}$ and a real number $\lambda \in \mathbb{R}$ we define

$$\begin{aligned} A + B &= (a_{ij} + b_{ij}) \in \mathbb{R}^{m \times n} && \text{(Sum of two matrices),} \\ \lambda A &= (\lambda a_{ij}) \in \mathbb{R}^{m \times n} && \text{(Scalar multiplication).} \end{aligned}$$

In the case $\lambda = -1$ we write $(-1)A = -A$ and $A - B$ means $A + (-1)B$.

The matrices A and B need to be of the same size, otherwise the sum $A + B$ is not defined. A special case of the addition of matrices is given by the addition of vectors. For $u, v \in \mathbb{R}^n$ and $\lambda \in \mathbb{R}$ we have

$$u = \begin{pmatrix} u_1 \\ \vdots \\ u_n \end{pmatrix}, v = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}, \quad u + v = \begin{pmatrix} u_1 + v_1 \\ \vdots \\ u_n + v_n \end{pmatrix}, \quad \lambda v = \begin{pmatrix} \lambda v_1 \\ \vdots \\ \lambda v_n \end{pmatrix}.$$

Definition 2.3. The product of a matrix $A = (a_{ij}) \in \mathbb{R}^{m \times n}$ and a vector $v \in \mathbb{R}^n$ is defined by

$$Av = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix} \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} a_{11}v_1 + a_{12}v_2 + \dots + a_{1n}v_n \\ a_{21}v_1 + a_{22}v_2 + \dots + a_{2n}v_n \\ \vdots \\ a_{m1}v_1 + a_{m2}v_2 + \dots + a_{mn}v_n \end{pmatrix} \in \mathbb{R}^m.$$

We have: $(m \times n\text{-matrix}) \cdot (\text{vector of size } n) = (\text{vector of size } m)$.

Proposition 2.4. For any $A \in \mathbb{R}^{m \times n}$, $x, y \in \mathbb{R}^n$ and $\lambda \in \mathbb{R}$,

(i) $A(x + y) = Ax + Ay$,

(ii) $A(\lambda x) = \lambda(Ax)$.

Definition 2.5. The **transpose** of a matrix $A = (a_{ij}) \in \mathbb{R}^{m \times n}$ is the matrix $A^T = (a_{ji}) \in \mathbb{R}^{n \times m}$.

Most of the time we will consider column vectors v as in (2.1). To save (vertical) space we often write in the text $v = (v_1, \dots, v_n)^T \in \mathbb{R}^n$.

Definition 2.6. The **dot-product** of two vectors $u, v \in \mathbb{R}^n$ is defined by

$$u \bullet v = u^T v = u_1 v_1 + \dots + u_n v_n.$$

Definition 2.7. A **subspace** of \mathbb{R}^n is a subset $U \subseteq \mathbb{R}^n$ that satisfies:

(i) The zero vector (identity element) is a member of U ,

(ii) If $u, v \in U$, then $u + v \in U$ (closure under addition),

(iii) For all $\lambda \in \mathbb{R}$, if $u \in U$ then $\lambda u \in U$.

Definition 2.8. For a subspace $U \subset \mathbb{R}^n$, the **orthogonal complement** of U is defined by

$$U^\perp = \{v \in \mathbb{R}^n \mid u \bullet v = 0, \forall u \in U\}.$$

Proposition 2.9. Let $U \subset \mathbb{R}^n$ be a subspace. Any $y \in \mathbb{R}^n$ can be written uniquely as

$$y = y_U + y_\perp,$$

with $y_U \in U$ and $y_\perp \in U^\perp$.

Definition 2.10. Let $U \subset \mathbb{R}^n$ be a subspace. We define the **orthogonal projection** onto U to be the map

$$P_U : \mathbb{R}^n \longrightarrow \mathbb{R}^n \\ x \longmapsto x_U$$

Definition 2.11. Let $A \in \mathbb{R}^{m \times n}$ be a matrix.

(i) The **kernel** of A is defined by the set

$$\ker(A) = \{v \in \mathbb{R}^n \mid Av = \mathbf{0}\}.$$

(ii) The **image** of A is defined by the set

$$\text{im}(A) = \{y \in \mathbb{R}^m \mid Ax = y \text{ for some } x \in \mathbb{R}^n\}.$$

The linear system $Ax = y$ has a solution if and only if $y \in \text{im}(A)$.

Lemma 2.12. For $A \in \mathbb{R}^{n \times m}$ we have

$$(\text{im}(A))^\perp = \ker(A^T).$$

3 Basics of supervised learning

We will start by considering supervised learning, i.e. our algorithms will learn from some training data. For this we introduce the following notations.

Definition 3.1. Let \mathcal{X}, \mathcal{Y} be arbitrary sets. In the following we will use the following notation.

(i) Input values (Feature space): \mathcal{X} .

(ii) Output value (Label space): \mathcal{Y} .

(iii) Training example: $(x, y) \in \mathcal{X} \times \mathcal{Y}$.

(iv) Training set (with n training examples): $\mathcal{T} = ((x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)})) \in (\mathcal{X} \times \mathcal{Y})^n$.

(v) Hypothesis: A function $h : \mathcal{X} \rightarrow \mathcal{Y}$.

(vi) Learning algorithm: An algorithm to create a hypothesis h out of a training set \mathcal{T} .

3.1 Linear regression

One simple learning algorithm is given by linear regression. In this example, we have a dataset of $n = 6$ students containing the time they lived in Nagoya (the feature space \mathcal{X}) together with the number of Tebasaki they ate (label space \mathcal{Y}) during this time.

Weeks living in Nagoya	Tebasaki eaten
2	5
7	20
13	14
16	32
22	22
27	38

Python Example 3.2. If we want to plot above data in Python we can use the following code:

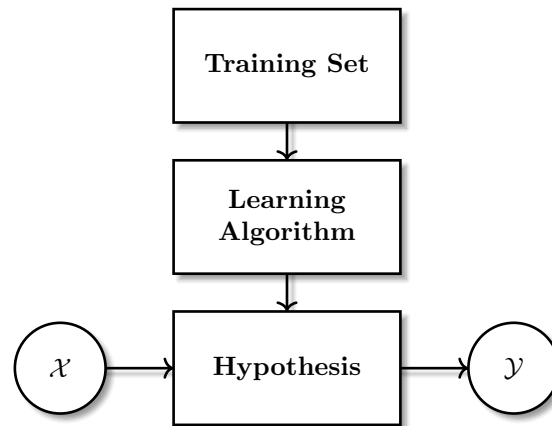


Figure 3: An illustration of the process of supervised learning.

```

1 import numpy as np # For doing math
2 from matplotlib import pyplot as plt # For plotting
3
4 # Training set
5 Tx = np.array([2, 7, 13, 16, 22, 27])
6 Ty = np.array([5, 20, 14, 32, 22, 38])
7
8 # Draw the Training set
9 plt.figure(figsize=(10,8))
10 plt.plot(Tx,Ty,'o',markersize=10)
11
12 # Give labels to the axis
13 plt.xlabel("Weeks living in Nagoya", fontsize=18)
14 plt.ylabel("Tebasaki eaten", fontsize=18)
15
16 #Show the graph
17 plt.show()
    
```

As a result we obtain the following graph:

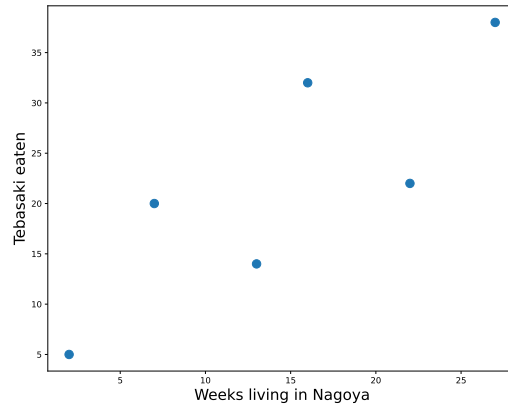


Figure 4: Plot of the data

Looking at the plot of the data, one might think that there exists some correlation between the time lived in Nagoya and the number of Tebasaki eaten. This correlation can be expressed as a hypothesis function $h : \mathcal{X} \rightarrow \mathcal{Y}$ that predicts the number of Tebasaki eaten given the time spent in Nagoya. Then, one might suggest that the hypothesis be a linear function defined by $h_\theta = \theta_0 + \theta_1 x$ for $\theta_0, \theta_1 \in \mathbb{R}$. We also write $\theta = (\theta_0, \theta_1)^T \in \mathbb{R}^2$. To add a graph of h_θ to Figure 4 one can add the following code to Python Example 3.2: `plt.plot(Tx, t0+ t1*Tx)` (Set $t0 = \theta_0$ and $t1 = \theta_1$ to desired values beforehand).

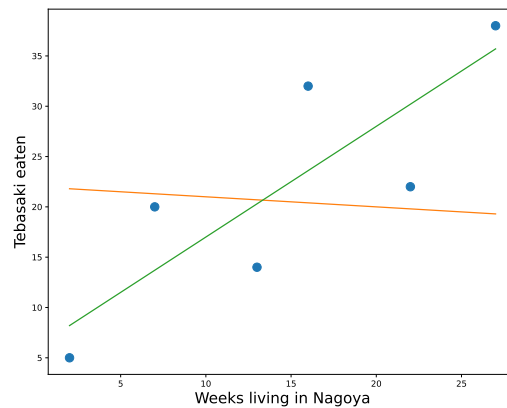


Figure 5: The plot of the data with the graphs of $h_{\theta'}$ and h_θ where $\theta' = (22, -0.1)^T$ and $\theta = (6, 1.1)^T$.

We will now generalize the idea of linear regression to d features. Set $x = (x_0, x_1, x_2, \dots, x_d)^T \in \mathbb{R}^{d+1}$ (Recall Definition 2.5). Let us set $x_0 = 1$, and $\mathcal{X} = \mathbb{R}^{d+1}$, and $\mathcal{Y} = \mathbb{R}$. Let us assume that the

hypothesis is in the form

$$h_\theta(x) := \theta_0 + \theta_1 x_1 + \cdots + \theta_d x_d = \sum_{i=0}^d \theta_i x_i = \theta^T x \quad (3.1)$$

with **parameters** $\theta = (\theta_0, \theta_1, \dots, \theta_d)^T \in \mathbb{R}^{d+1}$.

For a given training set $\mathcal{T} = ((x^{(1)}, y^{(1)}), \dots, (x^{(t)}, y^{(t)})) \in (\mathcal{X} \times \mathcal{Y})^n$, we want to determine the best choice for the parameter θ . To do this, we need a way to judge how appropriate a parameter θ is. To this end, we define the **cost function** to be a function $J : \mathcal{X} \rightarrow \mathcal{Y}$ that depends on the training set \mathcal{T} . We define the best θ to be the one which minimizes J . One example is the least-squares function

$$J(\theta) = \frac{1}{2} \sum_{j=1}^n (h_\theta(x^{(j)}) - y^{(j)})^2. \quad (3.2)$$

3.1.1 Linear regression with gradient descent

Continuing on from the previous section, our goal is to find $\theta \in \mathbb{R}^{d+1}$ that minimizes $J(\theta)$. To do this, we use a method called the **gradient descent method**.

The **gradient** of a multivariable scalar-valued function J , denoted by ∇J , is defined as the direction of greatest change of J . It is given by the expression

$$\nabla J = \begin{pmatrix} \frac{\partial}{\partial \theta_0} J \\ \frac{\partial}{\partial \theta_1} J \\ \vdots \\ \frac{\partial}{\partial \theta_d} J \end{pmatrix}$$

Algorithm 3.3 (Gradient Descent). *The gradient descent algorithm, as its name suggests, tries to find the minimum value of J by going against the gradient (the direction of steepest ascent). It is comprised of the following steps:*

(i) Start with a random starting value, e.g. $\theta = 0 = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$.

(ii) Change the parameters in the opposite direction of the gradient. To do this, we subtract the a vector proportional to the gradient from the current parameters. The proportionality constant is called the **learning rate**, $\alpha \in \mathbb{R}$, $\alpha > 0$. The new θ is therefore given by

$$\theta := \theta - \alpha \nabla J(\theta).$$

(iii) Repeat step ii) until $J(\theta)$ does not change a lot anymore.

Lemma 3.4. *The j^{th} component of the gradient of the cost function J at θ is given by*

$$\nabla J(\theta)_j = \sum_{j=0}^d (h_\theta(x^{(j)}) - y^{(j)}) x^{(j)}.$$

J is the cost function defined in Equation (3.2), and $h(\theta)$ has the form described in Equation (3.5).

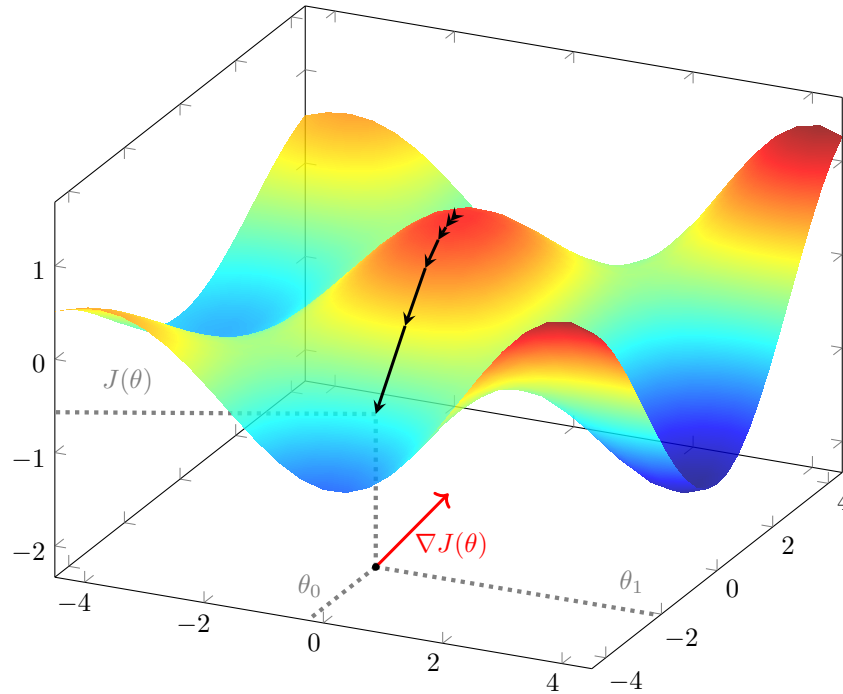


Figure 6: Visualization of the first few steps of the gradient descent.

Proof. We calculate $\nabla J(\theta)_j := \frac{d}{d\theta_j} J$ for $j = 0, \dots, d$.

$$\begin{aligned}
 \nabla J(\theta)_j &= \frac{\partial}{\partial \theta_j} J(\theta) = \sum_{j=0}^d \frac{1}{2} \frac{\partial}{\partial \theta_j} \left[\left(h_\theta(x^{(j)}) - y^{(j)} \right)^2 \right] \\
 &= \sum_{j=0}^d \frac{1}{2} \cdot 2 \left(h_\theta(x^{(j)}) - y^{(j)} \right) \cdot \frac{\partial}{\partial \theta_j} \left[h_\theta(x^{(j)}) - y^{(j)} \right] \\
 &= \sum_{j=0}^d \left(h_\theta(x^{(j)}) - y^{(j)} \right) \cdot \frac{\partial}{\partial \theta_j} \left[\sum_{i=0}^d \theta_i x^{(i)} \right] \\
 &= \sum_{j=0}^d \left(h_\theta(x^{(j)}) - y^{(j)} \right) x^{(j)}.
 \end{aligned}$$

□

By this result, step ii) in the gradient descent algorithm can be written as:

$$\theta := \theta + \alpha \sum_{j=1}^n (y^{(j)} - h_\theta(x^{(j)})) x^{(j)}$$

Similarly to how one would find the extrema of a function of one variable by finding the points where the derivative vanishes, we can do the same for multivariate functions (such as the cost function). This can be done explicitly, and in the case of linear regression, one obtains a unique solution (which is the solution of the normal equation). This analytical approach is described in detail in [N, Section 1.2].

3.1.2 Solving linear regression with linear algebra in the case $\mathcal{X} = \mathcal{Y} = \mathbb{R}$

We now present an alternative method of solving the problem of minimizing the cost function. This method is based in linear algebra. Let us assume that we have a training set $\mathcal{T} \in (\mathcal{X} \times \mathcal{Y})^n$ with $\mathcal{X} = \mathcal{Y} = \mathbb{R}$. If all training examples lie on a straight line, then there exists θ that satisfies

$$A\theta = \begin{pmatrix} 1 & x^{(1)} \\ 1 & x^{(2)} \\ \vdots & \vdots \\ 1 & x^{(n)} \end{pmatrix} \begin{pmatrix} \theta_0 \\ \theta_1 \end{pmatrix} = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{pmatrix} = y.$$

For a training set $\mathcal{T} = ((x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)}))$ we define

$$y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{pmatrix}.$$

However, in an overwhelming majority of cases, there exists no such line that passes through all training examples. In this case, the linear system $A\theta = y$ has no solutions. As such, we have to find the best alternative : the line that minimizes the norm (distance) between the line and all training examples.

We now consider the case of d features and an training set with n training examples. This means $\mathcal{T} = ((x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)}))$ with $x^{(j)} = (x_0^{(j)}, x_1^{(j)}, \dots, x_d^{(j)}) \in \mathbb{R}^{d+1}$ (where we always set $x_0^{(j)} = 1$) and $y^{(j)} \in \mathbb{R}$ for $j = 1, \dots, n$. In this case we define the matrix A by

$$A = \begin{pmatrix} | & & | \\ x^{(1)} & \dots & x^{(n)} \\ | & & | \end{pmatrix}^T = \begin{pmatrix} 1 & x_1^{(1)} & \dots & x_d^{(1)} \\ \vdots & \vdots & \dots & \vdots \\ 1 & x_1^{(n)} & \dots & x_d^{(n)} \end{pmatrix} \in \mathbb{R}^{n \times d+1}.$$

Then by the definition of the cost function J we see that we have

$$\|A\theta - y\| \text{ is minimal} \iff J(\theta) \text{ is minimal,}$$

since

$$A\theta - y = \begin{pmatrix} 1 & x_1^{(1)} & \dots & x_d^{(1)} \\ \vdots & \vdots & \dots & \vdots \\ 1 & x_1^{(n)} & \dots & x_d^{(n)} \end{pmatrix} \begin{pmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{pmatrix} - \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{pmatrix} = \begin{pmatrix} h_\theta(x^{(1)}) \\ \vdots \\ h_\theta(x^{(n)}) \end{pmatrix} - \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{pmatrix} = \begin{pmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ \vdots \\ h_\theta(x^{(n)}) - y^{(n)} \end{pmatrix}$$

and therefore

$$\|A\theta - y\|^2 = \sum_{j=1}^n (h_\theta(x^{(j)}) - y^{(j)})^2 = 2J(\theta).$$

And therefore $J(\theta)$ is minimal exactly when $\|A\theta - y\|$ is minimal.

Proposition 3.5. *If $\theta \in \mathbb{R}^{d+1}$ is a solution to*

$$A^T A\theta = A^T y, \tag{3.3}$$

then $\|A\theta - y\|$ is minimal and consequently $J(\theta)$ is minimal.

Proof. We want to find θ such that $\delta = \|A\theta - y\|$ is minimized. We note that $\delta \geq 0$, and $\delta = 0$ iff $\exists y$ such that $A\theta = y$. The closest point from the line (that is in the image of A) is achieved when $A\theta = P_{\text{im}(A)}(y)$. Since $A\theta = P_{\text{im}(A)}(y)$, we have that $A\theta - y \in \text{im}(A)^\perp = \ker(A^T)$. Therefore,

$$A^T(A\theta - y) = 0 \iff A^T A\theta - A^T y = 0 \iff A^T A\theta = A^T y.$$

□

In the proposition above, θ is not necessarily unique (there might be multiple solutions to Equation (3.3)). Also note that the matrix A is not necessarily invertible.

Proposition 3.6. *If $\ker(A) = \{0\}$, then $A^T A$ is invertible.*

Proof. We observe that $A^T A$ is a square matrix. Due to this, it is invertible if and only if

$$\ker(A^T A) = \{0\}.$$

Let $x \in \ker(A^T A)$. Then we have that:

$$\begin{aligned} A^T A x = 0 &\implies x^T A^T A x = 0 \\ &\implies (Ax)^T (Ax) = 0 \\ &\implies (Ax) \bullet (Ax) = 0 \\ &\implies Ax = 0. \end{aligned}$$

Thus, $x \in \ker(A)$, but $\ker(A) = \{0\}$, so $x = 0$. Thus, $\ker(A^T A) = \{0\}$, and thus $A^T A$ is invertible. □

As a result, if $\ker(A) = \{0\}$, a unique solution to Equation (3.3) exists and is given by

$$\theta = (A^T A)^{-1} A^T y. \tag{3.4}$$

We obtain this equation by multiplying equation (3.3) by $(A^T A)^{-1}$ from the left. Most of the time, the matrix A has more rows than columns, and due to this, the columns are usually linearly independent. As a result, we can often use equation (3.4).

Example 3.7. *We give an explicit example. Let*

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{pmatrix} \quad y = \begin{pmatrix} 3 \\ 0 \\ 0 \end{pmatrix}$$

The linear system $A\theta = y$ has no solutions $\theta \in \mathbb{R}^2$, but we can try to find the orthogonal projection of y onto $\text{im}(A)$. Recall, that we can calculate the image of A , by bringing the augmented matrix $(A \mid z)$ for $z = (z_1, z_2, z_3)^T$ onto row-reduced echelon form.

$$\begin{pmatrix} 1 & 1 & z_1 \\ 1 & 2 & z_2 \\ 1 & 3 & z_3 \end{pmatrix} \sim \begin{pmatrix} 1 & 1 & z_1 \\ 0 & 1 & z_2 - z_1 \\ 0 & 2 & z_3 - z_1 \end{pmatrix} \sim \begin{pmatrix} 1 & 1 & z_1 \\ 0 & 1 & z_2 - z_1 \\ 0 & 0 & z_3 - z_1 - 2(z_2 - z_1) \end{pmatrix} \sim \begin{pmatrix} 1 & 0 & 2z_1 - z_2 \\ 0 & 1 & z_2 - z_1 \\ 0 & 0 & z_3 + z_1 - 2z_2 \end{pmatrix}$$

So the elements $z = (z_1, z_2, z_3)^T$ in the image of A are exactly those with $z_3 = 2z_2 - z_1$, since then $z_3 + z_1 - 2z_2 = 0$. Let $v = (1, -2, 1)^T$. Then,

$$\text{im}(A) = \{z \in \mathbb{R}^3 \mid z \bullet v = 0\} = \text{span}\{v\}^\perp$$

According to Lemma 2.12, this should equal $\ker(A^T)$. To prove this, we need to bring the augmented matrix $(A^T \mid 0)$ onto row-reduced echelon form.

$$(A^T \mid 0) = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 2 & 3 & 0 \end{pmatrix} \sim \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 2 & 0 \end{pmatrix} \sim \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

From this, we conclude that $\ker(A^T) = \text{span}\{v\} = \text{im}(A)^\perp$. We then separate y into $y_{\text{im}(A)}$ (component of y that lies in $\text{im}(A)$) and y_\perp , the component of y perpendicular to $\text{im}(A)$. We have that

$$\begin{pmatrix} 3 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} u \\ v \\ 2v - u \end{pmatrix} + \begin{pmatrix} t \\ -2t \\ t \end{pmatrix}$$

Notice that $(1, 1, 1)^T \in \text{im}(A)$. By linearity of the scalar product,

$$y \bullet \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} u \\ v \\ 2v - u \end{pmatrix} \bullet \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} + \begin{pmatrix} t \\ -2t \\ t \end{pmatrix} \bullet \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \iff 3 = u + v + 2v - u \implies v = 1.$$

From this, we have that $t = \frac{v}{2} = 0.5$, and that $u = 3 - t = 2.5$. Therefore, we conclude that

$$P_{\text{im}(A)}(y) = \begin{pmatrix} 2.5 \\ 1 \\ -0.5 \end{pmatrix}.$$

Next, we would like to use the normal equation. Since the columns of A are linearly independent, we can use equation (3.4) to directly compute $A\theta = P_{\text{im}(A)}(y)$. We obtain

$$\theta = A(A^T A)^{-1} A^T y = \begin{pmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} 3 & 6 \\ 6 & 14 \end{pmatrix}^{-1} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 3 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2.5 \\ 1 \\ -0.5 \end{pmatrix}.$$

As such, we have proven that both methods will yield the same result (albeit the second method would be faster and better-suited for computers).

3.1.3 Solving linear regression with linear algebra in the case $\mathcal{X} = \mathbb{R}^d, \mathcal{Y} = \mathbb{R}^m$

Todo: Write out the general idea here Assume we have a training set $\mathcal{T} = ((x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)})) \in (\mathcal{X} \times \mathcal{Y})^n$ and want to describe it by a hypothesis of the form

$$h_{\theta}(x) := \theta_0 + \theta_1 x_1 + \dots + \theta_d x_d = \sum_{i=0}^d \theta_i x_i = \theta^T x \quad (3.5)$$

with parameters $\theta = (\theta_0, \theta_1, \dots, \theta_d)^T \in \mathbb{R}^{d+1}$ and $x \in \mathbb{R}^{d+1}$ (with $x_0 = 1$).

3.1.4 Linear regression for polynomial interpolation

Todo: mention that linear regression can not only be used for hypothesis of the form $h_{\theta}(x) = \sum_{i=0}^d \theta_i x_i$, but in general for something like $h_{\theta}(x) = \sum_{i=0}^d \theta_i f_i(x_i)$, where f_i can be actually an arbitrary function (e.g. they could be polynomials or log/exp).

3.2 Logistic regression

In this section we will discuss logistic regression. While it has a similar name to linear regression, it is used mostly in binary classification (which is not a form of regression).

Classification is a process of (using training examples) to build a function to classify data into discrete classes. This is in contrast with regression, where the goal is to predict continuous values.

Binary classification is a type of classification where our label space is given by $\mathcal{Y} = \{0, 1\}$. In other words, there are only two possible outcomes.

Let us now describe an example where logistic classification may be used. Set the feature space \mathcal{X} to be the hours spent studying for the exam and the label space \mathcal{Y} to be $\{0, 1\}$ where $y = 0$ represents the student failing the exam, while $y = 1$ represents the student passing the exam.

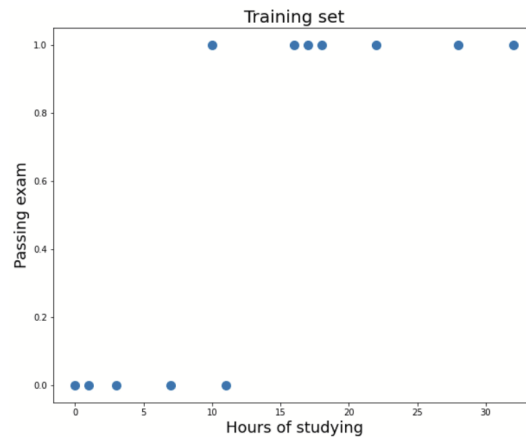


Figure 7: Training set with six training examples.

We notice that a linear function like in Linear Regression will not be a good hypothesis for the data.

A function whose form fits the data is the **logistic function** S defined by

$$S(x) = \frac{1}{1 + e^{-x}},$$

and its graph looks as follows

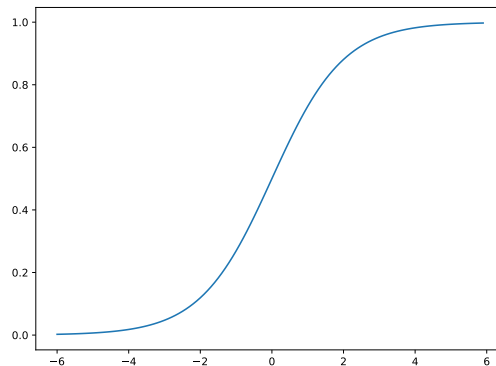


Figure 8: The graph of the logistic function $S(x)$ for $-6 \leq x \leq 6$.

The logistic function is an example of a **sigmoid function**, which in general denote bounded, differentiable, real functions that are defined for all real input values and that have a non-negative derivative at each point and exactly one inflection point (ELI5: functions which look like an "S"). They can be used to give an interpolation between 0 and 1.

The model for the hypothesis in the logistic regression case is given by

$$h_{\theta}(x) = S(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}},$$

where $\theta \in \mathbb{R}^{d+1}$ are the parameters we want to find.

In the $d = 1$ the hypothesis is given by

$$h_{\theta}(x) = S(\theta_0 + \theta_1 x_1) = \frac{1}{1 + e^{-\theta_0 - \theta_1 x_1}},$$

Recall (or just learn) that $P(A|B)$ refers to the conditional probability that event A occurs, given that event B occurred. For fixed θ , the hypothesis $h_{\theta}(x)$ can be interpreted as the conditional probability of passing the exam ($y = 1$) assuming that one studied x hours.

$$P(y = 1 | x; \theta) = h_{\theta}(x).$$

The probability of failing the exam is therefore:

$$P(y = 0 | x; \theta) = 1 - h_{\theta}(x).$$

We can combine both into one single function, which gives back the above cases for $y \in \{0, 1\}$:

$$P(y | x; \theta) = h_{\theta}(x)^y \cdot (1 - h_{\theta}(x))^{1-y}.$$

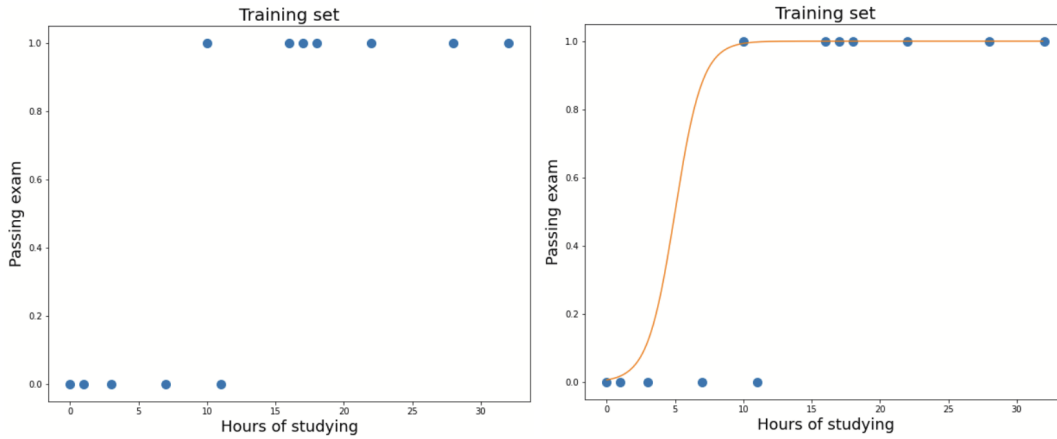


Figure 9: The plot of the training set and the hypothesis (sigmoid) function.

For a training set $\mathcal{T} = ((x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)}))$, we define the **likelihood** of θ by

$$L(\theta) = \prod_{j=1}^n P(y^{(j)} | x^{(j)}; \theta) = \prod_{j=1}^n h_{\theta}(x^{(j)})^{y^{(j)}} \cdot (1 - h_{\theta}(x^{(j)}))^{1-y^{(j)}}$$

$$L(\theta) = \prod_{j=1}^n P(y^{(j)} | x^{(j)}; \theta) = \prod_{j=1}^n h_{\theta}(x^{(j)})^{y^{(j)}} \cdot (1 - h_{\theta}(x^{(j)}))^{1-y^{(j)}}$$

The **log likelihood** of θ is given by

$$\ell(\theta) = \log L(\theta) = \sum_{j=1}^n \left(y^{(j)} \log h_{\theta}(x^{(j)}) + (1 - y^{(j)}) \log(1 - h_{\theta}(x^{(j)})) \right)$$

Maximize this function by using gradient ascent

$$\theta := \theta + \alpha \nabla \ell(\theta).$$

We need to calculate the gradient

$$\nabla \ell(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \ell(\theta) \\ \frac{\partial}{\partial \theta_1} \ell(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_d} \ell(\theta) \end{pmatrix}$$

Lemma 3.8. *The logistic function satisfies the following differential equation*

$$S'(x) = S(x)(1 - S(x)).$$

Proof. We differentiate S with respect to x by the chain rule:

$$\begin{aligned} S'(x) &= \frac{d}{dx} \left[\frac{1}{1 + e^{-x}} \right] = - \left(\frac{1}{1 + e^{-x}} \right)^2 \frac{d}{dx} [1 + e^{-x}] = \left(\frac{1}{1 + e^{-x}} \right) \cdot \left(\frac{e^{-x}}{1 + e^{-x}} \right) \\ &= \left(\frac{1}{1 + e^{-x}} \right) \cdot \left(1 - \frac{1}{1 + e^{-x}} \right) = S(x)(1 - S(x)). \end{aligned}$$

□

Proposition 3.9. *The gradient of the log likelihood function ℓ is given by*

$$\nabla \ell(\theta) = \sum_{j=1}^n \left(y^{(j)} - h_{\theta}(x^{(j)}) \right) x^{(j)}.$$

Proof. We need to show that for $i = 0, \dots, d$ we have

$$\frac{\partial}{\partial \theta_i} \ell(\theta) = \sum_{j=1}^n \left(y^{(j)} - h_{\theta}(x^{(j)}) \right) x_i^{(j)}.$$

We take the derivative of ℓ with respect to θ_i :

$$\begin{aligned} \frac{\partial}{\partial \theta_i} \ell(\theta) &= \sum_{j=1}^n \left(y^{(j)} \frac{\partial}{\partial \theta_i} \left[\ln h_{\theta}(x^{(j)}) \right] + (1 - y^{(j)}) \frac{\partial}{\partial \theta_i} \left[\ln(1 - h_{\theta}(x^{(j)})) \right] \right) \\ &= \sum_{j=1}^n \left(y^{(j)} \frac{1}{h_{\theta}(x^{(j)})} \frac{\partial}{\partial \theta_i} \left[h_{\theta}(x^{(j)}) \right] + \frac{(1 - y^{(j)})}{1 - h_{\theta}(x^{(j)})} \frac{\partial}{\partial \theta_i} \left[1 - h_{\theta}(x^{(j)}) \right] \right) \\ &= \sum_{j=1}^n \left(y^{(j)} \frac{h'_{\theta}(x^{(j)})}{h_{\theta}(x^{(j)})} + \frac{(y^{(j)} - 1)h'_{\theta}(x^{(j)})}{1 - h_{\theta}(x^{(j)})} \right) x_i^{(j)} \\ &= \sum_{j=1}^n \left(\frac{y^{(j)}h'_{\theta}(x^{(j)})(1 - h_{\theta}(x^{(j)})) + (y^{(j)} - 1)h'_{\theta}(x^{(j)})h_{\theta}(x^{(j)})}{h_{\theta}(x^{(j)})(1 - h_{\theta}(x^{(j)}))} \right) x_i^{(j)} \end{aligned}$$

Note that $h_{\theta}(x^{(j)}) = S(\theta^T x)$, so $h'_{\theta}(x^{(j)}) = h_{\theta}(x^{(j)})(1 - h_{\theta}(x^{(j)}))$. Thus, we have

$$\frac{\partial}{\partial \theta_i} \ell(\theta) = \sum_{j=1}^n \left(y^{(j)} - y^{(j)}h_{\theta}(x^{(j)}) + y^{(j)}h_{\theta}(x^{(j)}) - h_{\theta}(x^{(j)}) \right) x_i^{(j)} = \sum_{j=1}^n \left(y^{(j)} - h_{\theta}(x^{(j)}) \right) x_i^{(j)}.$$

□

The update rule for the gradient ascent is therefore

$$\theta := \theta + \alpha \nabla \ell(\theta) = \theta + \alpha \sum_{j=1}^n \left(y^{(j)} - h_{\theta}(x^{(j)}) \right) x^{(j)}$$

Python Example 3.10. We can implement this method in Python using the following code:

```

1 import numpy as np # For doing math
2 from matplotlib import pyplot as plt # For plotting
3
4 # Training set
5 Tx = np.array([0, 1,3, 7, 10,11,16,17,18,22,28,32])
6 Ty = np.array([0, 0, 0, 0, 1,0,1,1,1,1,1])
7
8 # Hypothesis h_theta(x)
9 # If we have d features, we expect x to be of length d+1, with x[0]=1.
10 def h(theta, x):
11     return(1/(1+np.exp(- np.transpose(theta).dot(x) )))
12
13 # Number of training examples
14 n = len(Tx)
15
16 # Log likelihood in the d=1 case (just one feature given as the entries in Tx)
17 def loglike(theta):
18     ret=0
19     for j in range(n):
20         ret+= Ty[j]*np.log( h(theta,[1,Tx[j]]) ) + (1-Ty[j])*np.log( 1.0 - h(theta,[1,Tx[
21             j]]) )
22     return(ret)
23
24 # The (normalized) gradient for the log likelihood at a theta
25 def gradient(theta):
26     g=np.array([0.0,0])
27     for j in range(len(Tx)):
28         g[0]+= (Ty[j] - h(theta,[1,Tx[j]]))*1
29         g[1]+= (Ty[j] - h(theta,[1,Tx[j]]))*Tx[j]
30     return(g/np.linalg.norm(g))
31
32 # We now start the Gradient Ascent method.
33 # Start with some value for theta
34 theta=np.array([-10,0])
35
36 listt0 = np.array([])
37 listt1 = np.array([])
38
39 # learning rate
40 alpha=0.1
41
42 # number of steps
43 steps = 20000
44
45 # gradient ascent
46 for s in range(steps):
47     listt0 = np.append(listt0,theta[0]) # save data for drawing
48     listt1 = np.append(listt1,theta[1])
49     theta = theta + alpha*gradient(theta)
50
51 print("Gradient ascent gives after",steps,"steps: ", theta)

```

3.3 Naive Bayes

All learning algorithms we considered so far are examples of so-called **discriminative learning algorithms**. This means that we tried to learn the probability $P(y | x)$, where y was a label and x a feature. In other words: Discriminative learning algorithms learn how likely a label is for a given

feature (e.g. how likely is it, that a student passed the exam $y = 1$, given that the student studied x hours?). We did this by learning an explicit hypothesis $h_\theta : \mathcal{X} \rightarrow \mathcal{Y}$.

In contrast, a so-called **generative learning algorithm** learns $P(x | y)$, i.e. the probability that a feature appears given a label (class). So in the exam example, we might ask how the learning hours look like for a student who passes the exam. In addition, a generative learning algorithm will also learn $P(y)$, the probability that a certain label appears (e.g. how likely is it in general that an arbitrary student passes the exam?).

The natural question which arises is: "How does knowing $P(x | y)$ and $P(y)$ help us when our goal is to assign a class y to a given feature x ?" The answer to this lies in **Bayes Rule**, which states that

$$P(y | x) = \frac{P(x | y)P(y)}{P(x)}.$$

Let us now introduce the notion of **conditionally independent** variables. Let A , B and C be events. A and B are conditionally independent given C if given knowledge that C occurs, knowledge on A provides no information on B , and knowledge on B provides no knowledge of A .

As an example, let A be "ability to do math", B be "foot size", and C be "age". A and B are not independent, since the foot size of a person hints at their age, which provides some knowledge on ability to do math. However, A and B are conditionally independent given C since if we know someone's age (C), knowing their ability to do math gives no information on their foot size, and vice versa.

We now introduce the **chain rule for probabilities**, which states that for events A and B ,

$$P(A, B) = P(A | B)P(B),$$

where $P(A, B)$ is the probability that both A and B occurs, $P(A | B)$ is the conditional probability of A given B , and $P(B)$ the probability of B .

We now give an example of a generative learning algorithm. In this algorithm, we use the **Naive Bayes assumption**, which asserts that all features are conditionally independent given the label. If x_1, x_2 are conditionally independent over y , then we have:

$$P(x_1 | y, x_2) = P(x_1 | y).$$

We then find an expression for $P(x | y)$. We do this by using the chain rule:

$$P(x | y) = P(x_1, \dots, x_d | y) = \prod_{i=1}^d P(x_i | y).$$

The model is then parameterized by the following:

$$\phi_{i|y=0} = P(x_i = 1 | y = 0), \quad \phi_{i|y=1} = P(x_i = 1 | y = 1) \quad \text{and} \quad \phi_{y=1} = P(y = 1).$$

By Bayes' rule we get (for a feature $x \in \mathcal{X}$):

$$P(y = 1 | x) = \frac{P(x | y = 1)P(y = 1)}{P(x)}.$$

We calculate $P(x)$ by the following:

$$P(x) = P(x | y = 0)P(y = 0) + P(x | y = 1)P(y = 1).$$

	\mathcal{X}	\mathcal{Y}
1	Do math today	0
2	Buy	1
3	Buy book	0
4	Today do math drugs	1
5	Buy drugs book today	1

Example 3.11. An example is email spam filtering. Here, the feature space \mathcal{X} are emails, and the labels are not spam ($y = 0$) and spam ($y = 1$). The table below gives the training set for this algorithm:

We now tabulate the probabilities that certain words appear in spam or not spam emails:

i		appears in non-spam: $P(x_i = 1 \mid y = 0)$	appears in spam: $P(x_i = 1 \mid y = 1)$
1	book	$\frac{1}{2}$	$\frac{1}{3}$
2	buy	$\frac{1}{2}$	$\frac{2}{3}$
3	do	$\frac{1}{2}$	$\frac{1}{3}$
4	drugs	0	$\frac{2}{3}$
5	math	$\frac{1}{2}$	$\frac{1}{3}$
6	today	$\frac{1}{2}$	$\frac{2}{3}$

When we get a new feature (i.e. someone sends us an email) $x = (x_1, \dots, x_d) \in \mathcal{X}$, we want to calculate the probability that this new email is spam. We do this by calculating for each word $i = 1, \dots, d$ the probabilities

$$\begin{aligned}
 P(x_i = 1 \mid y = 1) &= \phi_{i|y=1}, & P(x_i = 1 \mid y = 0) &= \phi_{i|y=0}, \\
 P(x_i = 0 \mid y = 1) &= 1 - \phi_{i|y=1}, & P(x_i = 0 \mid y = 0) &= 1 - \phi_{i|y=0}.
 \end{aligned}$$

We define the **indicator function** I for a statement S by

$$I(S) = \begin{cases} 1, & S \text{ is true} \\ 0, & S \text{ is false} \end{cases}.$$

Given a training set $\mathcal{T} = ((x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)}))$ we can calculate them by

$$\begin{aligned}
 \phi_{i|y=1} &= \frac{\sum_{j=1}^n I(x_i^{(j)} = 1 \wedge y^{(j)} = 1)}{\sum_{j=1}^n I(y^{(j)} = 1)} \\
 \phi_{i|y=0} &= \frac{\sum_{j=1}^n I(x_i^{(j)} = 1 \wedge y^{(j)} = 0)}{\sum_{j=1}^n I(y^{(j)} = 0)} \\
 \phi_{y=1} &= \frac{1}{n} \sum_{j=1}^n I(y^{(j)} = 1)
 \end{aligned}$$

The probability of a new email $x = (x_1, \dots, x_d)^T$ being spam is then

$$P(y = 1 \mid x) = \frac{P(x \mid y = 1)P(y = 1)}{P(x)}$$

$$= \frac{\prod_{i=1}^d P(x_i | y = 1) \cdot \phi_{y=1}}{\prod_{i=1}^d P(x_i | y = 1) \cdot \phi_{y=1} + \prod_{i=1}^d P(x_i | y = 0)(1 - \phi_{y=1})}.$$

Example 3.12 (Continuation of Example 3.11). *We now want to calculate the probability that the email "Buy books today" is spam. We consider the feature (in this case, $\mathcal{X} = \{0, 1\}^d$, where d is the number of words in dictionary).*

$$x = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \in \mathcal{X} = \{0, 1\}^6$$

Using the formulae given above, we obtain the following result:

$$P(y = 1) = \phi_{y=1} = \frac{3}{5} \quad P(y = 0) = 1 - \phi_{y=1} = \frac{2}{5}.$$

Using the data in the table we can read off $P(x_i = 1 | y = a) = \phi_{i|y=a}$ and $P(x_i = 0 | y = a) = 1 - \phi_{i|y=a}$ for $a \in \{0, 1\}$ and get

$$\prod_{i=1}^d P(x_i | y = 0) = \frac{1}{2} \cdot \frac{1}{2} \cdot \left(1 - \frac{1}{2}\right) \cdot (1 - 0) \cdot \left(1 - \frac{1}{2}\right) \cdot \frac{1}{2} = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{32},$$

$$\prod_{i=1}^d P(x_i | y = 1) = \frac{1}{3} \cdot \frac{2}{3} \cdot \left(1 - \frac{1}{3}\right) \cdot \left(1 - \frac{2}{3}\right) \cdot \left(1 - \frac{1}{3}\right) \cdot \frac{2}{3} = \frac{1}{3} \cdot \frac{2}{3} \cdot \frac{2}{3} \cdot \frac{1}{3} \cdot \frac{2}{3} \cdot \frac{2}{3} = \frac{16}{729}.$$

Using the formula we have for $P(y = 1 | x)$, we obtain

$$P(y = 1 | x) = \frac{\frac{16}{729} \cdot \frac{3}{5}}{\frac{16}{729} \cdot \frac{3}{5} + \frac{1}{32} \cdot \frac{2}{5}} = \frac{256}{499} \approx 0.51$$

and therefore the probability that the email "Buy books today" is spam is 51%.

Above approach has an obvious problem: What if we encounter an email which contains a word we do not know? If we would just add this word to our dictionary, then the probabilities of this word appearing in a spam and in a non-spam email would be 0, which in the end would lead to the non-defined expression $P(y = 1 | x) = \frac{0}{0}$. Similarly since $P(x_4 = 1 | y = 0) = 0$, any email that contains the word 'drugs' is spam. These examples illustrates the problem of zero probability. The basic idea to fix this, is to assume that every event has a non-zero probability. This is called **Laplace smoothing**, which assigns a non-zero probability to any event. Instead of the parameters described above, we use the following parameters:

$$\tilde{\phi}_{i|y=1} = \frac{1 + \sum_{j=1}^n I(x_i^{(j)} = 1 \wedge y^{(j)} = 1)}{2 + \sum_{j=1}^n I(y^{(j)} = 1)}$$

$$\tilde{\phi}_{i|y=0} = \frac{1 + \sum_{j=1}^n I(x_i^{(j)} = 1 \wedge y^{(j)} = 0)}{2 + \sum_{j=1}^n I(y^{(j)} = 0)}$$

and then use

$$\begin{aligned} P(x_i = 1 \mid y = 1) &= \tilde{\phi}_{i|y=1}, & P(x_i = 1 \mid y = 0) &= \tilde{\phi}_{i|y=0}, \\ P(x_i = 0 \mid y = 1) &= 1 - \tilde{\phi}_{i|y=1}, & P(x_i = 0 \mid y = 0) &= 1 - \tilde{\phi}_{i|y=0}. \end{aligned}$$

to calculate

$$P(y = 1 \mid x) = \frac{\prod_{i=1}^d P(x_i \mid y = 1) \cdot \phi_{y=1}}{\prod_{i=1}^d P(x_i \mid y = 1) \cdot \phi_{y=1} + \prod_{i=1}^d P(x_i \mid y = 0)(1 - \phi_{y=1})}.$$

One possible interpretation of this is that we add 4 imaginary emails:

- (i) A spam email which contains every word.
- (ii) A spam email which does not contain any word.
- (iii) A non-spam email which contains every word.
- (iv) A non-spam email which does not contain any word.

Therefore it makes sense to add 2 to the denominator of $\tilde{\phi}_{i|y=1}$ (resp. $\tilde{\phi}_{i|y=0}$), since it counts the number of spam (resp. non-spam) emails. Adding 1 to the numerator also makes sense, since out of the 4 new emails just one of them contains the word and is spam (resp. non-spam). Notice that we will not give another value for $\phi_{y=1}$, since $\frac{1}{n} \sum_{j=1}^n I(y^{(j)} = 1)$ is usually always non-zero and not 1 as long as there are spam and non-spam emails in our training set (which should be the case for any usable training set).

Example 3.13 (Continuation II of Example 3.11). *The values for these parameters are given in the table below:*

i	$\tilde{\phi}_{i y=0}$	$\tilde{\phi}_{i y=1}$
1	1/2	2/5
2	1/2	3/5
3	1/2	2/5
4	1/4	3/5
5	1/2	2/5
6	1/2	3/5

Then the products of the probabilities are given by $P(x_i = 1 \mid y = a) = \tilde{\phi}_{i|y=a}$ and $P(x_i = 0 \mid y = a) = 1 - \tilde{\phi}_{i|y=a}$ for $a \in \{0, 1\}$, i.e. we get



$$\begin{aligned} \prod_{i=1}^d P(x_i \mid y = 0) &= \frac{1}{2} \cdot \frac{1}{2} \cdot \left(1 - \frac{1}{2}\right) \cdot \left(1 - \frac{1}{4}\right) \cdot \left(1 - \frac{1}{2}\right) \cdot \frac{1}{2} = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{3}{4} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{3}{128}, \\ \prod_{i=1}^d P(x_i \mid y = 1) &= \frac{2}{5} \cdot \frac{3}{5} \cdot \left(1 - \frac{2}{5}\right) \cdot \left(1 - \frac{3}{5}\right) \cdot \left(1 - \frac{2}{5}\right) \cdot \frac{3}{5} = \frac{2}{5} \cdot \frac{3}{5} \cdot \frac{3}{5} \cdot \frac{2}{5} \cdot \frac{3}{5} \cdot \frac{3}{5} = \frac{324}{15625}. \end{aligned}$$

Using the formula for the probability of $P(y = 1 \mid x)$, we obtain

$$P(y = 1 \mid x) = \frac{\prod_{i=1}^d P(x_i \mid y = 1) \cdot \phi_{y=1}}{\prod_{i=1}^d P(x_i \mid y = 1) \cdot \phi_{y=1} + \prod_{i=1}^d P(x_i \mid y = 0)(1 - \phi_{y=1})}$$

$$= \frac{\frac{324}{15625} \cdot \frac{3}{5}}{\frac{324}{15625} \cdot \frac{3}{5} + \frac{3}{128} \cdot \frac{2}{5}} \approx 0.57,$$

and therefore the new prediction is that the email "Buy books today" is 57% spam.

 **Todo:** include Python examples (e.g. Fall 2022, Lecture 7 Notebook, https://colab.research.google.com/drive/1XU8NcIWbf1io_dDbMKLKqnKuy9MmNqu2?usp=sharing) 

3.4 Gaussian Discriminant Analysis

We will now sketch another variant of the Naive Bayes algorithm. In this algorithm, we assume that $P(x | y)$ is distributed according to a multivariate normal distribution.

In a lot of real life scenarios a normal (or also called Gaussian) distribution appears. In the one dimensional case ($x \in \mathbb{R}$) this means, that we have the following **probability density function**

$$p(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2},$$

where $\mu \in \mathbb{R}$ is the **mean** and $\sigma \in \mathbb{R}$ is the **standard deviation**.

In d -dimension the normal distribution $\mathcal{N}(\mu, \Sigma)$, called **multivariate normal distribution**, is parameterized by a **mean vector** $\mu \in \mathbb{R}^d$ and a **covariance matrix** $\Sigma \in \mathbb{R}^{d \times d}$, where Σ is symmetric and positive semi-definite. The density function is given by

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{d}{2}} \sqrt{\det(\Sigma)}} \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu)\right).$$

By $x \sim \mathcal{N}(\mu, \Sigma)$ we mean that the variable x follows a multivariate normal distribution with mean vector μ and covariance matrix Σ .

For the **Gaussian discriminant analysis (GDA)** model we assume that we have d features, i.e. $\mathcal{X} = \mathbb{R}^d$, and again two labels $\mathcal{Y} = \{0, 1\}$. We want to model again $p(x | y)$ by assuming that

$$\begin{aligned} y &\sim \text{Bernoulli}(\phi), \\ x | y = 0 &\sim \mathcal{N}(\mu_0, \Sigma), \\ x | y = 1 &\sim \mathcal{N}(\mu_1, \Sigma) \end{aligned}$$

for some $\mu_0, \mu_1 \in \mathbb{R}^d$, $\Sigma \in \mathbb{R}^{d \times d}$ and $\phi \in \mathbb{R}$, i.e. we have

$$\begin{aligned} p(x|y=0) &= \frac{1}{(2\pi)^{\frac{d}{2}} \sqrt{\det(\Sigma)}} \exp\left(-\frac{1}{2}(x - \mu_0)^T \Sigma^{-1} (x - \mu_0)\right), \\ p(x|y=1) &= \frac{1}{(2\pi)^{\frac{d}{2}} \sqrt{\det(\Sigma)}} \exp\left(-\frac{1}{2}(x - \mu_1)^T \Sigma^{-1} (x - \mu_1)\right). \end{aligned}$$

Here $y \sim \text{Bernoulli}(\phi)$ means y follows a Bernoulli distribution, which just means that $p(y = 1) = \phi$ and $p(y = 0) = 1 - \phi$, which we can again write compactly as

$$p(y) = \phi^y (1 - \phi)^{1-y}.$$

The goal now is again to find the best possible parameters $\mu_0, \mu_1 \in \mathbb{R}^d$, $\Sigma \in \mathbb{R}^{d \times d}$ and $\phi \in \mathbb{R}$ for a given training set.

Given a training set $\mathcal{T} = ((x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)})) \in (\mathcal{X} \times \mathcal{Y})^n$ we define the **log-likelihood** by

$$\begin{aligned} \ell(\phi, \mu_0, \mu_1, \Sigma) &= \log \prod_{i=1}^n p(x^{(i)}, y^{(i)}; \phi, \mu_0, \mu_1, \Sigma) \\ &= \log \prod_{i=1}^n p(x^{(i)} | y^{(i)}; \mu_0, \mu_1, \Sigma) p(y^{(i)}; \phi). \end{aligned}$$

This measures how likely the parameters $\phi, \mu_0, \mu_1, \Sigma$ are for the given training set.

Proposition 3.14. *The log-likelihood gets maximized by choosing the following parameters*

$$\begin{aligned} \phi &= \frac{1}{n} \sum_{i=1}^n I(y^{(i)} = 1), \\ \mu_c &= \frac{\sum_{i=1}^n I(y^{(i)} = c) x^{(i)}}{\sum_{i=1}^n I(y^{(i)} = c)}, \quad (c \in \{0, 1\}) \\ \Sigma &= \frac{1}{n} \sum_{i=1}^n (x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T. \end{aligned}$$

Proof. $\{\overset{\text{iii}}{\Leftarrow}$ **Todo:** Find maxima by setting derivative (gradient) to zero $\overset{\text{iii}}{\Rightarrow}\}$.

□

4 Neural Networks

In the realm of machine learning and artificial intelligence, the ultimate aim is often to find ways to accurately approximate or model complex functions that map from \mathbb{R}^n to \mathbb{R}^m . These functions may represent anything from simple linear regressions to intricate relationships governing natural phenomena or financial markets.

4.1 Multi-layer fully connected feedforward neural networks

In this first section, our primary focus will be on **multi-layer fully connected feedforward neural networks**. A multi-layer fully connected feedforward neural network consists of an input layer, one or more hidden layers, and an output layer, where each neuron in a layer is connected to every neuron in the adjacent layers, and information moves in a single direction—from the input layer through the hidden layers to the output layer, without any cycles (see Figure 10). We will just refer to them as neural networks in the following.

Note that some methods we've already explored can be considered special cases within this broader framework. For instance, logistic regression, covered in Section 3.2, can be thought of as a neural network with just a single neuron in the output layer. In this simplified model, the input features are linearly combined using weighted sums followed by a bias term, much like the hidden layers in a more complex neural network. This linear combination then undergoes a non-linear transformation via an activation function, often the logistic sigmoid function, to produce the output.

Definition 4.1. (i) An **activation function** is a (usually non-linear) function $\sigma : \mathbb{R}^n \rightarrow \mathbb{R}^n$. Often activation functions are given as functions $\sigma : \mathbb{R} \rightarrow \mathbb{R}$, which are then extended to a function $\sigma : \mathbb{R}^n \rightarrow \mathbb{R}^n$ by setting

$$\sigma \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} := \begin{pmatrix} \sigma(x_1) \\ \vdots \\ \sigma(x_n) \end{pmatrix}.$$

(ii) The **rectified linear unit (ReLU)** is the activation function defined for $x \in \mathbb{R}$ by

$$\text{ReLU}(x) = \max\{0, x\}.$$

(iii) The **sigmoid** function is the activation function defined for $x \in \mathbb{R}$ by

$$S(x) = \frac{1}{1 + e^{-x}}.$$

(iv) The **softmax** function is given

$$\begin{aligned} \text{softmax} : \mathbb{R}^n &\rightarrow \mathbb{R}^n, \\ \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} &\mapsto \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}, \end{aligned}$$

where $y_i = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$ for $i = 1, \dots, n$.

A **neuron** consists of a weight vector $w \in \mathbb{R}^n$, a bias $b \in \mathbb{R}$ and an activation function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$. A layer will consist of a collection of neurons and we will usually assume that each neuron in a given layer has the same activation function. Collecting all weight vectors and biases of the neurons in a layer will give rise to a weight matrix and bias vector. Instead of viewing neurons as a single object we therefore just consider layers as a whole.

Definition 4.2. A (dense) layer $L = (W, b, \sigma)$ of input size $n \geq 1$ and (output) size $m \geq 1$ consists of

- (i) a weight matrix $W \in \mathbb{R}^{m \times n}$,
- (ii) a bias vector $b \in \mathbb{R}^m$,
- (iii) and an activation function $\sigma : \mathbb{R}^m \rightarrow \mathbb{R}^m$.

Definition 4.3. For $r \geq 1$ a r -layer neural network $N = (L^{[1]}, \dots, L^{[r]})$ of input size n and output size m , consists of a collection of layers $L^{[i]} = (W^{[i]}, b^{[i]}, \sigma^{[i]})$ for $i = 1, \dots, r$, such that $W^{[i]} \in \mathbb{R}^{m_i \times m_{i-1}}$ with $m_0 = n$ and $m_r = m$.

The following shows an example of a 3-layer neural network with input size 5 and output size 1 and $m_0 = 5, m_1 = 3, m_2 = 2$, and $m_3 = 1$:

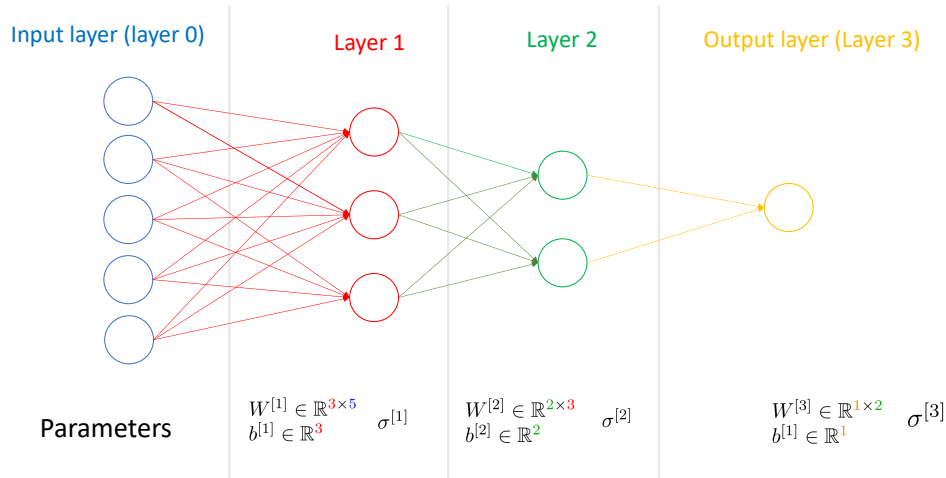


Figure 10: An example of a neural network with $(3 \cdot 5 + 3) + (2 \cdot 3 + 2) + (1 \cdot 2 + 1) = 29$ parameters.

Definition 4.4. Let $N = (L^{[1]}, \dots, L^{[r]})$ be a r -layer neural network of input size n and output size m . We want to view it as a function $N : \mathbb{R}^n \rightarrow \mathbb{R}^m$ by defining $N(x)$ for an **input** $x \in \mathbb{R}^n$ by the output of its last layer $N(x) = a^{[r]}$. Here we define for $i = 1, \dots, r$ the following:

- (i) The **linear part** of the layer $L^{[i]}$ is defined by

$$z^{[i]} = W^{[i]}a^{[i-1]} + b^{[i]},$$

where $a^{[i-1]} \in \mathbb{R}^{m_{i-1}}$ is the output from the previous layer. In the case $i = 1$ we set $a^{[0]} = x$.

- (ii) The **output** of the layer $L^{[i]}$ is defined by applying the activation function to the linear part, i.e.

$$a^{[i]} = \sigma^{[i]}(z^{[i]}) \in \mathbb{R}^{m_i}$$

Evaluating a neural network N at an input x as in above definition is called **forward pass**. In particular, for any input x we can always evaluate the values $z^{[i]}, a^{[i]}$ for all layers $i = 1, \dots, r$.

4.2 Training the neural network: Backpropagation

Our neural network is supposed to approximate a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ (In the example $n = 5$ and $m = 1$). So we want to find the best possible choices of parameters (i.e. weight matrices and biases) such that $N(x)$ is a good approximation for $f(x)$. Recall that we had the following notation (notice that we now use t for the number of training examples, since we use n for the dimension of the feature space):

- Input values (Feature space): $\mathcal{X} = \mathbb{R}^n$
- Output value (Label space): $\mathcal{Y} = \mathbb{R}^m$
- Training example: $(x, y) \in \mathcal{X} \times \mathcal{Y}$.
- Training set (with t training examples): $\mathcal{T} = ((x^{(1)}, y^{(1)}), \dots, (x^{(t)}, y^{(t)})) \in (\mathcal{X} \times \mathcal{Y})^t$.
- Learning algorithm: An algorithm to create a hypothesis h out of a trainings set \mathcal{T} .

Given a training set \mathcal{T} , a **cost function** is a map from the space of parameters to \mathbb{R} , which measures how good the current parameters are with respect to the training set.

For example, in Figure 10 we have 29 parameters, i.e. a cost function $J : \mathbb{R}^{29} \rightarrow \mathbb{R}$, which we want to minimize.

In this case we need to calculate the **gradient of J**

$$\nabla J = \begin{pmatrix} \frac{\partial}{\partial \theta_1} J \\ \frac{\partial}{\partial \theta_2} J \\ \vdots \\ \frac{\partial}{\partial \theta_{29}} J \end{pmatrix}$$

Suppose we have a training set $\mathcal{T} = ((x^{(1)}, y^{(1)}), \dots, (x^{(t)}, y^{(t)})) \in (\mathcal{X} \times \mathcal{Y})^t$.

For a neural network N , we will think of the collection of all weight matrices and biases as a vector $\theta \in \mathbb{R}^d$, called the **parameters**. For example, θ_1 could be the top left entry of the first weight matrix. The output of a neural network depends on the current choice of θ and therefore we write N_θ .

For example, we could use the sum of squares as a cost function

$$J(\theta) = \frac{1}{2} \sum_{j=1}^t \left\| N_\theta(x^{(j)}) - y^{(j)} \right\|^2.$$

Example 4.5. We will start with a simple one dimensional example, i.e. we assume that each layer just has one neuron. Consider now a 2-layer neural network with two layers of size 1.



This means, we have four parameters $w^{[1]}, w^{[2]}, b^{[1]}, b^{[2]} \in \mathbb{R}$ and we could write the parameter vector θ and the gradient of J as

$$\theta = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \end{pmatrix} = \begin{pmatrix} w^{[1]} \\ b^{[1]} \\ w^{[2]} \\ b^{[2]} \end{pmatrix}, \quad \nabla J = \begin{pmatrix} \frac{\partial J}{\partial w^{[1]}} \\ \frac{\partial J}{\partial b^{[1]}} \\ \frac{\partial J}{\partial w^{[2]}} \\ \frac{\partial J}{\partial b^{[2]}} \end{pmatrix}.$$

Let us now assume that we have one training example, i.e. $\mathcal{T} = ((x, y)) \in (\mathbb{R} \times \mathbb{R})^1$. In this case, the cost function is just $J(\theta) = \frac{1}{2} (N_\theta(x) - y)^2$. To calculate the gradient ∇J , we therefore want to understand the dependence of J on the parameters $w^{[1]}, w^{[2]}, b^{[1]}, b^{[2]} \in \mathbb{R}$, which can be done using the usual chain rule: Since we have

$$\begin{aligned} J(\theta) &= \frac{1}{2} (N_\theta(x) - y)^2 = \frac{1}{2} (a^{[2]} - y)^2, \\ a^{[2]} &= \sigma^{[2]}(z^{[2]}), \\ z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \end{aligned}$$

we get, for example (by using the Leibniz notation for the chain rule),

$$\begin{aligned} \frac{\partial J}{\partial b^{[2]}} &= \frac{\partial J}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial b^{[2]}} \\ &= (a^{[2]} - y) \cdot (\sigma^{[2]})'(z^{[2]}) \cdot 1. \end{aligned}$$

Similarly, we get

$$\begin{aligned} \frac{\partial J}{\partial W^{[2]}} &= \frac{\partial J}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial W^{[2]}} \\ &= (a^{[2]} - y) \cdot (\sigma^{[2]})'(z^{[2]}) \cdot a^{[1]}. \end{aligned}$$

Notice that in the second calculation we can use the result for $\delta^{[2]} := \frac{\partial J}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}}$ from the calculation before, i.e. $\frac{\partial J}{\partial W^{[2]}} = \delta^{[2]} \cdot a^{[1]}$. This already gives the two lowest entries of ∇J . For the other two, we can do a similar calculation and get

$$\begin{aligned} \frac{\partial J}{\partial b^{[1]}} &= \underbrace{\frac{\partial J}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}}}_{\delta^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial b^{[1]}} \\ &= \delta^{[2]} \cdot W^{[2]} \cdot (\sigma^{[1]})'(z^{[1]}) \cdot 1. \end{aligned}$$

Again, it is useful to set $\delta^{[1]} = \frac{\partial J}{\partial z^{[1]}}$. Above calculation then reads

Now we want to generalize this example and explain how to calculate the gradient for arbitrary neural networks. As a shorthand, for a vector $x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n$ and a function f that is differentiable with

respect to x_i for all $1 \leq i \leq n$, we define the derivative $\frac{\partial f}{\partial x}$ to be

$$\frac{\partial f}{\partial x} := \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}.$$

More generally, for a matrix $Y = \begin{pmatrix} y_{1,1} & \cdots & y_{1,n} \\ \vdots & \ddots & \vdots \\ y_{m,1} & \cdots & y_{m,n} \end{pmatrix} \in M_{m \times n}(\mathbb{R})$ and a function f that is differen-

tiable with respect to $y_{i,j}$ for all $1 \leq i \leq r, 1 \leq j \leq s$, we define the derivative $\frac{\partial f}{\partial Y}$ to be

$$\frac{\partial f}{\partial Y} := \begin{pmatrix} \frac{\partial f}{\partial y_{1,1}} & \cdots & \frac{\partial f}{\partial y_{1,s}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial y_{r,1}} & \cdots & \frac{\partial f}{\partial y_{r,s}} \end{pmatrix}.$$

We will also use \odot to represent element-wise multiplication. In other words, for two vectors of the

same dimension $v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}, w = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} \in \mathbb{R}^n$, we have

$$v \odot w = \begin{pmatrix} v_1 \cdot w_1 \\ v_2 \cdot w_2 \\ \vdots \\ v_n \cdot w_n \end{pmatrix}.$$

Besides these notations we will need the following generalization of the chain rule in order to describe Backpropagation for arbitrary neural networks:

Lemma 4.6. (*General chain rule*) Suppose the variable J depends on the variables $\theta_1, \dots, \theta_p$ via the intermediate variables g_1, \dots, g_k :

$$g_j = g_j(\theta_1, \dots, \theta_p), \quad \forall j \in \{1, \dots, k\}.$$

$$J = J(g_1, \dots, g_k).$$

Here we overload the meaning of g_j 's: they denote both the intermediate variables but also the functions used to compute the intermediate variables. Then we have for all $i = 1, \dots, p$

$$\frac{\partial J}{\partial \theta_i} = \sum_{j=1}^k \frac{\partial J}{\partial g_j} \frac{\partial g_j}{\partial \theta_i} \tag{7.29}$$

Proof. See your favorite Calculus 2 lecture. □

Proposition 4.7. Let $N = (L^{[1]}, \dots, L^{[r]})$ be a r -layer neural network. For $1 \leq i \leq r$, we define

$$\delta^{[i]} := \frac{\partial J}{\partial z^{[i]}}.$$

Then

$$\delta^{[r]} = \frac{\partial J}{\partial a^{[r]}} \odot (\sigma^{[i]})'(z^{[r]})$$

and for $1 \leq i < r$, we have

$$\delta^{[i]} = \left(W^{[i+1]T} \delta^{[i+1]} \right) \odot (\sigma^{[i]})'(z^{[i]}).$$

Proof. The equation for $\delta^{[r]}$ follows immediately from the chain rule.

Denote by m_i the size of layer L^i and let i, k be integers such that $1 \leq i < r$ and $1 \leq k \leq m_i$. Here, we denote the j -th row of W as W_j . We have,

$$\begin{aligned} \frac{\partial J}{\partial z_k^{[i]}}(z_k^{[i]}) &= \left(\frac{\partial J}{\partial a_k^{[i]}} \frac{\partial a_k^{[i]}}{\partial z_k^{[i]}} \right) (z_k^{[i]}) && \text{(chain rule)} \\ &= \left(\frac{\partial J}{\partial a_k^{[i]}} (\sigma^{[i]})' \right) (z_k^{[i]}) && \text{(by definition)} \\ &= \sum_{t=1}^{m_{i+1}} \left(\frac{\partial J}{\partial z_t^{[i+1]}} \frac{\partial z_t^{[i+1]}}{\partial a_k^{[i]}} (\sigma^{[i]})' \right) (z_k^{[i]}) && \text{(general chain rule (Lemma 4.6))} \\ &= \sum_{t=1}^{m_{i+1}} \delta_t^{[i+1]} \left(\frac{\partial (W^{[i+1]} a^{[i]} + b^{[i+1]})_t}{\partial a_k^{[i]}} (\sigma^{[i]})' \right) (z_k^{[i]}) && \text{(by definitions above)} \\ &= \sum_{t=1}^{m_{i+1}} \delta_t^{[i+1]} \left(\frac{\partial (W_t^{[i+1]} a^{[i]} + b_t^{[i+1]})}{\partial a_k^{[i]}} (\sigma^{[i]})' \right) (z_k^{[i]}) \\ &= \sum_{t=1}^{m_{i+1}} \delta_t^{[i+1]} w_{t,k}^{[i+1]} (\sigma^{[i]})'(z_k^{[i]}) \\ &= \left(W^{[i+1]T} \delta^{[i+1]} \right)_k (\sigma^{[i]})'(z_k^{[i]}). \end{aligned}$$

□

Proposition 4.8. For $1 \leq i \leq r$, we have

$$\frac{\partial J}{\partial W^{[i]}} = \delta^{[i]} a^{[i-1]T}.$$

Proof. For $1 \leq r \leq m_i, 1 \leq s \leq m_{i-1}$, we have

$$\frac{\partial J}{\partial w_{r,s}^{[i]}}(w_{r,s}^{[i]}) = \sum_{t=1}^{m_{i+1}} \left(\frac{\partial J}{\partial z_t^{[i]}} \frac{\partial z_t^{[i]}}{\partial w_{r,s}^{[i]}} \right) (w_{r,s}^{[i]}) \quad \text{(general chain rule)}$$

$$\begin{aligned}
 &= \sum_{t=1}^{m_{i+1}} \delta_t^{[i]} \left(\frac{\partial \sum_{u=1}^{m_i} w_{t,u}^{[i]} a_u^{[i]}}{\partial w_{r,s}^{[i]}} \right) (w_{r,s}^{[i]}) \\
 &= \delta_r^{[i]} a_s^{[i-1]} \qquad \left(\frac{\partial z_t^{[i]}}{\partial w_{r,s}^{[i]}} = 0 \text{ when } (r, s) \neq (t, u) \right).
 \end{aligned}$$

□

Proposition 4.9. For $1 \leq i \leq r$, we have

$$\frac{\partial J}{\partial b^{[i]}} = \delta^{[i]}.$$

Proof. For i, k such that $1 \leq i \leq r$ and $1 \leq k \leq m_i$, we have,

$$\frac{\partial J}{\partial b_k^{[i]}} (b_k^{[i]}) = \left(\frac{\partial J}{\partial z_k^{[i]}} \frac{\partial z_k^{[i]}}{\partial b_k^{[i]}} \right) (b_k^{[i]}) = \delta_k^{[i]}.$$

□

Algorithm 4.10. (Backpropagation) $\left\{ \begin{array}{l} \text{Todo: Write down the algorithm.} \end{array} \right\}$

4.3 Convolutional neural networks

$\left\{ \begin{array}{l} \text{Todo: Expand this section by giving an explicit example and a tensor flow implementation} \end{array} \right\}$
 Convolutional Neural Networks (CNNs) are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data (1D) and image data (2D). They have been highly successful in areas such as image recognition and classification.

4.3.1 Architecture

The architecture of a CNN is designed to take advantage of the 2D structure of an input image. This is achieved through the use of convolutional layers, pooling layers, and fully connected layers.

- **Convolutional Layers:** In these layers, a set of learnable filters are applied to the input image. Each filter is convolved across the width and height of the input image, computing the dot product between the entries of the filter and the input, producing a 2D activation map of that filter.
- **Pooling Layers:** These layers are used to reduce the spatial dimensions (width & height) of the input volume for the next convolutional layer. It is done to decrease the computational power required to process the data through dimensionality reduction. Furthermore, it is also useful for extracting dominant features which are rotational and positional invariant, thus providing the network with spatial invariance.
- **Fully Connected Layers:** After several convolutional and pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can thus be computed with a matrix multiplication followed by a bias offset.

4.3.2 Backpropagation in CNNs

Similar to fully connected networks, CNNs use backpropagation to train the network. However, due to the convolutional and pooling layers, the backpropagation algorithm requires some modifications.

- **Convolutional Layers:** The gradients with respect to the filter weights in a convolutional layer are computed by convolving the gradient of the loss function with respect to the output of the convolutional layer with the inputs to the convolutional layer.
- **Pooling Layers:** The pooling operation is non-linear and typically max-pooling is used. During backpropagation, the gradient is passed back only through the neuron which had the maximum value during the forward pass.

4.3.3 Advantages of CNNs

- **Parameter Sharing:** A feature learned in one part of an image can be applied to other parts of an image, reducing the number of parameters.
- **Local Connectivity:** Each neuron is connected only to a small region of the input image, which makes the network more robust and reduces the number of parameters.

In summary, CNNs leverage spatial hierarchies and patterns in data, making them highly efficient for tasks like image and video recognition, image classification, and many others where the input data has a spatial relationship.

5 Reinforcement Learning

Reinforcement learning is a type of machine learning in which an agent learns to interact with its environment in order to maximize a reward. In reinforcement learning, the agent receives feedback in the form of rewards for actions it takes within the environment. Over time, the agent learns to take actions that maximize its reward, thereby learning to behave optimally in the environment. There are several different algorithms used in reinforcement learning, including Q-learning, SARSA, and deep Q-network (DQN) learning.

Basic reinforcement learning is modeled as a Markov decision process (MDP). In an MDP, an agent makes a series of decisions based on its current state and receives a corresponding reward for each action it takes. The goal of the agent is to find a policy that maximizes its total reward over time. However, at any given time, the agent may not know the properties of each choice, and so it must act on incomplete information.

Definition 5.1. A Markov decision process (MDP) is a tuple (S, A, T, R) , where

(i) S is a set of states called the **state space**,

(ii) A is a set of actions called the **action space**,

(iii) T is a map

$$T : S \times A \times S \rightarrow [0, 1],$$

called the **transition probability function**,

(iv) R is a map

$$R : S \times A \times S \rightarrow \mathbb{R},$$

called the **reward function**.

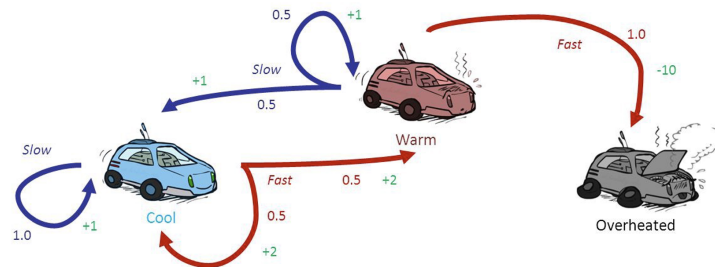
For fixed $s \in S$ and $a \in A$ the transition probability function gives a probability distribution over S , i.e. for $s' \in S$

$$T(s, a, s') = P(s' | s, a).$$

$T(s, a, s')$: The probability that one reaches state s' when taking action a in state s .

$R(s, a, s')$: The reward that one gets by going from state s to s' by doing action a .

Example 5.2. 👉 **Todo:** *Write out the car example* 👈



Dynamics of MDP:

- (i) Start at some state $s_0 \in S$.
 - (ii) Choose an action $a_0 \in A$.
 - (iii) Obtain a new state $s_1 \in S$ (with probability $T(s_0, a_0, s_1)$) and a reward $R(s_0, a_0, s_1)$.
 - (iv) Repeat until one reaches a terminal state or a fixed number of steps N .
- ($N = \infty$ possible)

Goal: Choose the actions a_0, a_1, a_2, \dots at each state such that

$$\sum_{j=0}^N R(s_j, a_j, s_{j+1})$$

gets big.

Problem: This could be an infinite sum and therefore one introduces a **discount factor** $\gamma \in [0, 1]$ and then considers the discounted total reward:

$$\sum_{j \geq 0} \gamma^j R(s_j, a_j, s_{j+1}).$$

Another interpretation: Immediate rewards count more than delayed rewards.

Definition 5.3. For given sequences of states (s_0, s_1, \dots) and actions (a_0, a_1, \dots) the **discounted total reward** (with discount $\gamma \in [0, 1]$) is given by

$$\sum_{j \geq 0} \gamma^j R(s_j, a_j, s_{j+1}).$$

Definition 5.4. (i) A **policy** is a function $\pi : S \rightarrow A$.

(ii) The **value of a policy** π at state $s \in S$ is defined by

$$V_\pi(s) = E \left[\sum_{j \geq 0} \gamma^j R(s_j, \pi(s_j), s_{j+1}) \mid s_0 = s \right]$$

$\{\text{ToDo: The expected value } E \text{ in the above formula needs more detailed explanation}\}$

Interpretation of $V_\pi(s)$: The expected (discounted) total reward when starting in state $s_0 = s$ by using the policy π to choose the action $a_t = \pi(s_t)$ in state s_t .

Remark 5.5. For a given policy π , we can calculate the values $V_\pi(s)$ explicitly as follows. Assume that there are n states $S = (s_1, \dots, s_n)$ and define the $n \times n$ matrix

$$M_\pi = (\delta_{i,j} - \gamma T(s_i, \pi(s_i), s_j))_{1 \leq i, j \leq n}.$$

Since $V_\pi(s_i) = \sum_{j=1}^n T(s_i, \pi(s_i), s_j) R(s_i, \pi(s_i), s_j) + \gamma \sum_{j=1}^n T(s_i, \pi(s_i), s_j) V_\pi(s_j)$ we see that $V_\pi = (V_\pi(s_1), \dots, V_\pi(s_n))^T$ is the solution of

$$M_\pi V_\pi = \begin{pmatrix} \sum_{j=1}^n T(s_1, \pi(s_1), s_j) R(s_1, \pi(s_1), s_j) \\ \vdots \\ \sum_{j=1}^n T(s_n, \pi(s_n), s_j) R(s_n, \pi(s_n), s_j) \end{pmatrix}.$$

A policy π^* is called **optimal** if it has maximal value for all states $s \in S$:

$$V_{\pi^*}(s) = \max_{\pi} V_\pi(s).$$

Example 5.6. $\{\text{ToDo: Explain the optimal policy for the car example (Example 5.2)}\}$

The **state-action value function** Q^* is defined for all $(s, a) \in S \times A$ as the expected (discounted) total reward for taking action $a \in A$ at state $s \in S$ following the optimal policy π^* :

$$Q^*(s, a) = \sum_{s' \in S} T(s, a, s') R(s, a, s') + \gamma \sum_{s' \in S} T(s, a, s') V_{\pi^*}(s')$$

Interpretation: This gives the expected best possible reward after choosing action a when in state s .

Having the state-action value function Q^* we can derive the optimal policy by

$$\pi^*(s) = \operatorname{argmax}_{a \in A} Q^*(s, a).$$

Idea: Find a good approximation Q for the function Q^* . Use this to define a policy by

$$\pi(s) = \operatorname{argmax}_{a \in A} Q(s, a).$$

Example 5.7. In the car example (Example 5.2) we have the states $S = \{C, W, O\}$ (cool, warm, overheated) and actions $A = \{f, s\}$ (fast, slow). Now we want to describe a way to obtain numerically an approximation Q of the state action value function. The function $Q : S \times A \rightarrow \mathbb{R}$ can be thought of as a table

	C (Cool)	W (Warm)	O (Overheated)
f (Fast)	$Q(C, f)$	$Q(W, f)$	$Q(O, f)$
s (Slow)	$Q(C, s)$	$Q(W, s)$	$Q(O, s)$

whose entries we want to find. First (Step 0) Choose starting values for Q . For example, random values as follows:

	C (Cool)	W (Warm)	O (Overheated)
f (Fast)	3	2	0
s (Slow)	1	4	0

After this we will proceed as follows.

Step 1: Choose a starting state, e.g. let us choose $s_0 = C$.

Step 2: Choose the action a_0 with the largest current value $Q(s_0, a_0)$. In our case $a_0 = f$, since $Q(C, f) = 3 > Q(C, s) = 1$.

(Since the values of our current table might be bad (or in order to explore undiscovered possibilities.. or just YOLO), a variant of Step 2 is to take a random action instead with a certain probability $\epsilon \in (0, 1)$. See Epsilon-Greedy Algorithm below).

Step 3: Take action $a_0 = f$ and receive a new state $s_1 \in S$ and reward $r_0 = R(s_0, a_0, s_1) \in \mathbb{R}$ from the environment. Let in our example assume that the environment gave back the state $s_1 = W$ (i.e. the car got warm). The reward we get is therefore $r_0 = R(s_0, a_0, s_1) = R(C, f, W) = 2$.

Step 4: Now we want to update $Q(s_0, a_0) = Q(C, f)$ based on what happened and what we know so far. In this example, the current value 3 was chosen randomly and does not make sense, but in later steps we should assume that the current value $Q(C, f)$ is the current best approximation we have. Therefore, we should also not completely disregard it. But we also just learned something new, i.e. that we get a reward 2 when taking action f in state C . Therefore, we want to update our value through

$$Q(C, f) = (1 - \alpha)Q(C, f) + \alpha(\text{something new...}),$$

where $\alpha \in [0, 1]$ is a **learning rate**, which indicates how much we take this new information into account. The (not good to use) extreme cases would be:

- $\alpha = 0$: We do not change the value $Q(C, f)$ at all.
- $\alpha = 1$: We ignore the old value and just use the new information.

The "something new" value above should be an approximation of $Q(C, f)$ just based on the information we just obtained, i.e. receiving $r_0 = 2$ and ending in state $s_1 = W$. A natural guess is to take 2 plus the maximal value we can obtain when we are in state W , which is 4. Since the 4 would come one step later, we also need to include the discount factor γ and get

$2 + \gamma \cdot 4$. In total, we get

$$\begin{aligned} Q(s_0, a_0) &= (1 - \alpha)Q(s_0, a_0) + \alpha(r_0 + \gamma \max_{a \in A} Q(s_1, a)) \\ &= (1 - \alpha)3 + \alpha(2 + \gamma 4). \end{aligned}$$

Let us assume in our example that $\alpha = \frac{1}{2}$ and $\gamma = \frac{3}{4}$. We therefore get as a new value for $Q(C, f)$:

$$Q(C, f) = \frac{3}{2} + \frac{1}{2}(2 + \frac{3}{4}4) = 4.$$

This leads to the new table

	<i>C (Cool)</i>	<i>W (Warm)</i>	<i>O (Overheated)</i>
<i>f (Fast)</i>	4	2	0
<i>s (Slow)</i>	1	4	0

From here on we can repeat Step 2 by assuming we are in state $s_1 = W$, i.e. we find a_1 such that $Q(s_1, a_1)$ is maximal (which is s), then take action $a_1 = s$ in Step 3, etc.

The following algorithm finds for all $s \in S$ and $a \in A$ a function $Q(s, a)$, which gives a good approximation for $Q^*(a, s)$.

Algorithm 5.8 (Q-learning algorithm). Start with random values for $Q(s, a)$ for all $s \in S$ and $a \in A$. (e.g. all zero).

One **episode** of the Q-learning algorithm is given as follows:

1. Choose a starting state $s_0 \in S$.
2. Look up the current best action in that state, i.e. $a_0 = \operatorname{argmax}_{a \in A} Q(s_0, a)$
or (with probability $\epsilon \in [0, 1]$) choose a random action $a_0 \in A$ (*Epsilon-Greedy Algorithm*).
3. Apply this action and get a new state s_1 and reward $r_0 = R(s_0, a_0, s_1)$.
4. Update the value $Q(s_0, a_0)$ as follows (**Bellman equation**)

$$Q(s_0, a_0) = (1 - \alpha)Q(s_0, a_0) + \alpha \left(r_0 + \gamma \max_{a \in A} Q(s_1, a) \right).$$

Here $\alpha \in [0, 1]$ is the **learning rate**.

5. If s_1 is not a terminal state repeat with step 2.

References

- [N] A. Ng: *CS229 Lecture Notes*. (available at https://cs229.stanford.edu/notes2022fall/main_notes.pdf)