Machine Learning and Its Safety and Reliability Issues

Xiaowei Huang
Gaojie Jin

6th October 2021

Contents

1 Chapter I: Foundations

As the starting point of this book, we introduce fundamental knowledge that will be useful for the later technical contents. After explaining basic concepts in machine learning, we will introduce probability theory and linear algebra, two mathematical foundations among many others for machine learning.

1.1 Machine Learning Basics

What is machine learning? Every machine learning algorithm is a software program that can improve its performance by learning from data (or examples). As shown in Figure 1.1, given a program (or model) f, the improvement (from f to f') is achieved by applying a learning algorithm on f and a set D of training instances.

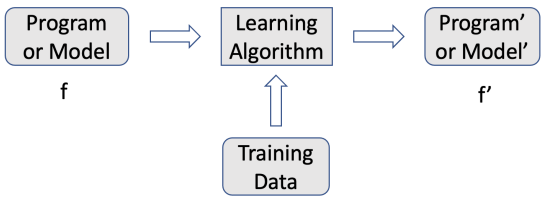


Figure 1.1: Machine learning

Usually, a learning algorithm cannot fully comprehend the data with a single pass. Therefore, the learning over a dataset D is an iterative process, i.e., the learning step shown in Figure 1.1 is repeated, until a termination condition is satisfied. A termination condition can be e.g., a number of iterations, an accuracy threshold, a convergence condition, etc.

Based on the above process and depending on the nature of the data and how the learning algorithm interacts with the data, there can be different learning tasks and learning schemes. We will briefly discuss the basic categories of them in Section 1.1.4 and Section 1.1.5, leaving the details of the learning algorithms to Chapter 3 and Chapter 4. In the following, we explain the data representation (Section 1.1.1) and the datasets (Section 1.1.2).

1.1.1 Data Representation

Data representation refers to the way how the data instances are stored and represented. A suitable data representation can ease the storage, transmission, and processing of data, and more importantly, benefit the machine learning algorithms which heavily rely on not only the data but also the way how the data is operated.

Representing instances using feature vectors A dataset is formed of a finite set of data instances as well as their associated labels if available. One common way to represent a data instance is to use a fixed-length vector $\mathbf{x}$ to represent features (or attributes) of the data instance. Standard feature types include e.g.,

- nominal (including Boolean) type, such that there is no ordering among possible values of the feature. For example, $color \in \{red, blue, green\}$ is a nominal feature.
- ordinal type, such that possible values of the feature are totally ordered. For example, $size \in \{small, medium, large\}$ is an ordinal type.
- numeric (continuous) type, whose values are stored as groupings of bits, such as bytes and words. Numbers, such as integers and real numbers, are typical examples of numeric type. As an example, $weight \in [0...500]$ is a numeric type.

Example 1. *For the **iris** dataset [3], we have the following. The left picture is an illustration of an iris flower, where we can see the intuitive meanings of four features: sepal length, sepal width, petal length, petal width. The right table is a snapshot of the dataset, which has in total 150 instances. We can see that, all four features are represented as numeric values.*

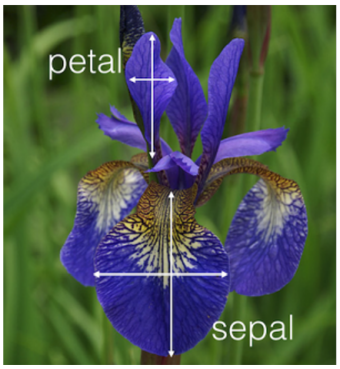


Table 1.1: An iris flower

index	Sepal Length	Sepal Width	Petal Length	Petal Width	Class Label
1	5.1	3.5	1.4	0.2	iris setosa
2	4.9	3.0	1.4	0.2	iris setosa
...					
50	6.4	3.5	4.5	1.2	iris versicolor
...					
150	5.9	3.0	5.1	1.8	iris virginica

Table 1.2: Iris dataset

We may write $\mathbf{x} = (x_1, ..., x_n)$ for an instance with n features, each of which has value x_i for $i \in \{1, ..., n\}$. Then, in case we are dealing with labelled data, we also need to represent the label of each instance $\mathbf{x}$. Depending on the nature of the problem, a label can be represented as either a scalar y (e.g., classification) or a vector $\mathbf{y}$ (e.g., object detection). There are also tasks in which each label is a structured object.

Example 2. *For the **iris** example, we can see from Table 1.2 that each instance is associated with a label indicating it is in one class (3 classes in total). If we use 1 for iris setosa, 2 for iris versicoler, and 3 for iris virginica, we may have $\boldsymbol{x}_1 = (5.1, 3.5, 1.4, 0.2)$, and $y_1 = 1$.*

Example 3. *In the* object detection *task, label is a set of bounding boxes, each of which is associated with not only a classification but also other information such as the size of box, the coordinates of the box in the image, etc.*

In the following, we may also write $(\mathbf{x}, y)$ for an instance, assuming that the label is a scalar number y.

Feature space We can think of each instance $\mathbf{x}$ as representing a point in a n-dimensional feature space where n is the number of features of $\mathbf{x}$.

Example 4. *Assume that we are working with a dataset which are sampled from the following underlying function:*

$$\begin{aligned} X &\in [0, 7] \\ Y &= \sin 7X + \epsilon \\ Z &= \cos 7X + \epsilon \\ \epsilon &\in [0, 1] \end{aligned} \tag{1.1}$$

such that all instances contain 3 numerical features X, Y, Z. ϵ denotes the noise. Each data instance is a point in a 3-dimensional space. The visualisation of the dataset is seen in Figure 1.2. The code for the generation of the figure is given in Section 1.4.

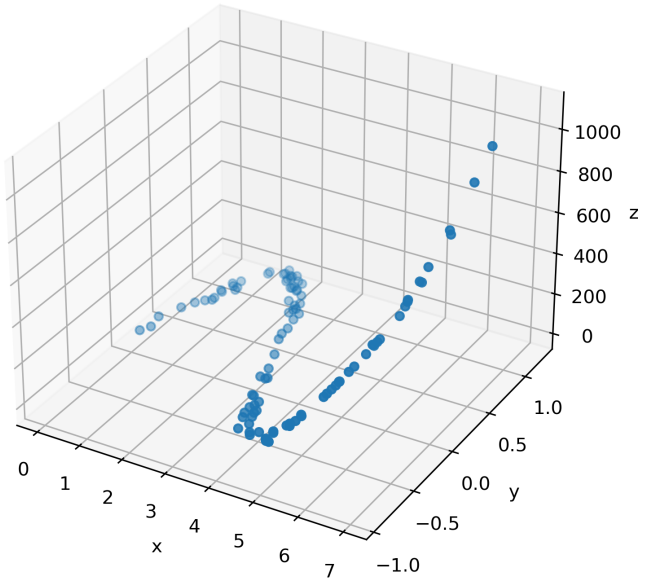


Figure 1.2: Visualisation of a 3-dimensional dataset

1.1.2 Datasets

A dataset is a collection of data instances. According to their different roles in the machine learning lifecycle, we may have multiple datasets.

i.i.d. assumption We often assume that training instances are independent and identically distributed (i.i.d.), i.e., the instances are sampled from the same probability distribution (i.e., identically distributed) and all of them are mutually independent. i.i.d. assumption has an important property for a valid dataset, because it enables the following property:

the larger the size of dataset becomes, the greater the probability of the data instances will closely resemble the underlying distribution.

We remark that, this property is key to many machine learning theoretical results, which are based on e.g., Chebyshev's inequality, Hoeffding's lemma, etc.

However, there are also cases where this assumption does not hold, such as

- instances sampled from the same medical image,
- instances from time series,
- etc.

For non-i.i.d. dataset, dedicated techniques have to be taken to make sure the learning algorithm can learn useful information instead of biased information.

Training, test, and validation datasets The collected dataset can be split into two datasets, one for training and the other for test. We call them *training dataset* and *test dataset*, respectively. The split does not have a fixed percentage, with e.g., 7:3 or 8:2 split being very commonly seen in practice. Within the training dataset, there might be a subset called *validation dataset* that is often used to control the learning process, such as the termination of the learning process or the selection of the learning directions. Figure 1.3 presents an illustrative diagram for the three datasets.

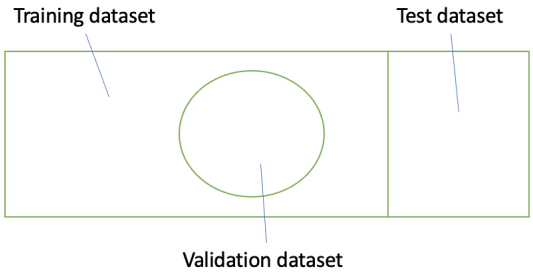


Figure 1.3: Training, test, and validation datasets

1.1.3 Hypothesis space and inductive bias

As suggested in Figure 1.1, a learning agent f is a program or model or function. It updates itself by learning from training data. Nevertheless, the function f has to be chosen from a function space $\mathcal{H}$, called hypothesis space. Normally, the hypothesis space is determined by the learning algorithm.

Example 5. *For a decision tree algorithm, the hypothesis space is the set of all possible trees such that each tree node represents a feature and the branches of a tree node represent the split of the possible feature values.*

Example 6. *For a linear regression algorithm, the hypothesis space is the set of linear functions $y = \boldsymbol{w}^T \boldsymbol{x} + b$, where $\boldsymbol{w} \in \mathbb{R}^{|\boldsymbol{x}|}$ and $b \in \mathbb{R}$.*

Example 7. *For a neural network whose corresponding function $f_{\boldsymbol{W}}$ is parameterised over learnable weights $\boldsymbol{W}$, the hypothesis space is the set of functions $f_{\boldsymbol{W}}$ such that each weight in $\boldsymbol{W}$ is a real number.*

The inductive bias (also known as learning bias) of a learning algorithm is the set of assumptions that the designer imposes on the hypotheses in $\mathcal{H}$ to guide the learner in its learning. Usually, the assumptions can be e.g.,

- restrictive assumption that limits the hypothesis space, or
- preference assumption that imposes ordering on hypothesis space, etc.

Example 8. *For decision tree learning, it is possible to ask for a preference over simpler trees, by following the Occam's razor.*

Example 9. *For neural networks, it is possible to apply regularisation techniques, such as L_1 or L_2 regularisation, so that the learning algorithm will have preference between weight matrices $\boldsymbol{W}$ embedded.*

1.1.4 Learning tasks

Different machine learning algorithms (and probably datasets) will be needed, according to different learning tasks.

Supervised learning One of the most frequently seen task is supervised learning, which learns a function according to a set of input-output pairs. The primary objective in supervised learning is to make sure that the learned function generalises, i.e., it is able to accurately predict label y for previously unseen $\mathbf{x}$.

Definition 1. *(Supervised Learning) Given a training set of instances sampled from an unknown target function h, i.e.,*

$$D = \{(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), ..., (\boldsymbol{x}^{(m)}, y^{(m)})\} \tag{1.2}$$

it is to learn a function $f \in \mathcal{H}$ that approximates the target function h, where $\mathcal{H}$ is a set of models (a.k.a. hypotheses). We call D a labelled dataset when each instance $\boldsymbol{x}$ is attached with a label y, and unlabelled, otherwise.

In the above definition, when y is discrete, it is a *classification* task (or concept learning). When y is continuous, it is a *regression* task. The function f is called classifier or regressor, depending on the tasks. For a classifier, it is often that, instead of directly returning a label, $f(\mathbf{x})$ returns a probabilistic distribution over the set C of labels such that

$$\sum_{c \in C} f(\mathbf{x})(c) = 1 \tag{1.3}$$

and we let its label be the one with maximum probability, i.e.,

$$\hat{y} = \arg\max_{c \in C} f(\mathbf{x})(c) \tag{1.4}$$

Note that, a predictive label $\hat{y}$ may be different from the ground truth label y.

Unsupervised learning Another popular learning task is unsupervised learning, which, instead of asking users to supervise the learning with example input-output pairs, asks the learning algorithm to discover patterns and information that was previously undetected. Formally,

Definition 2. *(Unsupervised Learning) Given a training set of instances without y's, i.e., an unlabelled dataset*

$$D = \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(m)}\} \tag{1.5}$$

it is to discover interesting regularities (such as structures and patterns) that characterize the instances.

Concretely, depending on the "interesting regularities", there are a few unsupervised learning tasks, i.e.,

- *clustering*, which is to find model $f \in \mathcal{H}$ that divides the training set into clusters such that the clusters satisfy certain intra-cluster similarity and inter-cluster dissimilarity.
- *anomaly detection*, which is to learn model $f \in \mathcal{H}$ that represents "normal" instances, so that the model can later be used to determine whether a new data x looks normal or anomalous.
- *dimensionality reduction*, which is to find model $f \in \mathcal{H}$ that represents each instance $\mathbf{x}$ with a lower-dimension feature vector while still preserving key properties of $\mathbf{x}$.

Semi-supervised learning In addition to the supervised and unsupervised learning, there are other learning tasks such as *semi-supervised learning*, which enables the learning to be proceeded with a smaller labelled dataset D_1 and a larger unlabelled dataset D_2. This becomes ever more important because we have more and more data but it is known that the labelling is usually done by human operators and is very costly.

1.1.5 Learning Schemes

Learning schemes determine how the learning is conducted. We have mainly two categories:

- *batch learning*, with which the learner is given the training dataset as a batch (i.e. all at once), and
- *online learning*, with which the learner receives the data instances sequentially, and updates the model after processing each new batch of data.

Moreover, we note the existence of a distinction between active and passive learning. For the former, it is generally believed that the learner has some role in determining on what data it will be trained. However, for the former, the learner is simply presented with a training dataset.

1.1.6 Density Estimation

Density estimation is to construct an estimation of the underlying function that generates the dataset D. However, an accurate estimation of the full distribution is computationally hard, and it might be sufficient to know, among a (possibly infinite) set of models, which model can lead to the best possibility of generating the dataset D. It is often that the set of models are parameterised over a set θ of random variables, such as the means and variance of a Gaussian distribution.

Maximum Likelihood Estimation (MLE) is a frequentist method. It is estimate the best model parameters θ that maximise the probability of observing the data from the joint probability distribution. Formally,

$$\theta_{MLE} = \arg\max_{\theta} P(D|\theta) \tag{1.6}$$

This resulting conditional probability $P(D|\theta_{MLE})$ is referred to as the likelihood of observing the data given the model parameters. Note that,

$$\arg\max_{\theta} P(D|\theta) = \arg\max_{\theta} \prod_{(\mathbf{x},y)\in D} P(\mathbf{x}|\theta) \tag{1.7}$$

Considering that the product of probabilities (between 0 to 1) is not numerically stable, we add the log term, i.e.,

$$\begin{aligned} \theta_{MLE} &= \arg\max_{\theta} \log P(D|\theta) \\ &= \arg\max_{\theta} \log \prod_{(\mathbf{x},y)\in D} P(\mathbf{x}|\theta) \\ &= \arg\max_{\theta} \sum_{(\mathbf{x},y)\in D} \log P(\mathbf{x}|\theta) \end{aligned} \tag{1.8}$$

Maximum a posteriori (MAP) queries is a Bayesian method. It to estimate the best model parameters that best explain an observed dataset. MAP query is also called MPE (Most Probable Explanation), as each setting can be seen as an explanation and MAP is to compute the most likely explanation. Formally,

$$\theta_{MAP} = \arg\max_{\theta} P(\theta|D) = \arg\max_{\theta} P(D|\theta)P(\theta) \tag{1.9}$$

As we can see that, the MAP computation involves the calculation of a conditional probability of observing the data given a model, weighted by a prior probability or belief about the model. The only difference between MLE and MAP is on the prior distribution $P(\theta)$, and if $P(\theta)$ is uniform, they are exactly the same.

Similar as MLE, we may add the log term and make Equation (1.9) into:

$$\begin{aligned} \theta_{MAP} &= \arg\max_{\theta} \log P(D|\theta) + \log P(\theta) \\ &= \arg\max_{\theta} \sum_{(\mathbf{x},y)\in D} \log P(\mathbf{x}|\theta) + \log P(\theta) \end{aligned} \tag{1.10}$$

This expression suggests that MAP can be seen as the MLE with a regularisation term.

Expected A Posteriori (EAP) By Equation (1.9), we know that θ_{MLE} is to compute the mode of the posterior distribution. This might not reflect the distributional information we need, and we may be interested in e.g., the expected value of θ under D.

$$\mathbb{E}(\theta|D) = \int_{\theta} \theta P(\theta|D) d\theta \tag{1.11}$$

1.2 Foundation of Probability Theory

1.2.1 Random Variables

In most of our contexts, a random variable is a function that assigns probability values to each of an experiment's (or an event's) outcomes. Intuitively, a random variable has a probability distribution, which represents the likelihood that any of the possible values would occur.

Example 10. *We have a population of students, such that we want to reason about their grades. In this case, we have*

- *a random variable $Grade$, with the set of possible values $V(Grade) = \{A, B, C\}$, and*
- *a function $P(Grade)$, which associates a probability with each outcome.*

Given $P(X)$ is a probability distribution, we have

$$\sum_{x \in V(X)} P(x) = 1 \tag{1.12}$$

and we may call $P(X)$ the marginal distribution of X, as opposed to the joint and conditional distributions. We will explain these concepts in detail later.

A probability distribution $P(X)$ is a multi-nominal distribution if there are multiple values for the random variable X, i.e., $|V(X)| > 1$. Moreover, if $V(X) = \{false, true\}$ then $P(X)$ is a Bernoulli distribution. Besides, $P(X)$ can be continuous, such that it may take on an uncountable set of values.

Example 11. *The $Grade$ random variable in Example 10 has three possible values and therefore it is associated with a multi-nominal distribution. Also, for a coin tossing example with a single coin that may be either head or tail, the random variable $Coin$ is associated with a Bernoulli distribution. Moreover, a real-valued random variable is continuous, even if it only takes values from a real interval.*

A random variable can also be multi-dimensional.

Example 12. *For the **iris** example as in Example 1, the data instance can be seen as a 4-dimensional continuous variable X, such that it has an underlying data distribution and the dataset is sampled from the data distribution. The label Y can be seen as another random variable.*

1.2.2 Joint and Conditional Distributions

A marginal probability is the probability of an event X occurring, i.e., $P(X)$. It may be thought of as an unconditional probability, as it is not conditioned on any other event.

Example 13. *Assume that we randomly draw a card from a standard deck of playing cards. Let C be the random variable, representing the color of the card drawn. Then, the probability that the card drawn is red is $P(C = red) = 0.5$, or simply $P(red) = 0.5$ if the random variable C is clear from the context.*

Example 14. *Given a standard deck of playing cards, the probability that a card drawn is a 4 is $P(four) = 1/13$.*

Joint probability $P(X, Y)$ is the probability of event X and event Y occurring. It is a statistical measure that calculates the likelihood of two events occurring at the same time.

Example 15. *Given a standard deck of playing cards, the probability that a card is a four and red, i.e.,* $P(four, red) = 2/52 = 1/26$. *Note: there are two red fours in a deck of 52, the 4 of hearts and the 4 of diamonds.*

The joint probability can be generalised to work with more than two events.

Conditional probability $P(X|Y)$ is the probability of event X occurring, given that event Y has already occurred.

Example 16. *Given a standard deck of playing cards, if we know that you have drawn a red card, what is the probability that it is a four? The answer is* $P(four|red) = 2/26 = 1/13$, *i.e., out of the 26 red cards (given a red card), there are two fours, so* $2/26 = 1/13$.

1.2.3 Independence and Conditional Independence

Without loss of generality, we assume that for the remaining of this section, all random variables are Boolean. Given two Boolean random variables X and Y, we expect that, in general, $P(X|Y)$ is different from $P(X)$, i.e., the fact that Y is true may change our probability over X.

Independence Sometimes an equality can occur, i.e, $P(X|Y) = P(X)$. i.e., learning that Y occurs does not change our probability of X. In this case, we say event X is independent of event Y, denoted as

$$X \perp Y \tag{1.13}$$

A distribution P satisfies $X \perp Y$ if and only if $P(X, Y) = P(X)P(Y)$.

Example 17. *Consider the joint probability table as shown in Table 1.3.*

X	Y	P(X,Y)
0	0	0.08
0	1	0.32
1	0	0.12
1	1	0.48

Table 1.3: A simple two variable joint distribution

First of all, we note that

$$\begin{array}{rcl} P(X=0) & = & 0.32 + 0.8 = 0.4 \\ P(X=1) & = & 0.6 \\ P(Y=0) & = & 0.08 + 0.12 = 0.2 \\ P(Y=1) & = & 0.8 \end{array} \tag{1.14}$$

Then, we notice that

$$\begin{array}{rcccccl} 0.08 & = & P(X=0,Y=0) & = & P(X=0)*P(Y=0) & = & 0.4*0.2 \\ 0.32 & = & P(X=0,Y=1) & = & P(X=0)*P(Y=1) & = & 0.4*0.8 \\ 0.12 & = & P(X=1,Y=0) & = & P(X=1)*P(Y=0) & = & 0.6*0.2 \\ 0.48 & = & P(X=1,Y=1) & = & P(X=1)*P(Y=1) & = & 0.6*0.8 \end{array} \tag{1.15}$$

that is,

$$P(X,Y) = P(X)*P(Y) \tag{1.16}$$

which suggests that X and Y are independent.

Example 18. *On the other hand, for the following Table 1.4, the two variables X and Y are not independent.*

X	Y	P(X,Y)
0	0	0.10
0	1	0.16
1	0	0.64
1	1	0.10

Table 1.4: Another simple two variable joint distribution

Conditional independence While independence is a useful property, we do not often encounter two independent events. A more common situation is when two events X and Y are independent given an additional event X, denoted as

$$X \perp Y|Z \tag{1.17}$$

A distribution P satisfies $X \perp Y|Z$ if and only if $P(X,Y|Z) = P(X|Z)P(Y|Z)$.

Note that, similar calculation as in Examples 17 and 18 can be done to work with conditional independence, with the only changes on replacing the computation of marginal probabilities $P(X)$ and $P(Y)$ with the computation of conditional probabilities $P(X|Z)$ and $P(Y|Z)$.

1.2.4 Querying Joint Probability Distributions

A joint distribution $P(X_1, ..., X_n)$ contains an exponential number 2^n of real probability values and it can be hard to make sense of them. It is desirable that we are able to infer useful information by making queries. In the following, we introduce a few categories of queries.

Probability queries are to compute distribution of a subset of random variables, given evidence (i.e., the values) of another subset of random variables. Formally, it is to compute

$$P(X_{i1}, ..., X_{ik}|X_{j1} = x_{j1}, ..., X_{jl} = x_{jl}) \tag{1.18}$$

where $x_{j1}, ..., x_{jl}$ are evidence for the random variables $X_{j1}, ..., X_{jl}$, respectively. Moreover, we have $\{X_{i1}, ..., X_{ik}\} \cup \{X_{j1}, ..., X_{jl}\} \subseteq \{X_1, ..., X_n\}$ and $\{X_{i1}, ..., X_{ik}\} \cap \{X_{j1}, ..., X_{jl}\} = \emptyset$.

Maximum a posteriori (MAP) queries In addition to probability queries which concern the (conditional) probability of the occurrence of events, we may be interested in MAP-style queries, which is to find a joint assignment to some subset of variables that has the highest probability. For simplicity, we let $\{X_1, ..., X_k\}, \{X_{k+1}, ..., X_l\}, \{X_{l+1}, ..., X_n\}$ be three disjoint subsets of the random variables $\{X_1, ..., X_n\}$. Then, the MAP query is of the form

$$\begin{aligned} & MAP(X_1, ..., X_k | X_{l+1}, ..., X_n) \\ = & \underset{x_1,...,x_k}{\arg\max} \sum_{x_{k+1},...,x_l} P(X_1 = x_1, ..., X_l = x_l | X_{l+1} = x_{l+1}, ..., X_n = x_n) \end{aligned} \tag{1.19}$$

Intuitively, we have the evidence for variables $\{X_{l+1}, ..., X_n\}$, and intend to find the joint assignment for $\{X_1, ..., X_k\}$. Because we do not have information about $\{X_{k+1}, ..., X_l\}$, we marginalise them.

Example 19. *Consider a probability table for variable X_1 as in Table 1.5.*

$X_1 = 0$	$X_1 = 1$
0.4	0.6

Table 1.5: Probability table for X_1

Then, we have $MAP(X_1) = 1$.

Example 20. *Consider a joint probability table for variables X_1 and X_2 as in Table 1.6.*

X_1	X_2	$P(X_1, X_2)$
0	0	0.04
0	1	0.36
1	0	0.3
1	1	0.3

Table 1.6: Joint probability table for X_1 and X_2

Then, we have $MAP(X_2) = 1$.

1.3 Linear algebra

1.3.1 Scalars, Vectors, Matrices, Tensors

Scalar A scalar is a single number. It is represented in lower-case italic, such as x.

Example 21. *We can use $x \in \mathbb{R}$, a real-valued scalar, to define the slope of the line. Moreover, we can use $x \in \mathbb{N}$, a natural number scalar, to define the number of units.*

Vector A vector is an array of numbers arranged in order. Each number in a vector is identified with an index. Vectors are represented as lower-case bold, such as $\mathbf{x}$. If x is n-dimensional and each element of $\mathbf{x}$ is a real number, we write $\mathbf{x} \in \mathbb{R}^n$.

Geometrically, we think of vectors as points in space such that each element of a vector gives coordinate along an axis. This is the same as we consider a data instance – usually represented as a vector – as a point in high-dimensional space.

Matrix A matrix is a 2-D array of numbers, such that each element is identified by two indices. Matrices are represented as bold typeface $\mathbf{A}$. We usually identify each element of $\mathbf{A}$ with the subscripts, i.e., use $\mathbf{A}_{i,j}$ to denote the element on the i-th row and j-th column. We may also write $\mathbf{A}[i:]$ for the i-th row of $\mathbf{A}$ and $\mathbf{A}[:j]$ the j-th column of $\mathbf{A}$. Similar as the vectors, we may write $\mathbf{A} \in \mathbb{R}^{m \times n}$ if $\mathbf{A}$ has m rows and n columns and each element is a real number.

Tensor It is very likely in a machine learning context that, we may need an array with more than two axes. We call a multi-dimensional array arranged on a regular grid with variable number of axes as a tensor. Tensors are also represented as bold typeface $\mathbf{A}$, and we may use the subscripts to identify its element, for example $\mathbf{A}_{i,j,k}$ to denote the element in a three-dimensional tensor. We remark that, a tensor may include more involved properties that a multi-dimensional array does not have, but we omit the details for the simplicity of the explanations.

1.3.2 Matrix Operations

We briefly review a few frequently used matrix operations.

Transpose of a Matrix The transpose of a matrix is an operator which flips a matrix over its diagonal. Given a matrix $\mathbf{A}$, we define its transpose $\mathbf{A}$ as that

$$\mathbf{A}^T_{ij} = \mathbf{A}_{ji} \text{ for all } i, j \tag{1.20}$$

Linear Transformation A linear transformation is a function from one vector space to another that respects the underlying (linear) structure of each vector space. Linear transformations can be represented by matrices. If T is a linear transformation mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$, and $\mathbf{x}$ is a column vector with n entries, then

$$T(\mathbf{x}) = \mathbf{A}\mathbf{x} \tag{1.21}$$

for some $m \times n$ matrix $\mathbf{A}$. Also, $\mathbf{A}$ is called transformation matrix of T.

Identity Matrix The identity matrix of size n is the $n \times n$ square matrix with ones on the diagonal and zeros elsewhere. We use $\mathbf{I}$ to denote an identity matrix.

Matrix Inverse An $n \times n$ matrix $\mathbf{A}$ is invertible if there exists another $n \times n$ matrix $\mathbf{A}$ such that

$$\mathbf{AB} = \mathbf{BA} = \mathbf{I} \tag{1.22}$$

We write $\mathbf{B}$ as the inverse of $\mathbf{A}$, denoted as $\mathbf{A}^{-1} = \mathbf{B}$,

1.3.3 Norms

Usually, a distance function is employed to compare data instances. Ideally, such a distance should reflect perceptual similarity between data instances, comparable to e.g., human perception for image classification networks. A distance metric should satisfy a few axioms which are usually needed for defining a metric space:

- $||\mathbf{x}|| \geq 0$ (non-negativity),
- $||\mathbf{x} - \mathbf{y}|| = 0$ implies that $\mathbf{x} = \mathbf{y}$ (identity of indiscernibles),
- $||\mathbf{x} - \mathbf{y}|| = ||\mathbf{y} - \mathbf{x}||$ (symmetry),
- $||\mathbf{x} - \mathbf{y}|| + ||\mathbf{y} - \mathbf{z}|| \geq ||\mathbf{x} - \mathbf{z}||$ (triangle inequality).

In practise, L_p-norm distances are used, including

- L_1 (Manhattan distance):

$$||\mathbf{x}||_1 = \sum_{i=1}^{n} |x_i| \tag{1.23}$$

- L_2 (Euclidean distance):

$$||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{n} x_i^2} \tag{1.24}$$

- L_∞ (Chebyshev distance):

$$||\mathbf{x}||_\infty = \max_i |x_i| \tag{1.25}$$

Moreover, we also consider L_0-norm as $||\mathbf{x}||_0 = |\{x_i \mid x_i \neq 0, i = 1..n\}|$, i.e., the number of non-zero elements. Note that, L_0-norm does not satisfy the triangle inequality. In addition to these, there exist other distance metrics such as Fréchet Inception Distance [5].

Given a data instance x and a distance metric L_p, the *neighbourhood* of $\mathbf{x}$ is defined as follows.

Definition 3 (d-Neighbourhood)**.** *Given a data instance $\boldsymbol{x}$, a distance function L_p, and a distance d, we define the* d-neighbourhood $\eta(\boldsymbol{x}, L_p, d)$ *of $\boldsymbol{x}$ w.r.t. L_p as*

$$\eta(\boldsymbol{x}, L_p, d) = \{\hat{\boldsymbol{x}} \mid ||\hat{\boldsymbol{x}} - \boldsymbol{x}||_p \leq d\}, \tag{1.26}$$

the set of data instances whose distance to $\boldsymbol{x}$ is no greater than d with respect to L_p.

1.4 Practicals

Hardware/software environment Your machine needs to have Python3 or Anaconda installed. To install the python packages, you have the following two options:

Using Pip for Installing First make sure you have installed python and pip (or pip3) (check through 'pip -V' in Windows Commands (cmd) or MacOS Terminal), then install the software packages:

```
$ pip (or pip3) install matplotlib
```

Create Conda Virtual Environment You can download the Anaconda through https://www.anaconda.com/products/individual, create an environment and install some necessary software packages:

```
$ conda create --name aisafety
$ conda activate aisafety
$ conda install pandas numpy matplotlib tensorflow scikit-learn pytorch torchvision
```

You could install Jupyter notebook through https://jupyter.org/install.

Test Installation Once installed, you can check whether the installation is successful by running the following commands. Note that, the first command is to activate the Python environment, in which the remaining commands are executed.

```
$ python3
$ import numpy
$ import scipy
$ import sklearn
$ import matplotlib
$ import pandas
$ exit()
```

Once the installation is successful, create a new file **lab1.py** and type in the following lines:

```
import numpy as np

x = np.arange(100)
y = np.array(5)

z=x+y
```

You need to make sure to understand the meaning of the command. Then, you can the following command to check the result:

```
$ python (or python3) lab1.py
```

1.4.1 Visualising a synthetic dataset

Below is the code for the visualisation of the synthetic dataset as given in Example 4.

```
import numpy as np
import matplotlib.pyplot as plt

xdata = 7 * np.random.random(100)
ydata = np.sin(xdata) + 0.25 * np.random.random(100)
zdata = np.exp(xdata) + 0.25 * np.random.random(100)

fig = plt.figure(figsize=(9, 6))
# Create 3D container
ax = plt.axes(projection = '3d')
# Visualize 3D scatter plot
ax.scatter3D(xdata, ydata, zdata)
# Give labels
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
# Save figure
plt.savefig('3d_scatter.png', dpi = 300);
```

To run this code (in the file "1.4.1.txt"), you need to activate the environment you have created earlier, with the following command:

```
$ conda activate aisafety
$ python3 1.4.1.txt
```

This will be needed for all future practicals.

1.4.2 Basic Python Operations

In the following, we provide a few exercises. Please complete them sequentially.

1. Using array indexing to give the last ten values of z

```
xLastTen = x[90:] # Or x[-10:]
```

2. Update the code so that x goes from 0 - 1000 in steps of 10

```
xUpdate = np.arange(0, 1000, 10)
```

3. Take dot product between x and x

```
xDotProduct = x.dot(x)
```

4. Take (*) product between x and x

```
xAsteriskProduct = x * x
```

5. Reshape x so that it is no longer a 100 value array but a 10x10 matrix

```
xReshape = xUpdate.reshape((10, 10))
```

6. Multiply the first row by 1, the second by 2, the third by 3 and so on

```
yNew = np.arange(1,11)
zNew = xReshape * yNew[:, np.newaxis]
print(zNew)
```

7. Using the matplotlib library to plot each row of this matrix as a single series on the same graph

```
for i in range(10):
   plt.plot(zNew[i])
plt.show()
```

8. Using the matplotlib library to plot each row of this matrix as a single series on separate sub-plots of the same figure and save this figure as figure1.png

```
for i in range(10):
    ax = plt.subplot(5, 2, i + 1)
    plt.plot(zNew[i])
plt.savefig('figure1.png')
plt.show()
```

1.5 Exercises

1. Give an example for a supervised learning problem, an unsupervised learning problem and a semi-supervised learning problem;
2. Give an example of a supervised learning problem that is a classification task;
3. Give an example of a supervised learning problem that is a regression task;
4. Give an example of a clustering problem;
5. For all the above problems, figure out the features and labels for them;
6. Please write a program to output the following information:
   a) How many samples are in the **iris** dataset;
   b) How many features are given for each sample in the **iris** dataset?
   c) What is the value range for each feature?
7. According to Table 1.7 about two random variables $Intelligence$ and $Grade$, please

	Intelligence = Low	Intelligence = High
Grade = A	0.07	0.18
Grade = B	0.28	0.09
Grade = C	0.3	0.08

Table 1.7: Joint probability for student grade and intelligence

compute the values $P(Grade = B \mid Intelligence = Low)$ and $MAP(Grade)$.

2 Chapter 2: Evaluation of Machine Learning Models

Once a machine learning model is trained, it is desirable to understand if it performs well enough in terms of some pre-specified properties. These properties are defined according to the requirements specified by the designer or the user of the machine learning model. Therefore, they might be problem specific. Nevertheless, there are typical evaluation methods that are frequently adopted by machine learning practitioners and researchers.

In this chapter, we review a set of evaluation methods. Before introducing concrete evaluation methods, we explain the ground truth of a learning problem and the underlying data distribution. Without loss of generality, in this chapter, we consider classification task.

2.1 Ground Truth and Underlying Data Distribution

In principle, when dealing with a classification task where the data instances have m real-valued features, a machine learning model f is to approximate a target, ground truth function h. That is, the ultimate objective of an evaluation is to understand the gap between $f(\mathbf{x})$ and $h(\mathbf{x})$ for all legitimate instances $\mathbf{x} \in \mathbb{R}^m$. Unfortunately, the input domain $\mathbb{R}^m$ is continuous and may contain an uncountable number of instances. It is unlikely that an evaluation method can exhaustively enumerate all possible instances. Therefore, an evaluation method may need to strike a balance between the exhaustiveness and the cost.

For a problem with m features, we let $X_1, ..., X_m$ be the m random variables, each of which corresponds to one of the features. Let

$$P_h(X_1, ..., X_m) \tag{2.1}$$

be the underlying data distribution of h, such that each instance $\mathbf{x} = (x_1, ..., x_m)$ has a probability density $P_h(\mathbf{x})$. We have that

$$\int_{\mathbf{x} \in \mathbb{R}^m} P_h(\mathbf{x}) = 1 \tag{2.2}$$

For a real world problem, $P_h(\cdot)$ may follow a highly non-linear distribution and it is possible that there are many $\mathbf{x} \in \mathbb{R}^m$ that $P_h(\mathbf{x}) = 0$.

The datasets we are dealing with – such as training, test, and validation datasets – are all assumed to be obtained by sampling from this distribution.

2.2 Usual Model Evaluation Methods

The first set of evaluation methods are typically applied to various practical applications.

2.2.1 Test Accuracy and Error

Given the input of a machine learning model follows an unknown distribution P_h, it is straightforward that we may estimate how good the model is by using a set of data sampled from the distribution. Test accuracy uses a set of test instances D_{test}, or test dataset, to evaluate the model f by letting

$$Acc(f, D_{test}) = \frac{1}{|D_{test}|} \sum_{(\mathbf{x},y)\in D_{test}} (1 - \mathbf{I}(f(x), y)) \tag{2.3}$$

where $\mathbf{I}(\cdot, \cdot)$ denotes the 0-1 loss, i.e., $\mathbf{I}(y_1, y_2) = 0$ when $y_1 = y_2$, and $\mathbf{I}(y_1, y_2) = 1$ otherwise.

Moreover, the test error is

$$Err(f, D_{test}) = 1 - Acc(f, D_{test}) \tag{2.4}$$

2.2.2 Accuracy w.r.t. Training Set Size

It is interesting to understand, for different machine learning models and different datasets, how the test accuracy changes with respect to the size of training dataset. Figure 2.1 presents a comparison between decision tree and logistic regression over the California housing dataset. We can see that, in this example, the accuracy of the decision tree increases almost linearly with respect to the size of the training dataset, while the logistic regression increases quickly when the size of training dataset is small but converges (without significant increase) afterwards.

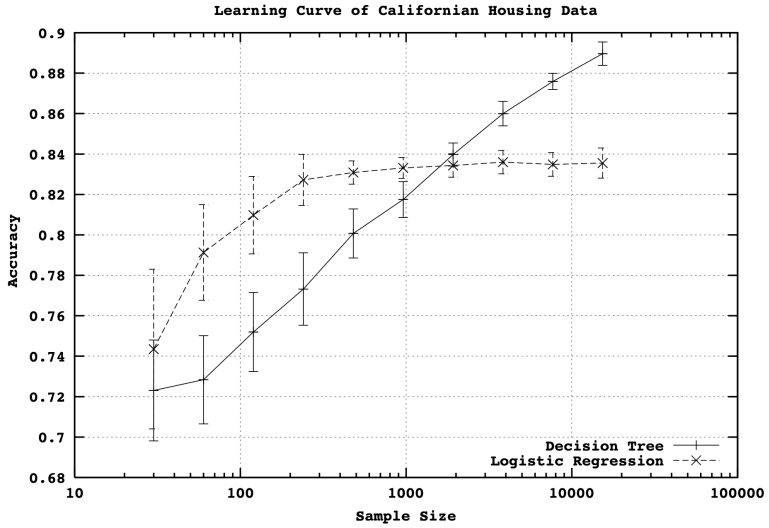


Figure 2.1: Accuracy w.r.t. Training Set Size [11]

How to draw curve? The algorithm proceeds by following Algorithm 1.

Algorithm 1: ***ConstructLearaningCurve***($D_{training}, D_{test}$), where $D_{training}$ is a set of training instances and D_{test} is a set of test instances

```
for each sample size s on learning curve do
    counter = 0
    acc = []
    while counter < k do
        randomly select s instances from D_training
        learn a model f
        evaluate the model f on test set D_test to determine accuracy a
        acc = acc.append(a)
        counter = counter + 1
    end
    plot (s, average accuracy and error bar over acc)
end
```

2.2.3 Multiple Training/Test Partitions

For a real world application, we may not have enough data to make sufficiently large training and test datasets. In this case, the resulting model may be sensitive to the sizes of the datasets. Specifically, a larger test dataset gives us more reliable estimate of accuracy (i.e. a lower variance estimate), but a larger training dataset will be more representative of how much data we actually have for learning process.

In such cases, a single training dataset does not tell us how sensitive accuracy is to a particular training and test split, and we may consider using multiple training/test partitions to evaluate.

Random resampling We can address the issue by repeatedly randomly partitioning the available data D into training dataset $D_{training}$ and test dataset D_{test}.

When randomly selecting training datasets, we may want to ensure that class proportions are maintained in each selected dataset.

Cross validation partitions the data into k subsets, and iteratively leaves one out for test and uses the rest for training. The resulting accuracy is the average over all the iterations. In general, cross validation makes efficient use of the available data for testing.

In practice, 10-fold cross validation is common, but smaller values of k are often used when learning takes a lot of time.

2.2.4 Confusion Matrix

Up to now, we have some statistic measurements on roughly how good a model is. However, we have not been able to take a look at what types of mistakes the model makes. To this end, confusion matrix is often used. Confusion matrix is a matrix where each row represents the number of instances in a *predicted* class, while each column represents the number of instances in an *actual* class (or vice versa). Therefore, those numbers on the main diagonal represent the

number of instances whose predictive and true labels are the same, and those numbers not on the main diagonal represent mis-classifications.

Example 22. *Equation (2.6) presents a confusion matrix for the* ***digits*** *dataset over 360 test instances, for a Naive Bayes classifier we trained. We can see that, only four instances are mis-classified, two of them are supposed to be label 1 but classified as 2, one of them is supposed to be 6 but classified as 8, and one of them is supposed to be 9 but classified as 5.*

$$\begin{array}{c} \\ \textbf{y=0} \\ \textbf{y=1} \\ \textbf{y=2} \\ \textbf{y=3} \\ \textbf{y=4} \\ \textbf{y=5} \\ \textbf{y=6} \\ \textbf{y=7} \\ \textbf{y=8} \\ \textbf{y=9} \\ \\ \end{array} \begin{pmatrix} 28 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 41 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 34 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 42 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 46 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 24 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 39 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 32 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 1 & 0 & 36 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 34 \end{pmatrix} \begin{array}{c} \\ \textbf{y=0} \\ \textbf{y=1} \\ \textbf{y=2} \\ \textbf{y=3} \\ \textbf{y=4} \\ \textbf{y=5} \\ \textbf{y=6} \\ \textbf{y=7} \\ \textbf{y=8} \\ \textbf{y=9} \\ \\ \end{array} \tag{2.5}$$

Binary classification problem Consider a binary classification problem. Without loss of generality, we assume that the two classes are $positive$ and $negative$, respectively. Then, we have the following confusion matrix

$$\begin{array}{c} \textbf{y=positive} \\ \textbf{y=negative} \\ \\ \end{array} \begin{pmatrix} \text{true positives (TP)} & \text{false positives (FP)} \\ \text{false negatives (FN)} & \text{true negatives (TN)} \end{pmatrix} \begin{array}{c} \\ \textbf{y=positive} \\ \textbf{y=negative} \\ \\ \end{array} \tag{2.6}$$

where TP denotes the number of true positives, FP the number of false positives, TN the number of true negatives, and FN the number of false negatives. Note that, $|D_{test}| = \text{TP} + \text{FP} + \text{TN} + \text{FP}$

We note that, in this case,

$$Acc(f, D_{test}) = \frac{\text{TP} + \text{TN}}{|D_{test}|} \text{ and } Err(f, D_{test}) = \frac{\text{FP} + \text{FN}}{|D_{test}|} \tag{2.7}$$

Is accuracy an adequate measure of predictive performance? Probably not. For example, when there is a large class skew negative, a high accuracy may be misleading.

Example 23. *For a dataset of 1,000 instance, 97% of them are supposed to be negative. In this case, a 98% accuracy may simply be the case that the classifier classifies all negative instances correctly but 2/3 of the positive instances wrongly. This is undesirable for e.g., medical diagnosis, where most of the cases are negative but a true negative may lead to serious consequence.*

Other accuracy metrics To deal with the problem given in Example 23, we may consider

$$\text{true positive rate (recall)} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.8}$$

which focuses on instances whose ground truth are positive. The greater the true positive is, the better the classifier.

Example 24. *The recall of the case in Example 23 is 1/3.*

Similarly, we may consider

$$\text{false positive rate} = \frac{\text{FP}}{\text{TN} + \text{FP}} \tag{2.9}$$

which focuses on instances whose ground truth are negative. However, opposite to the case of true positive rate, the smaller the false positive is, the better the classifier. Moreover, we may be interested in

$$\text{positive predictive value (precision)} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.10}$$

which focuses on those instances whose predictive values are positive.

2.2.5 ROC curves and AUC

A Receiver Operating Characteristic (ROC) curve plots the true positive rate (Equation 2.8) vs. the false positive rate (Equation 2.9) as a threshold on the confidence of an instance being positive is varied. Figure 2.2 presents an ROC curve on the **iris** dataset and a logistic classifier.

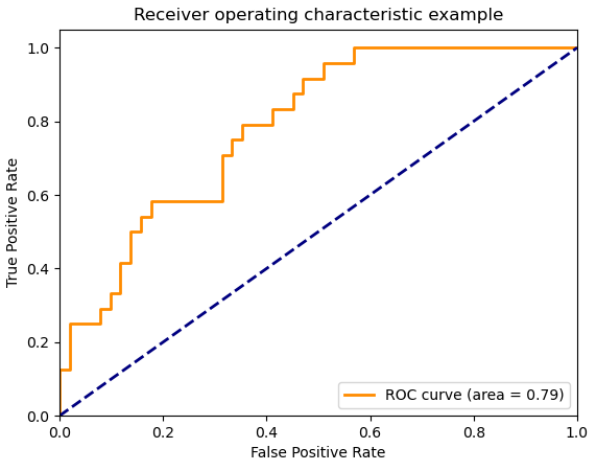


Figure 2.2: An ROC curve for **iris** dataset and a logistic classifier

How to draw ROC curve? First of all, we can get a function **calculate_TP_FP_rate(y_test, y_test_preds)** to compute the TP rate and FP rate given the ground truth labels **y_test** and the predicted labels **y_test_preds**. Then, the following Algorithm 2 enables the computation of a set of (TP rate, FP rate) by working with a set of probability thresholds. The key is to compute predicted labels **y_test_preds** based on the predictions **y_predict** and a probability threshold p.

Algorithm 2: ***ROC-curve***(y_test,y_predict), where y_test is the vector of ground truth label, and y_predict is the vector of predictions

```
probability_thresholds = np.linspace(0, 1, num = 100)
for p in probability_thresholds do
    y_test_preds = []
    for prob in y_predict do
        if prob > p then
            y_test_preds.append(1)
        else
            y_test_preds.append(0)
        end
    end
    tp_rate, fp_rate = calculate_TP_FP_rate(y_test, y_test_preds)
    tp_rates.append(tp_rate)
    fp_rates.append(fp_rate)
end
return tp_rates, fp_rates
```

How to read the ROC curves? A skilful model will assign a higher probability to a randomly chosen real positive occurrence than a negative occurrence on average. This is what we mean when we say that the model has skill. Generally, skilful models are represented by curves that bow up to the top left of the plot. A model with no skill is represented at the point (0.5, 0.5). A model with no skill at each threshold is represented by a diagonal line from the bottom left of the plot to the top right and has an AUC of 0.5. A model with perfect skill is represented at a point (0,1). A model with perfect skill is represented by a line that travels from the bottom left of the plot to the top left and then across the top to the top right.

2.2.6 PR curves

A precision/recall curve plots the precision (Equation 2.10) vs. recall (Equation 2.8) (TP-rate) as a threshold on the confidence of an instance being positive is varied.

2.3 Safety and Reliability Issues

The second set of evaluation methods are more involved, and are intensively studied in recent years. Despite the success of machine learning in many areas, serious concerns have been raised in applying machine learning to real-world safety-critical systems such as self-driving cars, automatic medical diagnosis, etc. So, the second group of evaluation methods focuses on a few safety and reliability issues of machine learning.

Simply speaking, a learned model f is to approximate a target function h. Therefore, the erroneous behaviour of f exists when it is inconsistent with h. We use $f_j(\mathbf{x})$ to denote its j-th element of $f(\mathbf{x})$. Then, we have the following definition for the classification task.

Definition 4 (Erroneous Behavior of a Classifier)**.** *Given a (trained) classifier $f\colon \mathbb{R}^n \to \mathbb{R}^k$, a target function $h\colon \mathbb{R}^n \to \mathbb{R}^k$, an erroneous behavior of the classifier f is exhibited by a legitimate input $x \in \mathbb{R}^n$ such that*

$$\arg\max_j f_j(\boldsymbol{x}) \neq \arg\max_j h_j(\boldsymbol{x}) \tag{2.11}$$

Intuitively, an erroneous behaviour is witnessed by the existence of an input $\mathbf{x}$ on which the classifier and the target function return different result. Note that, the legitimate input $\mathbf{x}$ can be any input in the input domain $\mathbb{R}^n$ and does not have to be a training instance.

2.3.1 Generalisation Error

One of the key successes of machine learning is that it is able to work with unseen data, i.e., data instances that are not within the training dataset. It is meaningful to understand how good a model is on unseen data. Given an instance $\mathbf{x}$, we use a loss function to measures the discrepancy between the true label y and the model's predicted label $f(\mathbf{x})$, written as $\mathcal{L}(y, f(\mathbf{x}))$.

Given a test dataset D sampled i.i.d. from the underlying distribution $\mathcal{D}$, the model f's empirical loss is

$$\mathcal{L}_{emp}(f, D) \stackrel{\text{def}}{=} \frac{1}{|D|} \sum_{(\mathbf{x},y)\in D} \mathcal{L}(y, f(\mathbf{x})) \tag{2.12}$$

while the expected loss is

$$\mathcal{L}_{exp}(f, \mathcal{D}) \stackrel{\text{def}}{=} \mathbb{E}_{(\mathbf{x},y)\in\mathcal{D}} \mathcal{L}(y, f(\mathbf{x})) \tag{2.13}$$

Apparently, $\mathcal{L}_{emp}(f, D)$ is the approximation of $\mathcal{L}_{exp}(f, D)$, since the underlying distribution $\mathcal{D}$ is unknown to the learning algorithm. Their gap

$$GE(f, D) = |\mathcal{L}_{emp}(f, D) - \mathcal{L}_{exp}(f, D)| \tag{2.14}$$

is called generalisation error. Recall that, the test error $Err(f, D_{test})$ (Equation (2.4)) is an empirical loss when the loss function $\mathcal{L}$ is the 0-1 loss and the dataset D is the test dataset.

Generalisation error is closely related to the overfitting problem of machine learning algorithms. A machine learning model is overfitted if it performs well on training data samples but badly on test data samples. Usually, a large generalisation error indicates the overfitting.

2.3.2 Adversarial Examples

Adversarial examples [13] represent another class of erroneous behaviours that also introduce safety implications. Here, we take the name "adversarial example" due to historical reasons. Actually, as suggested in the below definition, it represents a mis-match of the decisions made by a human and by a neural network, and does not necessarily involve an adversary.

Intuitively, as shown in Figure 2.3, it is possible that, given an instance $\mathbf{x}$, it is classified correctly, i.e., $y = f(\mathbf{x})$, but a small perturbation on $\mathbf{x}$ (such as the one-pixel change as in Figure 2.3) may lead to a change of classification, i.e., $f(\mathbf{x} + \epsilon) \neq f(\mathbf{x})$. Such misclassification may have serious security implications, as the traffic light example in Figure 2.3.

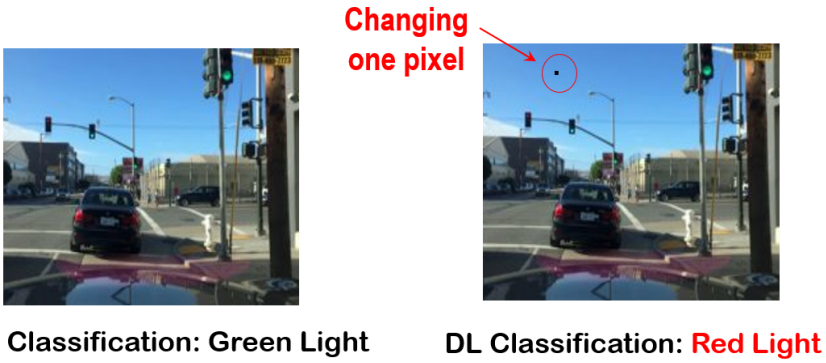


Figure 2.3: By changing one pixel in a "Green-Light" image, a state-of-the-art image classifier misclassifies it as "Red-Light" [14]

For the **iris** dataset, if we create a new data instance, indexed as 151 in the below Table 2.2, such that the only difference with the one indexed as 150 is the petal width, from 1.8 to 1.6. For a well trained decision tree classifier, the new instance may be classified as a different class.

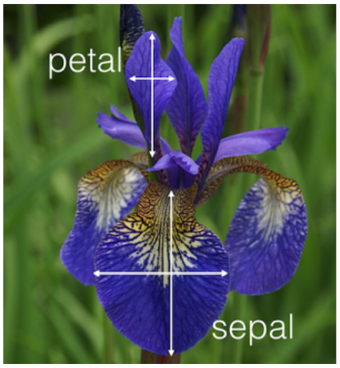


Table 2.1: An iris flower

index	Sepal Length	Sepal Width	Petal Length	Petal Width	Class Label
1	5.1	3.5	1.4	0.2	iris setosa
2	4.9	3.0	1.4	0.2	iris setosa
...					
50	6.4	3.5	4.5	1.2	iris versicolor
...					
150	5.9	3.0	5.1	1.8	iris virginica
151	5.9	3.0	5.1	1.6	iris versicolor

Table 2.2: Iris dataset

Definition 5 (Adversarial Example). *Given a (trained) deep neural network* $f: \mathbb{R}^{s_1} \to \mathbb{R}^{s_K}$, *a human decision oracle* $h: \mathbb{R}^{s_1} \to \mathbb{R}^{s_K}$, *and a legitimate input* $x \in \mathbb{R}^{s_1}$ *with* $\arg\max_j f_j(x) =$

$\arg\max_j h_j(x)$, *an adversarial example to a DNN is defined as:*

$$\begin{aligned} \exists \hat{x}: \quad & \arg\max_j h_j(\hat{x}) = \arg\max_j h_j(x) \\ & \wedge \, ||x - \hat{x}||_p \leq d \\ & \wedge \, \arg\max_j f_j(\hat{x}) \neq \arg\max_j f_j(x) \end{aligned} \tag{2.15}$$

where $p \in$, $p \geq 1$, $d \in$, *and* $d > 0$.

Intuitively, x is an input on which the DNN and an human user have the same classification and, based on this, an adversarial example is another input $\hat{x}$ that is classified differently than x by network f (i.e., $\arg\max_j f_j(\hat{x}) \neq \arg\max_j f_j(x)$), even when they are geographically close in the input space (i.e., $||x - \hat{x}||_p \leq d$) and the human user believes that they should be the same (i.e., $\arg\max_j h_j(\hat{x}) = \arg\max_j h_j(x)$).

Measurement of Adversarial Examples Definition 5 explains the adversarial example, but given there may be multiple adversarial examples satisfying the conditions, there needs to be some measurement by which we may decide that some adversarial examples are more interesting than others.

Definition 6. *(Quality of Adversarial Examples) An adversarial example is usually measured from two aspects:*

- *magnitude of perturbation* $||\boldsymbol{x} - \boldsymbol{x}'||$, *where* $|| \cdot ||$ *is a norm distance such as those in Section 1.3.3,*
- *probability gap between and after the perturbation, i.e.,* $|f_y(\boldsymbol{x}) - f_y(\boldsymbol{x}')|$.

Therefore, instead of concerning any adversarial example satisfying Definition 5, we may be interested in the following optimisation problem, for some instance $(\mathbf{x}, y) \in \mathcal{D}$,

$$\begin{aligned} \text{minimise} \quad & ||\mathbf{x} - \mathbf{x}'|| - \lambda |f_y(\mathbf{x}) - f_y(\mathbf{x}')| \\ \text{subject to} \quad & f(\mathbf{x}) \neq f(\mathbf{x}') \\ & ||\mathbf{x} - \mathbf{x}'|| \leq \delta \end{aligned} \tag{2.16}$$

where λ is a hyper-parameter balancing two objectives, and δ indicates the maximum perturbation that may be considered.

2.3.3 Poisoning Attacks

Poisoning attack occurs when the adversary injects malicious data into training process, and hence get a machine learning algorithm to learn something it should not. There are two types of poisoning attacks, one for data poisoning attacks and the other for backdoor attacks.

2.3.4 Backdoor Attacks

Given a *triggered* input $\mathbf{x}^\alpha = \mathbf{x} + \Delta$, where Δ is the trigger stamped on a "clean" input $\mathbf{x}$, the predicted label will always be the label y_α that is set by the attacker, regardless of what the input $\mathbf{x}$ might be. In other words, as long as the triggered input $\mathbf{x}^\alpha$ is present, the backdoor model will always classify the input to the attacker's target label (i.e. y_α). However, for "clean" inputs, the backdoor model behaves as the original model without any observable performance reduction. For example, Figure 2.4 presents an example on MNIST dataset.

(a) clean inputs representing different digits (b) backdoor inputs, all classified as 8

Figure 2.4: All MNIST images of handwritten digit with a backdoor trigger (a white patch close to the bottom right of the image) are mis-classified as digit 8.

2.4 Practicals

We train a simple perceptron model and output the confusion matrix.

```
from sklearn import datasets
dataset = datasets.load_digits()
X = dataset.data
y = dataset.target

print("===== Get Basic Information ======")
observations = len(X)
features = len(dataset.feature_names)
classes = len(dataset.target_names)
print("Number of Observations: " + str(observations))
print("Number of Features: " + str(features))
print("Number of Classes: " + str(classes))

print("===== Split Dataset ======")
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20)

print("===== Model Training ======")
from sklearn.linear_model import Perceptron
clf = Perceptron(tol=1e-3, random_state=0)
clf.fit(X_train, y_train)

print("===== Model Prediction ======")
print("Labels of all instances:\n%s"%y_test)
y_pred = clf.predict(X_test)
print("Predictive outputs of all instances:\n%s"%y_pred)

print("===== Confusion Matrix ======")
from sklearn.metrics import classification_report, confusion_matrix
print("Confusion Matrix:\n%s"%confusion_matrix(y_test, y_pred))
print("Classification Report:\n%s"%classification_report(y_test, y_pred))
```

2.5 Exercises

1. Write a program to implement
   - ROC curve
   - PR curve

   and check how it works on the example given in Section 2.4.
2. Compare a few training/test splits (0.9/0.1, 0.8/0.2, 0.7/0.3) and check their differences on training and test accuracy.
3. Compare a few training/test splits (0.9/0.1, 0.8/0.2, 0.7/0.3) and check their differences on confusion matrix.
4. Try to find a data poisoning strategy to make the trained model mis-classify on a given training input.
5. Read the literature to understand the state-of-the-art for backdoor attacks.

3 Chapter 3: Simple Machine Learning Models

In this chapter, we will go through a few key machine learning algorithms that do not belong to deep learning. While deep learning is popular, these algorithms are still playing key roles in various applications. These algorithms are more transparent than deep learning, and their results can be easier to be explained.

We will consider decision tree, K-nearest neighbor, linear/logistic regression, gradient descent, naive Bayes.

3.1 Decision Tree

Decision tree is one of the simplest, yet popular, machine learning algorithms. It has a very long history of research and application, and has many variants. In this section, we only consider one of its variants.

Figure 3.1 shows a decision tree for the **iris** dataset.

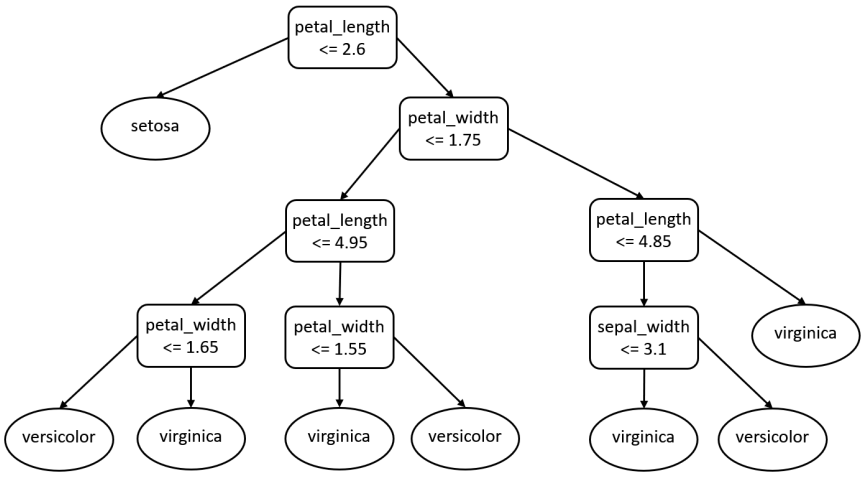


Figure 3.1: A decision tree for **iris** dataset [6]

3.1.1 Learning Algorithm

Algorithm 3 provides a program sketch of a function ***ConstructSubTree***(D) for D a set of training instances. Intuitively, given D, this function will construct and return a tree T_D

to classify the training instances in D. The tree construction is a recursive process, i.e., the construction of T_D is completed by the construction of its children $T_{D_1}, ..., T_{D_k}$, which are implemented by calling ***ConstructSubTree***(D_i) for $i \in \{1, ..., k\}$ such that $D = \bigcup_{i=1}^{k} D_i$. Once D satisfies the stopping criteria, a leaf node is constructed and the recursive process terminates by making T_D be the leaf node.

Algorithm 3: ***ConstructSubTree***(D), where D is a set of training instances

```
1  X_i = DetermineSplittingFeature(D)
2  if StoppingCriteriaMet(D) then
3  |   make a leaf node N
4  |   determine class label or regression value for N
5  else
6  |   make an internal node
7  |   S = FindBestSplit(D, X_i)
8  |   for each outcome k of S do
9  |   |   D_k = subset of instances that have outcome k
10 |   |   k-th child of N = ConstructSubTree(D_k)
11 |   end
12 end
13 return subtree rooted at N
```

Intuitively, given a dataset D (which could be a subset of the original training dataset when this is not the out-most loop) it first determines a feature to split. Then, it checks whether the stopping criteria holds. If holds, it makes a leaf node and returns. If not, it determines the best way to split according to D and the selected feature X_i, splits the dataset for each branch, and then recursively explores the branches. From Algorithm 3, we can see that there are three functions: ***DetermineSplittingFeature***, ***FindBestSplit***, and ***StoppingCriteriaMet***, which we will explain below.

Determine splitting feature

The function ***DetermineSplittingFeature*** is to select from the set X of features a feature X_i. This feature X_i will then be used later in ***FindBestSplit*** to determine how to construct children nodes. Before explaining how to select feature, we discuss a basic principle of decision tree learning.

Occam's razor attributes to William of Ockham of 14th century, an English philosopher. He said that "when you have two competing theories that make exactly the same predictions, the simpler one is the better". Similar vision also exists in e.g., Ptolemy's statement that "We consider it a good principle to explain the phenomena by the simplest hypothesis possible".

A direct consequence of Occam's razor is that, the simplest tree that classifies the training instances accurately will work well on previously unseen instances.

Computation of the smallest tree that accurately classifies the training set, unfortunately, is shown NP-hard [7]. Therefore, it will be interesting to find a sub-optimal solution for the tree construction.

Heuristics for greedy search We consider information-theoretic heuristics to greedily choose features to split. The basic idea is to use uncertainty as the heuristics, i.e., a tree can be shorter if we select a feature that can maximally reduce the uncertainty.

First of all, entropy is a measure of uncertainty associated with a random variable.

Definition 7. *(Entropy) Let X be a random variable with possible values $V(X)$. Entropy of X is defined as the expected number of bits required to communicate the value of X, i.e.,*

$$H(X) = -\sum_{x \in V(X)} P(x) \log_2 P(x) \tag{3.1}$$

Conditional entropy measures the entropy of a random variable with some known information from the other random variable.

Definition 8. *(Conditional Entropy) Let X and Y be two random variables with possible values $V(X)$ and $V(Y)$, respectively. Conditional entropy of X given Y quantifies the amount of information needed to describe the outcome of X given that the value of Y is known, i.e.,*

$$H(X|Y) = -\sum_{y \in V(Y)} P(y) H(X|y) \tag{3.2}$$

where

$$H(X|y) = -\sum_{x \in V(X)} P(x|y) log_2 P(x|y) \tag{3.3}$$

Based on them, we can define mutual information and information gain.

Definition 9. *(Mutual Information, or Information Gain) Let X and Y be two random variables. Their mutual information, a measure of the mutual dependence between the two variables, is defined as follows:*

$$I(X, Y) = H(X) - H(X|Y) \tag{3.4}$$

In the context of selecting features to split, we have a random variable X for the training data and a random feature X_i for a specific feature, the following information gain

$$\textbf{\textit{InfoGain}}(X_i, X) = H(X) - H(X|X_i) \tag{3.5}$$

is to measure the mutual dependence between feature X_i and the training data. Apparently, a larger information gain represents a stronger mutual dependence and a split on that feature will lead to more drastic decrease of the uncertainty in the dataset. Therefore, we have our heuristics as the information gain.

Gain ratio is an improvement to the information gain, which may bias towards features with many values. Consider a feature that uniquely identifies each training instance, splitting on this feature would result in many branches, each of which is "pure" (has instances of only one class). The information gain in this case is maximal.

The gain ration is a normalised information gain, as follows.

Definition 10. *(Gain ratio) Let X and Y be two random variables with possible values $V(X)$ and $V(Y)$, respectively. The gain ratio is the information gain normalized over the entropy, i.e.,*

$$\textbf{\textit{GainRatio}}(X_i, X) = \frac{\textbf{\textit{InfoGain}}(X_i, X)}{H(X)} = \frac{H(X) - H(X|X_i)}{H(X)} \tag{3.6}$$

Find best split

The function ***FindBestSplit*** is to, given a feature, determine how to generate a set of children nodes. Assume that we have determined a feature f as the splitting feature.

On Numeric Features Algorithm 4 presents an algorithm to determine candidate splits. Intuitively, it first partitions the dataset D into a set of smaller datasets $s_1, ..., s_k$ such that each smaller dataset has the same value for feature f. Then, it sorts the datasets $s_1, ..., s_k$ according to the value. Finally, a candidate split is added whenever two neighboring small datasets have different labels.

Algorithm 4: DetermineCandidateNumericSplit(D, X_i), where D is a set of training instances and X_i is a feature

```
C = {};
S = partition instances in D into sets s_1, ..., s_k where the instances in each set have the
  same value for X_i
let v_j denote the value of X_i for set s_j
sort the sets in S using v_j as the key for each s_j
for each pair of adjacent sets s_j, s_{j+1} in sorted S do
    if s_j and s_{j+1} contain a pair of instances with different class labels then
        add candidate split X_i ≤ (v_j + v_{j+1}/2) to C
    end
end
return C
```

Stopping Criteria

Stopping criteria determines when to form a leaf node. Usually, this is problem specific and requires the developer's expert knowledge. However, we should certainly terminate when

C1 all of instances are of the same class,

and in most cases a termination should be warrant when

C2 we've exhausted all of the candidate splits

In many cases, we consider the termination according to

C3 the accuracy to a validation dataset.

Alternatively, instead of considering an early termination, we may consider growing a large tree and then pruning back, i.e., conducting the following two steps iteratively until reaching an accuracy threshold

- evaluate the impact on the accuracy on validation dataset after pruning each node
- greedily remove the one that least reduces the accuracy on validation dataset

3.1.2 Classification Probability

It is noted that, in Algorithm 3, we need to determine "class label or regression value" for leaf nodes. Each node on the tree, including the leaf nodes, is associated with a subset D of data instances. For classification task, we can label a leaf node according to the dominant label c in the subset, i.e.,

$$c = \arg\max_{c \in C} \{|\{y = c | (\mathbf{x}, y) \in D\}|\} \tag{3.7}$$

For regression task, we can have the regression value as

$$c = \frac{1}{|D|} \sum_{(\mathbf{x},y) \in D} y \tag{3.8}$$

Moreover, as discussed in Section 1.1.4, for a classification task, it is normally expected that it will return a probability distribution over the classes. To do this, we can let

$$P(c) = \frac{|\{y = c | (\mathbf{x}, y) \in D\}|}{|D|} \tag{3.9}$$

for all $c \in C$. Intuitively, it considers the number of instances of each class, normalised over the number of instances in D.

3.1.3 Adversarial Example

In the following, we present a heuristic algorithm to search for adversarial example in a given decision tree with a given instance $\mathbf{x}$. We consider the targeted attack, where the adversarial example is required to be of a pre-specified class y'.

The algorithm proceeds with the following steps:

1. Given an input $\mathbf{x}$, it will lead to some leaf node z with label y.
2. Consider a targeted label $y' \neq y$, we find the shortest path from z to any leaf node with label y'. Let the new leaf node be z'.
3. Then, we can identify the common ancestor of z and z' on the shortest path, and construct a path from the root node to the common ancestor and then to z'.
4. Construct an input $\mathbf{x}'$ from the constructed path such that $||\mathbf{x} - \mathbf{x}'||$ is minimised. If $||\mathbf{x} - \mathbf{x}'|| < \delta$ then we return $\mathbf{x}'$ as an adversarial example.

Is $\mathbf{x}'$ an adversarial example? Yes, because $\mathbf{x}'$ follows a path from the root node to the leaf node z', which is labelled as y', different from y.

Is this approach complete? A complete approach is able to find an adversarial example if there is any. Unfortunately, the above algorithm is incomplete.

Sub-optimality This algorithm is also sub-optimal, i.e., the found $\mathbf{x}'$ does not necessarily be the optimal solution to the optimisation problem described in Equation (2.16).

3.1.4 Practicals

First of all, we need to load dataset. Here, we use the **sklearn** library's in-build dataset **iris** as an example.

```
from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
y = iris.target
```

We can print some necessary information about the dataset.

```
observations = len(X)
features = len(iris.feature_names)
classes = len(iris.target_names)
print("Number of Observations: " + str(observations))
print("Number of Features: " + str(features))
print("Number of Classes: " + str(classes))
```

Then, in the dataset, we make the training-test split. Here, we consider 8:2 split.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20)
```

Decision Tree For decision tree, we can do the following:

```
from sklearn.tree import DecisionTreeClassifier

tree = DecisionTreeClassifier()
tree.fit(X_train, y_train)
print("Training accuracy is %s"% tree.score(X_train,y_train))
print("Test accuracy is %s"% tree.score(X_test,y_test))
```

Basically, it initialises a classifier, and then fits the initialised classifier with the training data, and then output the accuracy on the test dataset.

We can also get predictions by having

```
print("Labels of all instances:\n%s"%y_test)
y_pred = tree.predict(X_test)
print("Predictive outputs of all instances:\n%s"%y_pred)
```

Other more detailed information may also be available, such as

```
from sklearn.metrics import classification_report, confusion_matrix
print("Confusion Matrix:\n%s"%confusion_matrix(y_test, y_pred))
print("Classification Report:\n%s"%classification_report(y_test, y_pred))
```

Command to Run Finally, if we put the code into the **simpleML.py** file, we can run the following commands to check the result:

```
$ conda activate aisafety
$ python3 decision_tree.py
```

Decision Tree Construction

In the following, we write our own code for tree construction. Let **decisionTree.py** be the new file. First of all, we get the **iris** dataset. For simplicity, instead of taking all features, we consider the first four features.

```
import math

def get_iris():
    from sklearn import datasets
    iris = datasets.load_iris()
    X = iris.data
    y = iris.target

    data_iris = []
    for i in range(len(X)):
        dict = {}
        dict['f0'] = X[i][0]
        dict['f1'] = X[i][1]
        dict['f2'] = X[i][2]
        dict['f3'] = X[i][3]

        dict['label'] = y[i]
        data_iris.append(dict)
    return data_iris

data = get_iris()
label = 'label'
```

Now, we can construct a decision tree. There are three main functionalities: check information gain to determine the feature for splitting, create leaf nodes if termination condition satisfied, and create branches and subtrees.

```
def makeDecisionTree(data, label, parent=-1, branch=''):

    global node, nodeMapping
    if parent >= 0:
        edges.append((parent, node, branch))

    # Find the variable (i.e., column) with maximum information gain
    infoGain = []
    columns = [x for x in data[0]]
    for column in columns:
        if not(column == label):
            ent = entropy(data, label)
            infoGain.append((findInformationGain(data, label, column, ent),
    column))
```

```
    splitColumn = max(infoGain)[1]

    # Create a leaf node if maximum information gain is not significant
    if max(infoGain)[0] < 0.01:
        nodeMapping[node] = data[0][label]
        node += 1
        return
    nodeMapping[node] = splitColumn
    parent = node
    node += 1
    branchs = { i[splitColumn] for i in data }# All out-going edges from
    current node
    for branch in branchs:
        # Create sub table under the current decision branch
        modData = [x for x in data if splitColumn in x and x[splitColumn] ==
    branch]
        for y in modData:
            if splitColumn in y:
                del y[splitColumn]

        # create sub-tree
        makeDecisionTree(modData, label, parent, branch)
```

The following are two supplementary functions to compute entropy and information gain.

```
def entropy(data, label):
    cl = {}
    for x in data:
        if x[label] in cl:
            cl[x[label]] += 1
        else:
            cl[x[label]] = 1
    tot_cnt = sum(cl.values())
    return sum([ -1 * (float(cl[x])/tot_cnt) * math.log2(float(cl[x])/tot_cnt
    ) for x in cl])
```

```
def findInformationGain(data, label, column, entropyParent):
    keys = { i[column] for i in data }
    entropies = {}
    count = {}
    avgEntropy = 0
    for val in keys:
        modData = [ x for x in data if x[column] == val]
        entropies[val] = entropy(modData, label)
        count[val] = len(modData)
        avgEntropy += (entropies[val] * count[val])

    tot_cnt = sum(count.values())
    avgEntropy /= tot_cnt
    return entropyParent - avgEntropy
```

Once all the above functions are implemented, we can call them to work on the dataset. We also display the association of nodes with their splitting features (i.e., nodemapping) and the edges of the tree.

```
node = 0
nodeMapping = {}
edges = []

makeDecisionTree(data, label)
print('nodemapping ==> ', nodeMapping, '\n\nedges ===>', edges)
```

After the construction of the decision tree, we may want to query the decision tree with a new unseen data. This starts from the following query function:

```
def query(i, data_x):
    next_q = False
    for e in edges:
        if e[0]==i:
            next_q=True
            break
    if next_q:
        for e in edges:
            if e[0]==i and e[2]==data_x[str(nodeMapping[i])]:
                i = e[1]
                query(i, data_x)
    else:
        print('predict_label:', nodeMapping[i])
```

The following is the query command.

```
data_x = get_iris()[68]
query(0, data_x)
print()
print('original_data:', data_x)
print('original_path:',path,' predict_label:', label_x)
```

Adversarial Attack on Decision Tree Construction

The following is a simple implementation of an adversarial attack:

```
#ATTACK
attack_label = None
attack_path = None

def judge_e(i):
    next_ = False
    for e in edges:
        if e[0]==i:
            next_=True
            break
    return next_

def atk_path(path_,i):
    global attack_label, attack_path
    for e in edges:
        ppath = copy.deepcopy(path_)
        if e[0]==i:
            ppath.append(e[1])
            if judge_e(e[1]):
```

```
                atk_path(ppath,e[1])
            elif nodeMapping[e[1]]!=label_x and attack_label==None:
                attack_path = ppath
                attack_label = nodeMapping[e[1]]

def attack():
    for i in range(1,len(path)):
        atk_path(path[:-i],path[-1-i])
        if attack_label != None:
            break

attack()
print('attack_path:',attack_path,' attack_label:', attack_label)
```

3.2 K-Nearest Neighbor

Most of the machine learning application have at least two stages: learning stage and deployment stage. K-nn is a lazy learner, i.e., unlike the decision tree which learns a model (i.e., tree) during the the leaning stage, it does nothing during the learning stage. In deployment stage, it directly computes the result by utilising the information from the training dataset D.

Definition 11. *(K-nn) Given a training dataset D, a number k, a distance measure $||\cdot||$, and a new instance x, it is to*

1. *find k instances in D that are closest to x according to $||\cdot||$, and*
2. *summarise learning result from the labelling information of the k instances, e.g., assign the most occurring label of the k instances to x.*

When to consider? Usually, K-nn is useful when there are less than 20 features per instance and we have lots of training data.

Advantages of K-nn includes e.g., no training is needed, able to learn complex target functions, do not lose information, etc. We will discuss its disadvantages later.

Classification Let $(x^{(1)}, y^{(1)}), ..., (x^{(k)}, y^{(k)})$ be the k nearest neighbors. Formally, the classification is to assign the following label to x:

$$\hat{y} \leftarrow \underset{v \in V(Y)}{\arg\max} \sum_{i=1}^{k} \delta(v, y^{(i)}) \tag{3.10}$$

where

$$\delta(a, b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases} \tag{3.11}$$

Intuitively, it return the class that have the most number of instances in the k training instances.

We can also consider its weighted variant, e.g.,

$$\hat{y} \leftarrow \underset{v \in V(Y)}{\arg\max} \sum_{i=1}^{k} w_i \delta(v, y^{(i)}) \tag{3.12}$$

where

$$w_i = \frac{1}{d(x, x^{(i)})} \tag{3.13}$$

Intuitively, it considers not only the occurrence number of a label but also the quality of those occurrence, i.e., those occurrence that are closer to x has a higher weight.

Regression is to assign the following value to x:

$$\hat{y} \leftarrow \frac{1}{k}\sum_{i=1}^{k} y^{(i)} \tag{3.14}$$

We can also consider its weighted variant, e.g.,

$$\hat{y} \leftarrow \frac{\sum_{i=1}^{k} w_i y^{(i)}}{\sum_{i=1}^{k} w_i} \tag{3.15}$$

Issues The following are a few key issues of K-nn.

- The choice of hyper-parameter k. Actually, the increasing of k reduces variance, but increases bias.
- For high-dimensional space, the nearest neighbor may not be very close at all. This requires a large dataset when there are many features.
- Memory-based technique is needed. Naively, it must take a pass through the data for each classification. This can be prohibitive for large data sets.

Irrelevant features in instance-based learning In K-nn, the learning can be seriously affected by the irrelevant features. For example, an instance may be classified correctly with the existing set of features, but will be classified wrongly after adding a new, noise feature.

One way around this limitation is to weight features differently, so that the importance of noise features is reduced. Assume that an instance x is expressed as a function $f(x) = w_0 + w_1 x_1 + ... + w_n x_n$, for the instance $x = (x_1, ..., x_n)$ of n features. We can find weights w_i by having

$$\arg \min_{w_0,...,w_n} \sum_{i=1}^{k} (f(x^{(i)}) - y^{(i)}) \tag{3.16}$$

Then, we have $f(x)$ as the returned value of the k-nn.

3.2.1 Speeding up k-NN

Here we discuss techniques to reduce the memory usage of the K-nn algorithm. For the case where there are two features, we may use Voronoi diagram, which can be computed in $O(m \log m)$. When there are more than two features, the Voronoi diagram is of size $O(m^{N/2})$.

There are two general strategies for alleviating this weakness. The first one is to avoid retaining every training instance (edited nearest neighbor) and the second is to use a smart data structure to look up nearest neighbors (e.g. a k-d tree).

Edited instance-based learning is to select a subset of the instances that still provide accurate classifications. It can be either incremental deletion or incremental growth. Incremental deletion starts with all training data in the memory, and then remove an instance (x, y) if other training instance provides correct classification for (x, y). Incremental growth starts with an empty memory, and add an instance (x, y) if other training instance in memory do not provide correct classification for (x, y).

In the following, we introduce k-d tree. A k-d tree is similar to a decision tree except that each internal node stores one instance. The tree splits on the median value of the feature having the highest variance.

Once we have a k-d tree, instead of going through all instances for a classification, we can apply a search algorithm to find the nearest neighbors. Algorithm 5 presents a candidate algorithm.

Algorithm 5: QueryKdTree(T, x), where T is a k-d tree and x is an instance

```
Queue = {}
bestDist = ∞
Queue.push(root,0)
while Queue is not empty do
    (node,bound) = Queue.pop()
    if bound ≥ bestDist then
        return bestNode.instance
    end
    dist = distance(x, node.instance)
    if dist < bestDist then
        bestDist = dist
        bestNode = node
    end
    if x[node.feature]-node.threshold > 0 then
        Queue.push(node.left, x[node.feature] - node.threshold)
        Queue.push(node.right, 0)
    else
        Queue.push(node.left, 0)
        Queue.push(node.right, node.threshold - x[node.feature])
    end
    return bestNode.instance
end
```

3.2.2 Classification Probability

For a classification task, K-nn can only return the label according to Equation (3.10). In cases where the probability of classification is needed, there are ad hoc ways for this purpose. For example, we can let

$$P(c|\mathbf{x}) = \frac{k_c + s}{k + |C|s} \tag{3.17}$$

for all $c \in C$, where k_c is the number of instances in the k neighbors that are with label c, and s is a small positive constant that is used to avoid $P(c) = 0$ for those classes c that do not appear in the k neighbors.

Alternatively, let d_c be the average distance from $\mathbf{x}$ to the instances with label c in the k neighbors of $\mathbf{x}$, we can let

$$P(c|\mathbf{x}) = \frac{\exp(-d_c) + s}{\sum_{c \in C} \exp(-d_c) + |C|s} \tag{3.18}$$

Note that, we have $d_c = \infty$ (and thus $\exp(-d_c) = 0$) if there is no instance of label c in the k neighbors.

We remark that, the probability $P(c|\mathbf{x})$ in Equation (3.17) is actually piece-wise linear over the changing of data instance $\mathbf{x}$, because it relies on the number of instances k_c. On the other hand, the one in Equation (3.18) is smoother by considering the average distance d_c.

3.2.3 Adversarial Example

The following is a heuristic algorithm based on the definition of probability as in Equation (3.18). Assume that we have a data instance with label $(\mathbf{x}, y)$ and want to find another one $(\mathbf{x}', y')$ that is close to $\mathbf{x}$. The general idea is to move the instance $\mathbf{x}$ gradually along the direction where the probability of being classified as y decreases the most, until the class change.

First of all, we define

$$\mathbf{x}_i^s = \begin{cases} \mathbf{x} + \epsilon_i & \text{if s=+} \\ \mathbf{x} - \epsilon_i & \text{if s=-} \end{cases} \tag{3.19}$$

where ϵ_i is a 0-vector except for the entry for feature X_i, which has value $\epsilon > 0$. Then, the gradient along the direction defined by X_i and $s \in \{+, -\}$ is as follows:

$$\nabla_i^s f(\mathbf{x}) = \frac{f(\mathbf{x}_i^s) - f(\mathbf{x})}{||\mathbf{x} - \mathbf{x}_i^s||} \tag{3.20}$$

Then, for the next step, we move $\mathbf{x}$ along the following direction to $\mathbf{x}'$:

$$\arg\min_{i,s} \nabla_i^s f(\mathbf{x}) \tag{3.21}$$

and check if there is a misclassification on $\mathbf{x}'$. We repeat the above move until there is a misclassification.

Is $\mathbf{x}'$ an adversarial example? Yes, because the final $\mathbf{x}'$ is obtained by following a sequence of changes until witnessing a misclassification.

Is this approach complete? Unfortunately, the above algorithm is incomplete.

Sub-optimality The resulting $\mathbf{x}'$ does not necessarily be the optimal solution to the optimisation problem described in Equation (2.16).

3.2.4 Practicals

For K-nn, we have the following code to **simpleML.py** which assumes $k = 3$.

```
from sklearn import datasets
iris = datasets.load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
    test_size=0.20)
```

```
from sklearn.neighbors import KNeighborsClassifier
neigh = KNeighborsClassifier(n_neighbors=3)
neigh.fit(X_train, y_train)
print("Training accuracy is %s"% neigh.score(X_train,y_train))
print("Test accuracy is %s"% neigh.score(X_test,y_test))
```

```
print("Labels of all instances:\n%s"%y_test)
y_pred = neigh.predict(X_test)
print("Predictive outputs of all instances:\n%s"%y_pred)

from sklearn.metrics import classification_report, confusion_matrix
print("Confusion Matrix:\n%s"%confusion_matrix(y_test, y_pred))
print("Classification Report:\n%s"%classification_report(y_test, y_pred))
```

```
import numpy as np
from math import sqrt
from collections import Counter

# Implement kNN in details
def kNNClassify(K, X_train, y_train, X_predict):
    distances = [sqrt(np.sum((x - X_predict)**2)) for x in X_train]
    sort = np.argsort(distances)
    topK = [y_train[i] for i in sort[:K]]
    votes = Counter(topK)
    y_predict = votes.most_common(1)[0][0]
    return y_predict

def kNN_predict(K, X_train, y_train, X_predict, y_predict):
    acc = 0
    for i in range(len(X_predict)):
        if y_predict[i] == kNNClassify(K, X_train, y_train, X_predict[i]):
            acc += 1
    print(acc/len(X_predict))

print("Training accuracy is ", end='')
kNN_predict(3, X_train, y_train, X_train, y_train)
print("Test accuracy is ", end='')
kNN_predict(3, X_train, y_train, X_test, y_test)
```

```
import itertools
import copy

# Attack on KNN
def kNN_attack(K, X_train, y_train, X_predict, y_predict):
    m = np.diag([0.5,0.5,0.5,0.5])*4
```

```
    flag = True
    for i in range(1,5):
        for ii in list(itertools.combinations([0,1,2,3],i)):
            delta = np.zeros(4)
            for jj in ii:
                delta += m[jj]

            if y_predict != kNNClassify(K, X_train, y_train, copy.deepcopy(
    X_predict)+delta):
                X_predict += delta
                flag = False
                break

            if y_predict != kNNClassify(K, X_train, y_train, copy.deepcopy(
    X_predict)-delta):
                X_predict -= delta
                flag = False
                break
        if not flag:
            break

    print('attack data: ',X_predict)
    print('predict label: ',kNNClassify(K, X_train, y_train, X_predict))

X_test_ = X_test[0]
y_test_ = y_test[0]
print('original data: ', X_test_)
print('original label: ', y_test_)
kNN_attack(3, X_train, y_train, X_test_, y_test_)
```

3.3 Linear Regression

Given a set of training data $D = \{(\mathbf{x}^{(i)}, y^{(i)}) | i \in \{1, ..., m\}\}$ sampled identically and independently from a distribution $\mathcal{D}$, linear regression assumes that the relationship between the label Y and the n-dimensional variable X is linear. More specifically, it is assume that the hypothesis space $\mathcal{H}$ is within linear models.

Therefore, the goal is to find a parameterised function

$$f_{\mathbf{w}}(x) = \mathbf{w}^T\mathbf{x} \tag{3.22}$$

that minimises the following L_2 loss (or mean square error)

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{w}^T\mathbf{x}^{(i)} - y^{(i)})^2 \tag{3.23}$$

Note that, $\mathbf{w}^T\mathbf{x}^{(i)} - y^{(i)}$ represents the loss of the instance $\mathbf{x}^{(i)}$. If written in matrix form, it is

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{w}^T\mathbf{x}^{(i)} - y^{(i)})^2 = \frac{1}{m}||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2 \tag{3.24}$$

Variant: Linear regression with bias It is possible that we may consider the function f with bias term:

$$f_{\mathbf{w},\mathbf{b}}(x) = \mathbf{w}^T\mathbf{x} + \mathbf{b} \tag{3.25}$$

To handle this case, we can reduce it to the case without bias by letting

$$\mathbf{w}' = [\mathbf{w}; \mathbf{b}], \mathbf{x}' = [\mathbf{x}; 1] \tag{3.26}$$

Then, we have

$$f_{\mathbf{w},\mathbf{b}}(x) = \mathbf{w}^T\mathbf{x} + \mathbf{b} = (\mathbf{w}')^T(\mathbf{x}') \tag{3.27}$$

Intuitively, every instance is extended with one more feature whose value is always 1, and we assume that we already know the weight for this feature in $f_{\mathbf{w},\mathbf{b}}$, which is b.

Variant: Linear regression with lasso penalty We may also be interested in adapting the loss, e.g., consider the following loss

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{w}^T\mathbf{x}^{(i)} - y^{(i)})^2 + \lambda||\mathbf{w}||_1 \tag{3.28}$$

where the lasso penalty term $||\mathbf{w}||_1$ is to encourage the sparsity of the weights.

3.3.1 Linear Classification

To consider classification task where $y^{(i)}$'s are labels instead of regression values, a natural attempt is to change the hypothesis class $\mathcal{H}$ from the set of linear models to a set of piecewise linear models. For simplicity, assume that we are working with binary classification, i.e., $V(Y) = \{0, 1\}$. Then, the hypothesis class $\mathcal{H}$ is parameterised over $\mathbf{w}$ such that

$$f_{\mathbf{w}}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w}^T\mathbf{x} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.29}$$

What is the corresponding optimisation problem? With the hypothesis class $\mathcal{H}$, the classification problem is to find a parameterised function

$$f'_{\mathbf{w}}(x) = \mathbf{w}^T\mathbf{x} \tag{3.30}$$

that minimises the following loss

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m} I[step(\mathbf{w}^T\mathbf{x}^{(i)}), y^{(i)}] \tag{3.31}$$

where $step$ function is defined as: $step(m) = 1$ when $m > 0$ and $step(m) = 0$ otherwise, and $I(\hat{y}, y)$ is the 0-1 loss, i.e., $I(\hat{y}, y) = 0$ when $\hat{y} = y$ and $I(\hat{y}, y) = 1$ otherwise.

Difficulty of finding optimal solution Unfortunately, the finding of optimal solution to the optimisation problem in Equation (3.30) and (3.31) is NP-hard.

3.3.2 Logistic Regression

Given the above natural – but nevertheless naive – attempt does not necessarily lead to a good method for classification problem, it is needed to consider other options, such as logistic regression.

Linear regression attacks the classification problem by attempting to make the regression values as probability values. That is, if the return of $f_{\mathbf{w}}(\mathbf{x})$ is a probability value of classifying $\mathbf{x}$ as 0, then we are easy to infer the classification by using the regression value of $\mathbf{x}$. This, however, is not straightforward as the linear regression may return values outside of [0,1].

Sigmoid function First of all, we need to make sure that the regression value within [0,1]. This can be done through applying the sigmoid function

$$\sigma(a) = \frac{1}{1 + \exp(-a)} \tag{3.32}$$

which has the domain $(0, 1)$. Therefore, we can now update Equation (3.31) into

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(\sigma(\mathbf{w}^T\mathbf{x}^{(i)}) - y^{(i)})^2 \tag{3.33}$$

However, even so, it is still unclear whether $\sigma(\mathbf{w}^T\mathbf{x}^{(i)})$ represents the probability value.

Force $\sigma(\mathbf{w}^T\mathbf{x}^{(i)})$ into probability value To achieve this, we make the following interpretation

$$\begin{aligned} P_{\mathbf{w}}(y=1|\mathbf{x}) &= \sigma(\mathbf{w}^T\mathbf{x}^{(i)}) \\ P_{\mathbf{w}}(y=0|\mathbf{x}) &= 1 - P_{\mathbf{w}}(y=1|\mathbf{x}) \quad = 1 - \sigma(\mathbf{w}^T\mathbf{x}^{(i)}) \end{aligned} \tag{3.34}$$

Then, we can update the loss (Equation (3.31)) into

$$\hat{L}(f_{\mathbf{w}}) = -\frac{1}{m}\sum_{i=1}^{m} \log P_{\mathbf{w}}(y^{(i)}|\mathbf{x}^{(i)}) \tag{3.35}$$

By applying Equation (3.34), we have

$$\begin{aligned} \hat{L}(f_{\mathbf{w}}) = & -\frac{1}{m}\sum_{y^{(i)}=1} \log\sigma(\mathbf{w}^T\mathbf{x}^{(i)}) - \frac{1}{m}\sum_{y^{(i)}=0} \log[1-\sigma(\mathbf{w}^T\mathbf{x}^{(i)})] \\ = & -\frac{1}{m}\sum_{(\mathbf{x},y)\in D} y\log\sigma(\mathbf{w}^T\mathbf{x}) + (1-y)\log[1-\sigma(\mathbf{w}^T\mathbf{x})] \end{aligned} \tag{3.36}$$

3.3.3 Adversarial Examples

Considering a binary classification problem, i.e., $y \in \{+1, -1\}$ for any data instance $\mathbf{x}$, the loss for a single instance $(\mathbf{x}, y)$ as in Equation (3.36) is

$$\hat{L}(f_{\mathbf{w}}, (\mathbf{x}, y)) = \quad -y\log\sigma(\mathbf{w}^T\mathbf{x}) - (1-y)\log[1-\sigma(\mathbf{w}^T\mathbf{x})] \tag{3.37}$$

The computation of adversarial example is to solve the following optimisation problem:

$$\text{maximise}_{||\boldsymbol{\delta}||\leq\epsilon} \quad \hat{L}(f_{\mathbf{w}}, (\mathbf{x}+\boldsymbol{\delta}, y)) \tag{3.38}$$

which finds the greatest loss within the constraint over the perturbation $\boldsymbol{\delta}$. We note that, by Equation (3.37), this is equivalent to

$$\text{minimise}_{||\boldsymbol{\delta}||\leq\epsilon} \quad y\mathbf{w}^T\boldsymbol{\delta} \tag{3.39}$$

Now, considering the L_∞ norm, i.e., $||\boldsymbol{\delta}||_\infty \leq \epsilon$, the optimal solution to Equation (3.39) is

$$\boldsymbol{\delta}^* = \quad -y\epsilon sign(\mathbf{w}) \tag{3.40}$$

where $sign$ is the sign function extracting the sign of real numbers, such that

$$sign(w) = \begin{cases} -1 & w < 0 \\ 0 & w = 0 \\ 1 & w > 0 \end{cases} \tag{3.41}$$

and $sign(\mathbf{w})$ is the application of $sign$ on all entries of $\mathbf{w}$.

3.3.4 Practicals

For logistic regression, we add the following code to the **simpleML.py** file:

```
from sklearn import datasets
iris = datasets.load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data[:100], iris.
    target[:100], test_size=0.20)
```

```
from sklearn.linear_model import LogisticRegression
reg = LogisticRegression(solver='lbfgs', max_iter=10000)
reg.fit(X_train, y_train)
print("Training accuracy is %s"% reg.score(X_train,y_train))
print("Test accuracy is %s"% reg.score(X_test,y_test))
```

```
print("Labels of all instances:\n%s"%y_test)
y_pred = reg.predict(X_test)
print("Predictive outputs of all instances:\n%s"%y_pred)

from sklearn.metrics import classification_report, confusion_matrix
print("Confusion Matrix:\n%s"%confusion_matrix(y_test, y_pred))
print("Classification Report:\n%s"%classification_report(y_test, y_pred))
```

```
import numpy as np
from sklearn.metrics import accuracy_score
def sigmoid(x):
    z = 1 / (1 + np.exp(-x))
    return z

def add_b(dataMatrix):
    dataMatrix = np.column_stack((np.mat(dataMatrix),np.ones(np.shape(
    dataMatrix)[0])))
    return dataMatrix

def LogisticRegression_(x_train,y_train,x_test,y_test,alpha = 0.001 ,
    maxCycles = 500):
    x_train = add_b(x_train)
    x_test = add_b(x_test)
    y_train = np.mat(y_train).transpose()
    y_test = np.mat(y_test).transpose()
    m,n = np.shape(x_train)
    weights = np.ones((n,1))
    for i in range(0,maxCycles):
        h = sigmoid(x_train*weights)
        error = y_train - h
        weights = weights + alpha * x_train.transpose() * error

    y_pre = sigmoid(np.dot(x_train, weights))
    for i in range(len(y_pre)):
        if y_pre[i] > 0.5:
            y_pre[i] = 1
        else:
            y_pre[i] = 0
    print("Train accuracy is %s"% (accuracy_score(y_train, y_pre)))
```

```
    y_pre = sigmoid(np.dot(x_test, weights))
    for i in range(len(y_pre)):
        if y_pre[i] > 0.5:
            y_pre[i] = 1
        else:
            y_pre[i] = 0
    print("Test accuracy is %s"% (accuracy_score(y_test, y_pre)))

weights = LogisticRegression_(X_train, y_train,X_test,y_test)
```

```
import itertools
import copy

# Attack on LogisticRegression
def LogisticRegression_attack(weights, X_predict, y_predict):
    X_predict = add_b(X_predict)
    m = np.diag([0.5,0.5,0.5,0.5])*4
    flag = True
    for i in range(1,5):
        for ii in list(itertools.combinations([0,1,2,3],i)):
            delta = np.zeros(4)
            for jj in ii:
                delta += m[jj]
            delta = np.append(delta, 0.)

            y_pre = sigmoid(np.dot(copy.deepcopy(X_predict)+delta, weights))
            if y_pre > 0.5:
                y_pre = 1
            else:
                y_pre = 0
            if y_predict != y_pre:
                X_predict += delta
                flag = False
                break

            y_pre = sigmoid(np.dot(copy.deepcopy(X_predict)-delta, weights))
            if y_pre > 0.5:
                y_pre = 1
            else:
                y_pre = 0
            if y_predict != y_pre:
                X_predict -= delta
                flag = False
                break
        if not flag:
            break

    y_pre = sigmoid(np.dot(X_predict, weights))
    if y_pre > 0.5:
        y_pre = 1
    else:
        y_pre = 0
    print('attack data: ', X_predict[0,:-1])
```

```
    print('predict label: ', y_pre)

X_test_ = X_test[0:1]
y_test_ = y_test[0]
print('original data: ', X_test_)
print('original label: ', y_test_)
LogisticRegression_attack(weights, X_test_, y_test_)
```

3.4 Gradient Descent

Most machine learning algorithms involve optimization, and gradient descent is an effective method to find sub-optimal solutions.

Derivative of a function Given a function $\hat{L}(x)$, its derivative is the slope of $\hat{L}(x)$ at point x, written as $\hat{L}'(x)$. It specifies how to scale a small change in input to obtain a corresponding change in the output. Specifically,

$$\hat{L}(x+\epsilon) \approx \hat{L}(x) + \epsilon \hat{L}'(x) \tag{3.42}$$

Moreover, define the sign function as the following:

$$sign(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{otherwise} \end{cases} \tag{3.43}$$

Then, we have that

$$\hat{L}(x - \epsilon sign(\hat{L}'(x))) < \hat{L}(x) \tag{3.44}$$

Therefore, we can reduce $\hat{L}(x)$ by moving x in small steps with opposite sign of derivative.

Gradient Gradient generalizes notion of derivative where derivative is with respect to a vector.

$$\nabla_{\mathbf{x}}(\hat{L}(\mathbf{x})) = (\frac{\partial \hat{L}(\mathbf{x})}{\partial x_1}, ..., \frac{\partial \hat{L}(\mathbf{x})}{\partial x_n}) \tag{3.45}$$

where the partial derivative $\frac{\partial \hat{L}(\mathbf{x})}{\partial x_i}$ measures how the function $\hat{L}$ changes when only variable x_i increases at point $\mathbf{x}$.

Critical points are where every element of the gradient is equal to zero, i.e.,

$$\nabla_{\mathbf{x}}(\hat{L}(\mathbf{x})) = 0 \equiv \begin{cases} \frac{\partial \hat{L}(\mathbf{x})}{\partial x_1} = 0 \\ ... \\ \frac{\partial \hat{L}(\mathbf{x})}{\partial x_n} = 0 \end{cases} \tag{3.46}$$

Gradient descent on linear regression Given Equation (3.24), we have that

$$\begin{aligned} & \nabla_{\mathbf{w}} \hat{L}(f_{\mathbf{w}}) \\ = \; & \nabla_{\mathbf{w}} \tfrac{1}{m} ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2 \\ = \; & \nabla_{\mathbf{w}} [(\mathbf{X}\mathbf{w} - \mathbf{y})^T (\mathbf{X}\mathbf{w} - \mathbf{y})] \\ = \; & \nabla_{\mathbf{w}} [\mathbf{w}^T \mathbf{X}^T \mathbf{X}\mathbf{w} - 2\mathbf{w}^T \mathbf{X}^T \mathbf{y} + \mathbf{y}^T \mathbf{y}] \\ = \; & 2\mathbf{X}^T \mathbf{X}\mathbf{w} - 2\mathbf{X}^T \mathbf{y} \end{aligned} \tag{3.47}$$

Therefore, we can follow the following gradient descent algorithm to solve linear regression:

1. Set step size ϵ and tolerance δ to small positive numbers
2. While $||\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{y}||_2 > \delta$ do

$$\mathbf{x} \leftarrow \mathbf{x} - \epsilon(\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{y}) \tag{3.48}$$

3. Return $\mathbf{x}$ as a solution

Analytical solution on linear regression We may be able to avoid iterative algorithm and jump to the critical point by solving the following equation for $\mathbf{x}$:

$$\nabla_{\mathbf{w}}\hat{L}(f_{\mathbf{w}}) = 0 \tag{3.49}$$

By Equation (3.47), we have that

$$\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{y} = 0 \tag{3.50}$$

That is,

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{3.51}$$

3.5 Naive Bayes

Naive Bayes is a probabilistic algorithm that can be used for classification problems. Albeit simple and intuitive, naive Bayes performs very well in many practical applications such as spam filter email application.

Let Y be a random variable representing the label and $X_1, ..., X_n$ be random variables representing the n input features, respectively. The classification problem can be expressed as

$$\arg\max_{y \in V(Y)} P(Y = y | X_1 = x_1, ..., X_n = x_n) \tag{3.52}$$

which is to find the label y with the maximum probability given the instance $\mathbf{x} = (x_1, ..., x_n)$. This can be done by first estimating the following conditional probability table from the training dataset D

$$P(Y | X_1, ..., X_n) \tag{3.53}$$

and then apply Equation (3.52) on $\mathbf{x}$.

Bayes Theorem suggests that we have

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \tag{3.54}$$

where $P(Y)$ is the prior, $P(X)$ is the evidence, $P(X|Y)$ is the likelihood function, and $P(Y|X)$ is the posterior. By Bayes theorem, we have that

$$P(Y | X_1, ..., X_n) = \frac{P(X_1, ..., X_n | Y)P(Y)}{P(X_1, ..., X_n)} \tag{3.55}$$

That is, the computation of the conditional probability table $P(Y|X_1, ..., X_n)$ can be reduced to the computation of three tables $P(X_1, ..., X_n|Y)$, $P(X_1, ..., X_n)$, and $P(Y)$. Furthermore, noting that $P(X_1, ..., X_n)$ – representing the data distribution – is fixed for any $y \in V(Y)$, we have that

$$\begin{array}{rl} & \arg\max_{y \in V(Y)} P(Y = y | X_1 = x_1, ..., X_n = x_n) \\ \propto & \arg\max_{y \in V(Y)} P(X_1 = x_1, ..., X_n = x_n | Y = y)P(Y = y) \end{array} \tag{3.56}$$

Therefore, for the classification problem, it is sufficient to compute two probability tables: $P(X_1, ..., X_n|Y)$ and $P(Y)$.

Estimation of $P(Y)$ can be done by letting

$$P(Y = y) = \frac{\text{Number of instances whose label is } y}{\text{Number of all instances}} \tag{3.57}$$

for all $y \in V(Y)$. Can we use similar expression to estimate $P(X_1, ..., X_n|Y)$? Yes, we can, but it is not scalable.

Difficulty of estimating $P(X_1, ..., X_n|Y)$ directly Without loss of generality, we assume that all random variables are Boolean. Therefore, to estimation $P(Y)$, we need only compute once the Expression (3.57). If we want to estimate $P(X_1, ..., X_n|Y)$ with a similar expression as Expression (3.57), we will need $(2^n - 1) \times 2$ computations – an exponential computation.

Assumption Naive Bayes assumes that the input features $X_1, ..., X_n$ are conditionally independent given the label Y, i.e.,

$$P(X_1, ..., X_n|Y) = \prod_{i=1}^{n} P(X_i|Y) \tag{3.58}$$

With this assumption, $P(X_1, ..., X_n|Y)$ can be estimated by computing n tables $P(X_i|Y)$, each of which requires 2 computations. That is, instead of conducting $(2^n - 1) \times 2$ computations, we now need $2n$ computations – a linear time computation.

With Assumption (3.58), we have the Naive Bayes expression:

$$\begin{array}{rl} & \arg\max_{y \in V(Y)} P(Y = y|X_1 = x_1, ..., X_n = x_n) \\ \propto & \arg\max_{y \in V(Y)} P(Y = y) \prod_{i=1}^{n} P(X_i = x_i|Y = y) \end{array} \tag{3.59}$$

Algorithm The Naive Bayes classification algorithm proceeds in the following three steps:

1. For each value $y_k \in V(Y)$, we estimate

$$\pi_k = P(Y = y_k) \tag{3.60}$$

2. For each value x_{ij} of each feature X_i and each $y_k \in V(Y)$, we estimate

$$\theta_{ijk} = P(X_i = x_{ij}|Y = y_k) \tag{3.61}$$

3. Given a new input $\mathbf{x}_{new} = (x_1^{new}, ..., x_n^{new})$, we classify it by letting

$$\hat{y_{new}} \leftarrow \arg\max_{y_k} P(Y = y_k) \prod_i P(X_i = x_i^{new}|Y = y_k) \tag{3.62}$$

For Continuous Features The above works with categorical features. For continuous features, we can discretise the values of the feature into a set of categorical values, so that the above method works. We may also consider making assumption on the distribution of the continuous features.

For example, let $P(X_i|Y)$ be a Gaussian probability density function, i.e.,

$$P(X_i|Y) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-(X_i - \mu)^2}{2\sigma^2}\right) \tag{3.63}$$

such that the Gaussian distribution has the expected value μ and variance σ^2. We can estimate μ with

$$\mu = \frac{\sum_{(\mathbf{x},y) \in D} X_i(\mathbf{x})}{|D|} \tag{3.64}$$

where $X_i(\mathbf{x})$ returns the value of feature X_i in instance $\mathbf{x}$, and σ^2

$$\sigma^2 = \frac{1}{|D|-1} \sum_{(\mathbf{x},y)\in D} (X_i(\mathbf{x}) - \mu)^2 \tag{3.65}$$

Then we can have compute $P(X_i = x_i^{new}|Y = y_k)$ by applying Equation (3.63). Afterwards, we can get the classification by Equation (3.62).

3.5.1 Practicals

We can use Gaussian naive Bayes to **simpleML.py** as follows:

```
from sklearn import datasets
iris = datasets.load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
    test_size=0.20)
```

```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(X_train, y_train)
print("Training accuracy is %s"% gnb.score(X_train,y_train))
print("Test accuracy is %s"% gnb.score(X_test,y_test))
```

```
print("Labels of all instances:\n%s"%y_test)
y_pred = gnb.predict(X_test)
print("Predictive outputs of all instances:\n%s"%y_pred)

from sklearn.metrics import classification_report, confusion_matrix
print("Confusion Matrix:\n%s"%confusion_matrix(y_test, y_pred))
print("Classification Report:\n%s"%classification_report(y_test, y_pred))
```

```
from collections import Counter

class GaussianNB_:
    def __init__(self):
        self.prior = None
        self.avgs = None
        self.vars = None
        self.n_class = None

    def _get_prior(self, y):
        cnt = Counter(y)
        prior = np.array([cnt[i] / len(y) for i in range(len(cnt))])
        return prior

    def _get_avgs(self, X, y):
        return np.array([X[y == i].mean(axis=0) for i in range(self.n_class)
    ])

    def _get_vars(self, X, y):
        return np.array([X[y == i].var(axis=0) for i in range(self.n_class)])

    def _get_likelihood(self, row):
```

```
        return (1 / np.sqrt(2 * np.pi * self.vars) * np.exp(-(row - self.avgs
    )**2 / (2 * self.vars))).prod(axis=1)

    def fit(self, X, y):
        self.prior = self._get_prior(y)
        self.n_class = len(self.prior)
        self.avgs = self._get_avgs(X, y)
        self.vars = self._get_vars(X, y)

    def predict_prob(self, X):
        likelihood = np.apply_along_axis(self._get_likelihood, axis=1, arr=X)
        probs = self.prior * likelihood
        probs_sum = probs.sum(axis=1)
        return probs / probs_sum[:, None]

    def predict(self, X):
        return self.predict_prob(X).argmax(axis=1)

def get_acc(y, y_hat):
    a = 0
    for i in range(len(y)):
        if y[i]==y_hat[i]:
            a += 1
    return a/len(y)

clf = GaussianNB_()
clf.fit(X_train, y_train)

y_hat = clf.predict(X_train)
acc = get_acc(y_train, y_hat)
print("Train accuracy is %s"% acc)

y_hat = clf.predict(X_test)
acc = get_acc(y_test, y_hat)
print("Test accuracy is %s"% acc)
```

```
import itertools

# Attack on GaussianNB
def GaussianNB_attack(clf, X_predict, y_predict):
    m = np.diag([0.5,0.5,0.5,0.5])*4
    flag = True
    for i in range(1,5):
        for ii in list(itertools.combinations([0,1,2,3],i)):
            delta = np.zeros(4)
            for jj in ii:
                delta += m[jj]

            y_pre = clf.predict(copy.deepcopy(X_predict)+delta)
            if y_predict != y_pre:
                X_predict += delta
                flag = False
                break

```

```
            y_pre = clf.predict(copy.deepcopy(X_predict)-delta)
            if y_predict != y_pre:
                X_predict -= delta
                flag = False
                break
        if not flag:
            break

    print('attack data: ', X_predict)
    print('predict label: ', clf.predict(copy.deepcopy(X_predict)))

X_test_ = X_test[0:1]
y_test_ = y_test[0]
print('original data: ', X_test_)
print('original label: ', y_test_)
GaussianNB_attack(clf, X_test_, y_test_)
```

Index	Sepal Length	Sepal Width	Pedal Length	Pedal Width	Iris Class
1	5	3	1	0.5	0
2	4	3	1	0.5	0
3	4	3	1	0.5	0
4	5	3	1	0.5	0
5	4	3	1	0.5	0
6	7	3	4	1	1
7	6	3	4	1	1
8	6	3	4	1	1
9	4	2	3	1	1
10	6	3	6	2	2
11	5	2	5	2	2
12	7	3	5	2	2
13	5	2	5	2	2
14	6	2	5	1	2

Table 3.1: Part of **Iris** dataset

3.6 Exercises

Question 1. *Understand the basic idea of random forest by conducting research on the literature, and implement a random forest algorithm based on the decision tree algorithm to see if random forest performs better than decision tree on* ***Iris*** *dataset.*

Question 2. *Following the last question, please give an adversarial attack algorithm for random forest.*

Question 3. *Understand the basic idea of Bayesian linear regression by conducting research on the literature, and implement a Bayesian linear regression algorithm to compare its performance with linear regression.*

Question 4. *Write a program for the adversarial attack for logistic regression.*

Question 5.

4 Chapter 4: Deep Learning

This chapter will focus on the modern deep learning, explaining a few fundamental aspects, including perceptron and why we need multi-layer structures, how the convolutional neural networks extract features layer by layer, the back-propagation learning algorithm, and the functional layers of convolutional neural networks. We will also introduce adversarial attacks in deep learning.

4.1 Perceptron

Biological vs Artificial Neurons A human brain contains billions of neurons, which – as shown in Figure 4.1 – are inter-connected nerve cells that are involved in processing and transmitting chemical and electrical signals. Neurons use dendrites as branches to receive information from other neurons. The received information is processed by cell body or Soma. A neuron sends information to other neurons through a cable – called axon – and the synapse which connects an axon with other neurons' dendrites.

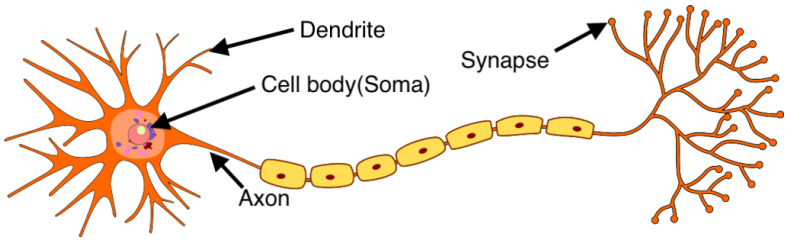


Figure 4.1: Biological Neuron (from Wikipedia)

In 1943, Warren McCullock and Walter Pitts published a simplified brain cell, called McCullock-Pitts (MCP) neuron. As shown in Figure 4.2, an artificial neuron represents a nerve cell as a simple logic gate with binary outputs. It takes inputs, weighs them separately, sums them up, and passes this sum through a nonlinear function to produce output.

Table 4.1 is a brief conceptual mapping between biological and artificial neurons.

Biological Neuron	Artificial Neuron
dentrites	input
cell body (soma)	node
axon	output
synapse	weights

Table 4.1: Mapping of Concepts between Biological and Artificial Neurons

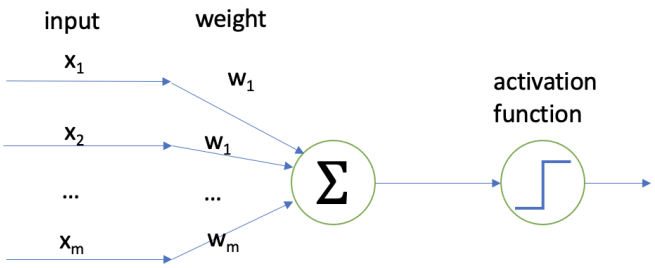


Figure 4.2: Artificial Neuron

Learning Algorithm of Perceptron A perceptron is an artificial neuron that does certain computations (such as detect features or execute business intelligence) on the input data. Perceptron was introduced by Frank Rosenblatt in 1957, when he proposed a perceptron learning rule based on the original MCP neuron. In July 1958, an IBM 704 – a 5-ton computer with the size of a room – was fed a series of punch cards. After 50 trials, the computer taught itself to distinguish cards marked on the left from cards marked on the right [10]. It was a demonstration of the "perceptron", and was "the first machine which is capable of having an original idea," according to its creator, Frank Rosenblatt.

A perceptron learning algorithm is a supervised learning of binary classifiers. The binary classifier processes an input $\mathbf{x} = (x_1, ..., x_m)$ as follows:

1. Use one weight w_i per feature X_i;
2. Multiply weights w_i with the respective input features x_i of $\mathbf{x}$, and add bias w_0;
3. If the result is greater than a pre-specified threshold, return 1. Otherwise, return 0.

The weights $w_1, ..., w_m$ and bias w_0 of the binary classifier need to be learned. Given a set D of training instances, the learning algorithm proceeds as follows:

- Initialize weights randomly,
- Take one sample $(\mathbf{x}_i, y_i) \in D$ and make a prediction $\hat{y}_i$,
- For erroneous predictions, update weights with the following rules:
  - If the output is $\hat{y}_i = 0$ but the label is $y_i = 1$, increase the weights.
  - If the output is $\hat{y}_i = 1$ but the label is $y_i = 0$, decrease the weights.

  that is, we let

$$w_i = w_i + \Delta w_i \quad \text{such that} \quad \Delta w_i = \eta(y_i - \hat{y}_i)\mathbf{x}_i \tag{4.1}$$

  where $\eta \ll 1$ is a constant representing the learning rate.

4.1.1 Expressivity of Perceptron

While simple, perceptron is the foundation of modern deep learning, which uses a network of perceptrons where there are multiple layers and there are multiple neurons per layer. Why do we need multi-layer perceptron (MLP) instead of just a single perceptron? This is a question related to the expressivity of a single perceptron. Actually, as we will show below, a single perceptron can express some useful functions but cannot do well for others.

Linearly separable function As perceptron is a linear classifier, it is able to work well on all linearly separable datasets, i.e., classify instances that can be separated with a linear function.

X_1	X_2	**y**
0	0	0
0	1	0
1	0	0
1	1	1

Table 4.2: Truth table for logic $\wedge$ (And)

Example 25. *Table 4.2 presents an example dataset of four instances for the logic operator $\wedge$. Actually, we generalise the logic operator $\wedge$ to the following function.*

$$f_{\wedge}(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 > 0.5 \text{ and } x_2 > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

In Figure 4.3, we use squares to represent 0 and triangles to represent 1. By learning a perceptron, it is possible to get a linear separation function as exhibited in the figure. We use different colors to denote different areas in which the instances should be classified accordingly. In Figure 4.3, we also generate a random sample of 100 instances and use the learned perceptron to predict the instances with different colors (with an accuracy close to 1.0).

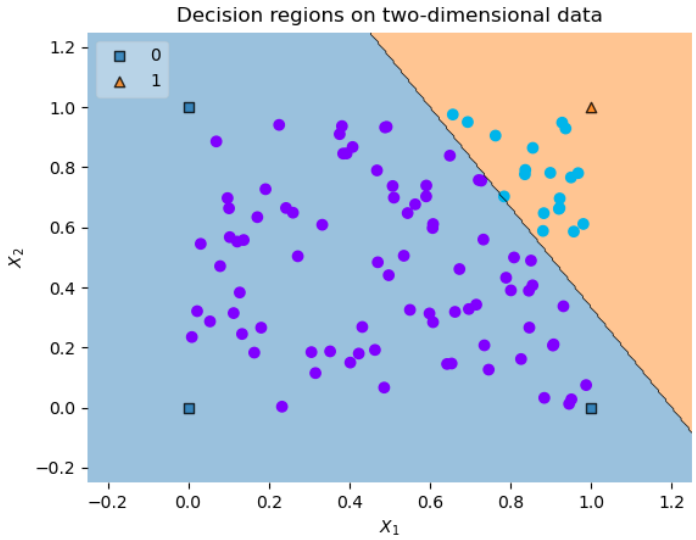


Figure 4.3: Visualisation of a perceptron learned from the data instances in Table 4.2

Example 26. *The other example is as shown in Table 4.3, presenting an example dataset of four instances for the logic operator $\vee$. Actually, we generalise the logic operator $\vee$ to the following*

function.

$$f_{\vee}(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 > 0.5 \text{ or } x_2 > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

X_1	X_2	**y**
0	0	0
0	1	1
1	0	1
1	1	1

Table 4.3: Truth table for logic $\vee$ (Or)

Figure 4.4 presents the visualisation of the learned perceptron. We can see that, the separating line is different from that of Figure 4.3. The separating lines in both figures are able to separate the data very well (with accuracy close to 1.0).

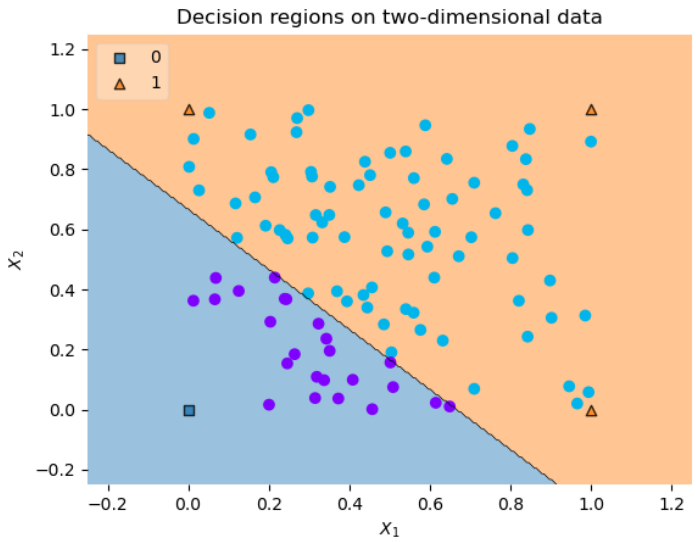


Figure 4.4: Visualisation of a perceptron learned from the data instances in Table 4.3

The above examples are all based on 2-dimensional dataset. The learned perceptrons are actually a line on the 2-dimensional space. When the dataset is d-dimensional, the perceptron is a d-dimensional hyper-plane.

Linearly inseparable function Unfortunately, not all datasets are linearly separable.

Example 27. *Table 4.4 is a dataset of four instances, which is generated from the following function:*

$$f_{\oplus}(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 > 0.5 \text{ and } x_2 < 0.5 \\ 1 & \text{if } x_1 < 0.5 \text{ and } x_2 > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

While we can still apply perceptron, the learned perceptron cannot reach high accuracy (∼ 0.5 accuracy).

X_1	X_2	**y**
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.4: Truth table for logic $\oplus$ (XOR)

Therefore, this example shows that the perceptron in itself does not have a sufficient expressiveness to work with complex functions. This contributes as the reason why we have to consider multiple layers and more neurons.

4.1.2 Multi-layer Perceptron

To deal with the XOR problem, a two-layer perceptron as shown in Figure 4.5 has been suggested.

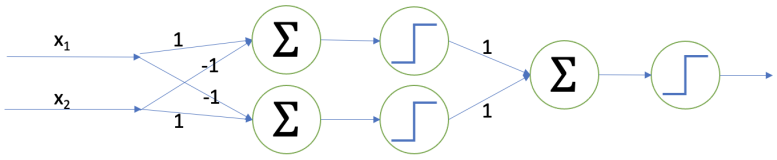


Figure 4.5: A two-layer perceptron to solve XOR problem

where the activation function is $ReLU(x) = \max(0, x)$.

If written in the matrix form, we have the following expressions for the training data, which shows that the above two-layer perceptron perfectly classifies the training data.

$$\begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ -1 & -1 \\ 1 & -1 \\ 0 & 0 \end{pmatrix} \text{ and } ReLU \begin{pmatrix} 0 & 0 \\ -1 & -1 \\ 1 & -1 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{pmatrix} \tag{4.5}$$

$$\begin{pmatrix}0 & 0\\ 0 & 1\\ 1 & 0\\ 0 & 0\end{pmatrix}\times\begin{pmatrix}1\\ 1\end{pmatrix}=\begin{pmatrix}0\\ 1\\ 1\\ 0\end{pmatrix}\quad\text{and}\quad ReLU\begin{pmatrix}0\\ 1\\ 1\\ 0\end{pmatrix}=\begin{pmatrix}0\\ 1\\ 1\\ 0\end{pmatrix} \tag{4.6}$$

4.1.3 Practicals

Train a perceptron First of all, we load and prepare datasets.

```
from sklearn import datasets
dataset = datasets.load_digits()
X = dataset.data
y = dataset.target

observations = len(X)
features = len(dataset.feature_names)
classes = len(dataset.target_names)
print("Number of Observations: " + str(observations))
print("Number of Features: " + str(features))
print("Number of Classes: " + str(classes))

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20)
```

Then, we can call **sklearn**'s library function to train a perceptron model.

```
from sklearn.linear_model import Perceptron

clf = Perceptron(tol=1e-3, random_state=0)
clf.fit(X_train, y_train)
print("Training accuracy is %s"% clf.score(X_train,y_train))
print("Test accuracy is %s"% clf.score(X_test,y_test))

print("Labels of all instances:\n%s"%y_test)
y_pred = clf.predict(X_test)
print("Predictive outputs of all instances:\n%s"%y_pred)

from sklearn.metrics import classification_report, confusion_matrix
print("Confusion Matrix:\n%s"%confusion_matrix(y_test, y_pred))
print("Classification Report:\n%s"%classification_report(y_test, y_pred))
```

Display class regions for Boolean functions In the following, we present how to generate the visualisation as in Figure 4.3 and Figure 4.4. First of all, we install a package **mlxtend**.

```
$ pip3 install mlxtend
```

Then, we need to load the data instances $\mathbf{X} = \{(0,0),(0,1),(1,0),(1,1)\}$ and their labels. Note that, in the below code, we use **logical_and**. You are able to use others such as **logical_or** and **logical_xor**.

```
import numpy as np
# Loading data
X_train = np.array([[0.0,0.0],[0.0,1.0],[1.0,0.0],[1.0,1.0]])
y_train = np.array(np.logical_and(X_train[:, 0] > 0.5, X_train[:, 1] > 0.5),
             dtype=int)
```

Once the data is loaded, we train a Perceptron, with initial parameters $\mathbf{w} = (1.5, 1.5)$. Note that, this is simply for the experiment, and it can be initialised to other values, or set as default by ignoring the parameter **coef_init**.

```
# Training a classifier
from sklearn.linear_model import Perceptron
clf = Perceptron(tol=1e-3, random_state=0)
clf.fit(X_train, y_train, coef_init=np.array([[1.5],[1.5]]))
```

We can print the final learned weights.

```
print(clf.coef_)
```

Finally, we can plot the regions for the classes.

```
# Plotting decision regions
from mlxtend.plotting import plot_decision_regions
plot_decision_regions(X_train, y_train, clf=clf, legend=2)
```

We may also generate a set of random points and plot them.

```
# Plotting randomly generated points
import matplotlib
import matplotlib.pyplot as plt
import random

n_sample = 100
X_test = np.array([[random.random() for i in range(2)] for j in range(
    n_sample)])
y_test = np.array(np.logical_and(X_test[:,0]>0.5,X_test[:,1]>0.5),dtype=int)
y_pred = clf.predict(X_test)
print(clf.score(X_test,y_test))
colors = matplotlib.cm.rainbow(np.linspace(0, 1, 5))
plt.scatter(X_test[:, 0],X_test[:, 1],color=[colors[i] for i in y_pred])
```

```
# Adding axes annotations
import matplotlib.pyplot as plt
plt.xlabel('X_1')
plt.ylabel('X_2')
plt.title('Decision regions on two-dimensional data')
plt.show()
```

4.2 Functional View

A deep neural network can be seen as a family of parametric, non-linear, and hierarchical representation learning functions. These learning functions are massively optimized with stochastic gradient descent (SGD) over some pre-specified objectives, such as the loss of a set of training instances. It is expected that, the learned functions will encode domain knowledge that are implicitly presented in the training instances.

4.2.1 Mappings between High-dimensional Spaces

Consider a feed-forward network as shown in Figure 4.6. Assume that it has $m + 1$ layers, where Layer-0 is the input layer, Layer-m is the output layer, and Layer-1 to Layer-$(m - 1)$ are the hidden layers.

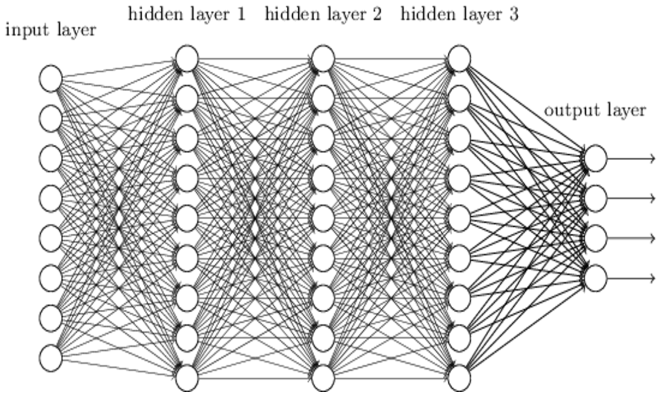


Figure 4.6: A 5 layer feed-forward network

Every layer is a function, so we have functions $f_1, ..., f_m$, for hidden layers and output layer, and because every function is parameterised, we use $\mathbf{W} = \{\mathbf{W}_1, ..., \mathbf{W}_m\}$ to denote their parameters. Based on these, a neural network can be written in a functional way as follows.

$$f_{\mathbf{W}}(\mathbf{x}; \mathbf{W}_1, ..., \mathbf{W}_m) = f_m(f_{m-1}(...f_1(\mathbf{x}; \mathbf{W}_1), \mathbf{W}_{m-1}); \mathbf{W}_m) \tag{4.7}$$

Alternatively, we may write

$$\begin{array}{rcccl} f_{\mathbf{W}}(\mathbf{x}) & = & \mathbf{v}_m & = & f_m(\mathbf{v}_{m-1}; \mathbf{W}_m) \\ & & \mathbf{v}_{m-1} & = & f_{m-1}(\mathbf{v}_{m-2}; \mathbf{W}_{m-1}) \\ & & ... & & \\ & & \mathbf{v}_2 & = & f_2(\mathbf{v}_1; \mathbf{W}_2) \\ & & \mathbf{v}_1 & = & f_1(\mathbf{v}_0; \mathbf{W}_1) \end{array} \tag{4.8}$$

where $\mathbf{v}_i$ is the output value of Layer-i. Note that, we have both $\mathbf{v}_i$ and $\mathbf{W}_i$ as vectors or matrices, because it is typical that there are many neurons per layer and the layer functions are parameterised with many parameters.

Take a closer look at Equation (4.8), given a function $\mathbf{v}_i = f_i(\mathbf{v}_{i-1}; \mathbf{W}_i)$, once the parameters $\mathbf{W}_i$ are learned, it is a transformation from $\mathbf{v}_{i-1}$ to $\mathbf{v}_i$. Let each layer-i have k_i neurons, we have

that $\mathbf{v}_{i-1}$ is a vector of k_{i-1} entries and $\mathbf{v}_i$ is a vector of k_i entries. Therefore, the transformation can be seen as a mapping from high-dimensional space $\mathbb{R}^{k_{i-1}}$ to $\mathbb{R}^{k_i}$. Generalise this to the entire network, we have

$$\mathbb{R}^{k_0} \overset{f_1}{\longmapsto} \mathbb{R}^{k_1} \overset{f_2}{\longmapsto} \ldots \overset{f_m}{\longmapsto} \mathbb{R}^{k_m} \tag{4.9}$$

Note that, k_0 is the number of input features and k_m is the number of class labels.

Training Objective The training typically intends to get the best weights $\mathbf{W}^*$ as follows.

$$\mathbf{W}^* \leftarrow \arg\min_{\mathbf{W}} \sum_{(\mathbf{x},y)\in D} L(y, \mathbf{v}_m) \tag{4.10}$$

where $L(y, f_{\mathbf{W}}(\mathbf{x}))$ is typically a loss function measuring the gap between actual label y with its current prediction $f_{\mathbf{W}}(\mathbf{x})$. However, the optimisation problem is highly dimensional and non-convex. Therefore, in most cases, the training ends up with an approximation $\hat{\mathbf{W}}$.

4.2.2 Recurrent Neural Networks

The above is mainly for feedforward neural networks (FNNs), which model a function $\phi\colon X \to Y$ that maps from input domain X to output domain Y: given an input $x \in X$, it outputs the prediction $y \in Y$. For a sequence of inputs $x_1, \ldots, x_n$, an FNN ϕ considers each input individually, that is, $\phi(x_i)$ is independent from $\phi(x_{i+1})$.

By contrast, a recurrent neural network (RNN) processes an input sequence by iteratively taking inputs one by one. A recurrent layer can be modeled as a function $\psi\colon X' \times C \times Y' \to C \times Y'$ such that $\psi(x_t, c_{t-1}, h_{t-1}) = (c_t, h_t)$ for $t = 1..n$, where t denotes the t-th time step, c_t is the cell state used to represent the intermediate memory and h_t is the output of the t-th time step. More specifically, the recurrent layer takes three inputs: x_t at the current time step, the prior memory state c_{t-1} and the prior cell output h_{t-1}; consequently, it updates the current cell state c_t and outputs current h_t.

RNNs differ from each other given their respective definitions, i.e., internal structures, of recurrent layer function ψ, of which long short-term memory (LSTM) in Equation (4.11) is the most popular and commonly used one.

$$\begin{aligned} f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\ i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ c_t &= f_t * c_{t-1} + i_t * \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \\ o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\ h_t &= o_t * \tanh(c_t) \end{aligned} \tag{4.11}$$

such that $\sigma(x) \in [0, 1]$ for any $x \in \mathbb{R}$, $\tanh$ is the hyperbolic tangent function such that $\tanh(x) \in [-1, 1]$ for any $x \in \mathbb{R}$, W_f, W_i, W_c, W_o are weight matrices, b_f, b_i, b_c, b_o are bias vectors, f_t, i_t, o_t are internal gate variables, h_t is the hidden state variable (utilising o_t), and c_t is the cell state variable. For the connection with successive layers, we only take the last output h_n as the output. For simplicity, when working with finite sequential data, we can also define a recurrent layer as $\psi\colon (X')^n \to Y'$, which takes, as input, a sequential data of length n and returns the last output h_n. Figure 4.7 presents an illustrative diagram for LSTM cell.

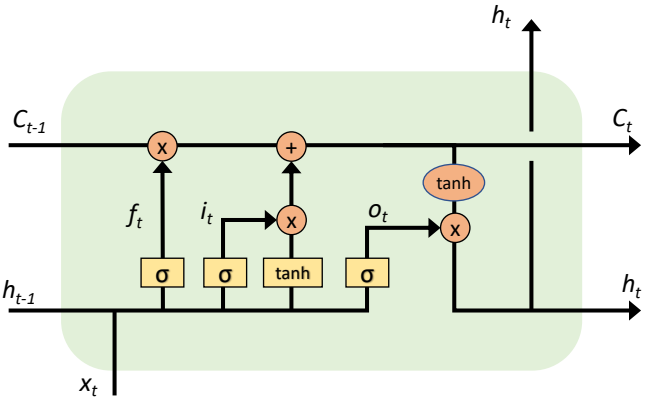


Figure 4.7: LSTM Cell

In LSTM, σ is the sigmoid function and $\tanh$ is the hyperbolic tangent function; W and b represent the weight matrix and bias vector, respectively; f_t, i_t, o_t are internal gate variables of the cell. In general, the recurrent layer (or LSTM layer) is connected to non-recurrent layers such as fully connected layers so that the cell output propagates further. We denote the remaining layers with a function $\phi_2 \colon Y' \to Y$. Meanwhile, there can be feedforward layers connecting to the RNN layer, and we let it be another function $\phi_1 \colon X \to X'$. As a result, the RNN model that accepts a sequence of inputs $x_1, \ldots, x_n$ can be modeled as a function φ such that $\varphi(x_1 \ldots x_n) = \phi_2 \cdot \psi(\prod_{i=1}^{n} \phi_1(x_i))$. Normally, the recurrent layer is connected to non-RNN layers such as fully connected layers so that the output h_n is processed further. We let the remaining layer be a function $\phi_2 \colon Y' \to Y$. Moreover, there can be feedforward layers connecting to the RNN layer, and we let it be a function $\phi_1 \colon X \to X'$. Then given a sequential input $x_1, \ldots, x_n$, the RNN is a function φ such that

4.2.3 Learning Representation and Features

Raw digital representation Every instance has to be represented in a digital form. For example, the 8th instance in the **digits** dataset is an image of digit 8 as shown in Figure 4.8.

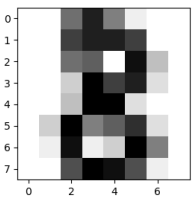


Figure 4.8: A small image of digit 8

Actually, it is stored as a matrix as follows:

$$\begin{pmatrix} 0 & 0 & 9 & 14 & 8 & 1 & 0 & 0 \\ 0 & 0 & 12 & 14 & 14 & 12 & 0 & 0 \\ 0 & 0 & 9 & 10 & 0 & 15 & 4 & 0 \\ 0 & 0 & 3 & 16 & 12 & 14 & 2 & 0 \\ 0 & 0 & 4 & 16 & 16 & 2 & 0 & 0 \\ 0 & 3 & 16 & 8 & 10 & 13 & 2 & 0 \\ 0 & 1 & 15 & 1 & 3 & 16 & 8 & 0 \\ 0 & 0 & 11 & 16 & 15 & 11 & 1 & 0 \end{pmatrix} \tag{4.12}$$

As another example, the videos are actually a sequence of images, and therefore it is stored as a 3-dimensional array.

Features are domain dependent. Evidently, for computer vision, pixels are input features, and for natural language processing, words are input features. In addition to input features which closely relate to data representation, we may use the term latent features or hidden features for those features in the hidden layers.

Feature Extraction is one of the key intermediate tasks for learning, for both traditional machine learning and deep learning. It is a process that identifies important features or attributes of the data. For traditional machine learning, as shown in Figure 4.9, it first extracts features and then applies a learnable classifier. The feature extraction is treated as a step independent of

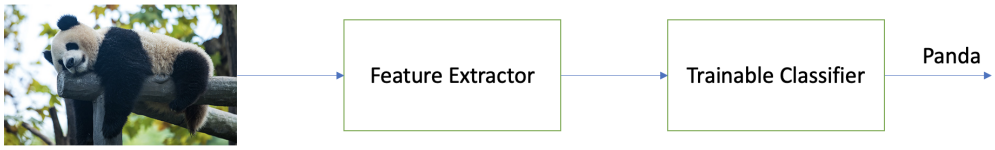


Figure 4.9: Flow of traditional machine learning

the classification. There are many different methods for feature extraction, for example, SIFT (scale-invariant feature transform).

Deep learning, however, requires only one step (i.e., end-to-end) to implement both feature extraction and classification, as shown in Figure 4.10. Both the feature extractor and the classifier

Figure 4.10: Flow of deep learning

are trained at the same time.

Feature extraction is closely related to the dimensionality reduction, i.e., to separate data as much as possible. Most data distributions and tasks are non-linear, so a linear assumption is often convenient, but not necessarily truthful. Therefore, to get non-linear machines without too much effort, we may have to consider non-linear features.

There are many ways to get non-linear features, including e.g.,

- application of non-linear kernels, e.g., polynomial, RBF, etc.
- explicit design of features, e.g., SIFT, HOG, etc.

The quality of features is usually evaluated against the following few criteria:

- invariance
- repeatabilty
- discriminativeness
- robustness

It is useful to note that these criteria may be conflicting. Hence, there needs to be a trade-off between criteria.

Data manifold Actually, most natural, high-dimensional data (e.g. faces) lie on lower dimensional manifolds. For example, Figure 4.11 is the so-called "swiss roll", where the data points are in 3-dimensional, but they all lie on 2-dimensional manifold. That is, the actual dimensionality of the manifold is 2, while the dimensionality of the input space is 3.

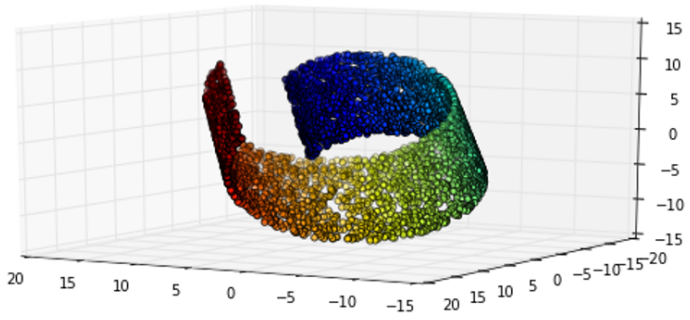


Figure 4.11: Data manifold – "swiss role" example

Therefore, although the data points may consist of thousands of features, they can be described as a function of only a few underlying parameters. That is, the data points are actually sampled from a low-dimensional manifold that is embedded in a high-dimensional space.

Difficulties of simply using dimensionality reduction or kernel The above observation suggests that our goal should be on discovering lower dimensional manifolds. We remark that, these manifolds are most probably highly non-linear.

The success of this requires two hypothesese:

- If we can compute the coordinates of the input (e.g., a face image) to this non-linear manifold then the data become separable. This hypothesis suggests the existence of *functional mapping*. For the "swiss role" example, there should be a (non-linear) function mapping from 3d space to 2d space, on which the data can be linearly separable.
- Semantically similar things lie closer than semantically dissimilar things. This implies the existence of applicable dimensional reduction methods.

While raw data live in huge dimensionality, semantically meaningful raw data prefer lower dimensional manifolds, which still live in the same huge dimensionality. Can we discover this manifold to embed our data on?

End-to-end learning of feature hierarchies The above discussions basically suggest that, it is an almost impossible task to manually craft features and also nontrivial to design algorithms (dimensionality reduction, functional mapping, etc) to compute features. This is in stark contrast with deep learning. Actually, one of the key advantages of convolutional neural networks is its ability to learn (or extract) features automatically.

In a CNN, there are a pipeline of successive layers, such that each layer's output is the input for the next layer. Layers produce features of higher and higher abstractions, such that the shadow layers extract low-level features (e.g. edges or corners), middle layers extract mid-level features (e.g. circles, squares, textures), and deep layers capture high level, class specific features (e.g. face detector). See Figure 4.12.

We remark that, for CNNs, it has been shown that, preferably, training data should be as raw as possible. That is, no additional feature extraction phase is needed.

Why learn the features? Manually designed features often take a lot of time to come up with and implement, a lot of time to validate, and are incomplete, as one cannot know if they are optimal for the task. On the other hand, learned features are easy to adapt, very compact and specific to the task at hand. Given a basic architecture in mind, it is relatively easy and fast to optimize, i.e., time spent for designing features now spent for designing architectures.

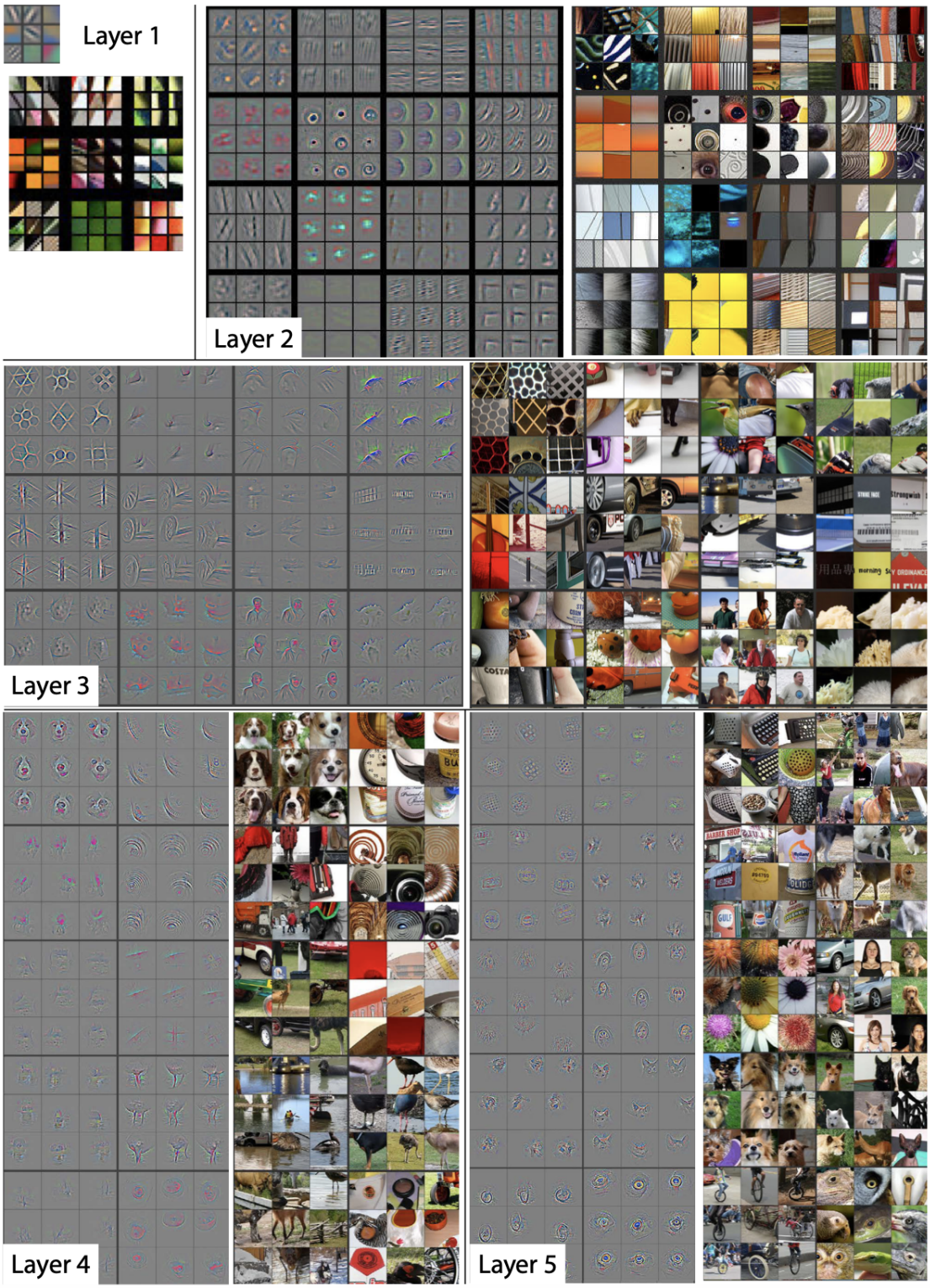


Figure 4.12: Visualisation of features in hidden layers [15]

4.2.4 Practicals

The following is a code to train a neural network with fully-connected layers.

Train a fully connected model First of all, we install two packages torch and torchvision.

```
$ pip3 install torch
$ pip3 install torchvision
```

Then, we setup hyper-parameters (e.g., batchsize, epoch, learning rate), device (e.g., CPU or GPU), and load training dataset (MNIST).

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
import argparse
import time

# Setup hyper-parameter
parser = argparse.ArgumentParser(description='PyTorch MNIST Training')
parser.add_argument('--batch-size', type=int, default=128, metavar='N',
                    help='input batch size for training (default: 128)')
parser.add_argument('--test-batch-size', type=int, default=128, metavar='N',
                    help='input batch size for testing (default: 128)')
parser.add_argument('--epochs', type=int, default=10, metavar='N',
                    help='number of epochs to train')
parser.add_argument('--lr', type=float, default=0.01, metavar='LR',
                    help='learning rate')
parser.add_argument('--no-cuda', action='store_true', default=False,
                    help='disables CUDA training')
parser.add_argument('--seed', type=int, default=1, metavar='S',
                    help='random seed (default: 1)')

args = parser.parse_args(args=[])

# Judge cuda is available or not
use_cuda = not args.no_cuda and torch.cuda.is_available()
#device = torch.device("cuda" if use_cuda else "cpu")
device = torch.device("cpu")

torch.manual_seed(args.seed)
kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}

# Setup data loader
transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
        ])
trainset = datasets.MNIST('../data', train=True, download=True,
                    transform=transform)
testset = datasets.MNIST('../data', train=False,
                    transform=transform)
```

```
train_loader = torch.utils.data.DataLoader(trainset,batch_size=args.
    batch_size, shuffle=True,**kwargs)
test_loader = torch.utils.data.DataLoader(testset,batch_size=args.
    test_batch_size, shuffle=False, **kwargs)
```

We can define a fully connected network as follows, with the structure 784-128-64-32-10,

```
# Define fully connected network
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 32)
        self.fc4 = nn.Linear(32, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        x = F.relu(x)
        x = self.fc4(x)
        output = F.log_softmax(x, dim=1)
        return output
```

Then, we define the training function, which computes loss and updates parameters for each minibatch.

```
# Training function
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        data = data.view(data.size(0),28*28)

        # Clear gradients
        optimizer.zero_grad()

        # Compute loss
        loss = F.cross_entropy(model(data), target)

        # Get gradients and update
        loss.backward()
        optimizer.step()
```

We also can define a predict function, which outputs training loss and test loss for each epoch.

```
# Predict function
def eval_test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
```

```
            data, target = data.to(device), target.to(device)
            data = data.view(data.size(0),28*28)
            output = model(data)
            test_loss += F.cross_entropy(output, target, size_average=False).item()
            pred = output.max(1, keepdim=True)[1]
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader.dataset)
    test_accuracy = correct / len(test_loader.dataset)
    return test_loss, test_accuracy
```

Finally, we define the main function and call the training function for each epoch.

```
# Main function, train the dataset and print training loss, test loss
def main():
    model = Net().to(device)
    optimizer = optim.SGD(model.parameters(), lr=args.lr)
    for epoch in range(1, args.epochs + 1):
        start_time = time.time()

        # Training
        train(args, model, device, train_loader, optimizer, epoch)

        # Get trnloss and testloss
        trnloss, trnacc = eval_test(model, device, train_loader)
        tstloss, tstacc = eval_test(model, device, test_loader)

        # Print trnloss and testloss
        print('Epoch '+str(epoch)+': '+str(int(time.time()-start_time))+'s', end=', ')
        print('trn_loss: {:.4f}, trn_acc: {:.2f}%'.format(trnloss, 100. * trnacc), end=', ')
        print('test_loss: {:.4f}, test_acc: {:.2f}%'.format(tstloss, 100. * tstacc))

if __name__ == '__main__':
    main()
```

4.3 Forward and Backward Computation

In Section 4.1, we have explained that multi-layer perceptron has more expressive power than single layer perceptron. In particular, it is able to find a two-layer perceptron to solve the XOR problem, while it is not possible for single-layer perceptron. However, we did not explain in Section 4.1 how to compute the weights for the two-layer perceptron for XOR. Moreover, we note that, the perceptron learning algorithm cannot be used for learning multi-layer perceptron. Actually, the learning algorithm for multi-layer perceptron, called backpropagation (BP), is one of the key milestones for the development of deep learning.

The BP algorithm computes the gradient of the loss function with respect to each weight by the chain rule. Instead of computing one gradient for each weight, it is able to compute compute the gradient for one layer at a time, and more importantly, it is able to iterate backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule. This makes it efficient enough to train large scale neural networks.

The BP algorithm is the foundation of deep learning, and in this section, we use a running example to explain its computation.

Running Example In this example, we have three layers, one input layer, one hidden layer, and one output layer. Each layer has two neurons. The connections between neurons are given in the left diagram of Figure 4.13.

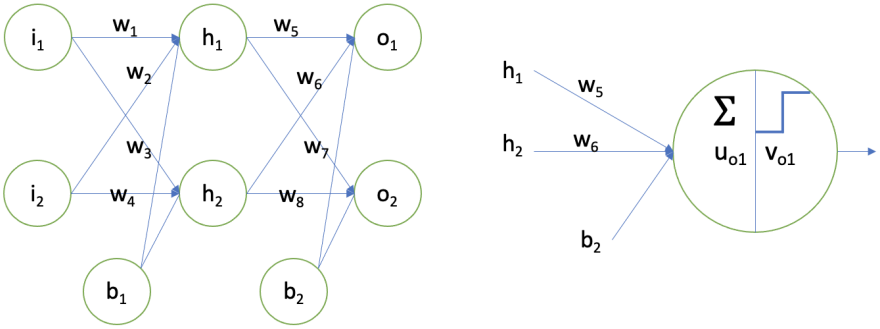


Figure 4.13: A simple neural network with a hidden layer (Left) and the illustration of a neuron with activation function (Right)

The diagram on the right is an illustration of a neuron with activation function. Each neuron has two values u and v, representing its values before and after the application of activation function, respectively.

The learning is a dynamic process with weights continuously updated until convergence. Now, we assume that, at some point, the current weights are

$$\begin{pmatrix} w_1 & w_2 \\ w_3 & w_4 \end{pmatrix} = \begin{pmatrix} 0.15 & 0.20 \\ 0.25 & 0.30 \end{pmatrix}, \quad \begin{pmatrix} w_5 & w_6 \\ w_7 & w_8 \end{pmatrix} = \begin{pmatrix} 0.40 & 0.45 \\ 0.50 & 0.55 \end{pmatrix}, \quad \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} 0.35 \\ 0.60 \end{pmatrix} \tag{4.13}$$

Note that, every row in a weight matrix stores the input weights for a neuron, for example, the

row $(0.15 \;\; 0.20)$ is associated with the neuron h_1. Also, each entry in a bias vector represents a bias for a layer.

4.3.1 Forward Computation

The first step of each iteration of the BP algorithm is to compute the loss by making a forward computation. Assume that we have an input $\mathbf{x} = (0.05, 0.10)^T$ for the network in Figure 4.13, we can have

$$\begin{pmatrix} u_{h_1} \\ u_{h_2} \end{pmatrix} = \begin{pmatrix} 0.15 & 0.20 \\ 0.25 & 0.30 \end{pmatrix} \times \begin{pmatrix} 0.05 \\ 0.10 \end{pmatrix} + \begin{pmatrix} 0.35 \\ 0.35 \end{pmatrix} = \begin{pmatrix} 0.3775 \\ 0.6425 \end{pmatrix} \tag{4.14}$$

Assume that the network uses the Sigmoid function σ as the activation function, we have

$$\begin{pmatrix} v_{h_1} \\ v_{h_2} \end{pmatrix} = \sigma(\begin{pmatrix} 0.3775 \\ 0.3925 \end{pmatrix}) \approx \begin{pmatrix} 0.5927 \\ 0.5969 \end{pmatrix} \tag{4.15}$$

as the output of the hidden layer. On the output layer, we have

$$\begin{pmatrix} u_{o_1} \\ u_{o_2} \end{pmatrix} = \begin{pmatrix} 0.40 & 0.45 \\ 0.50 & 0.55 \end{pmatrix} \times \begin{pmatrix} 0.5927 \\ 0.5969 \end{pmatrix} + \begin{pmatrix} 0.60 \\ 0.60 \end{pmatrix} = \begin{pmatrix} 1.1057 \\ 1.2247 \end{pmatrix} \tag{4.16}$$

Consider the Sigmoid function, we have

$$\begin{pmatrix} v_{o_1} \\ v_{o_2} \end{pmatrix} = \sigma(\begin{pmatrix} 1.1057 \\ 1.2247 \end{pmatrix}) \approx \begin{pmatrix} 0.7513 \\ 0.7729 \end{pmatrix} \tag{4.17}$$

as the output of the output layer. That is, $\hat{y} = (0.7513, 0.7729)$. Now, assuming that the label of $\mathbf{x}$ is $y = (0.01, 0.99)^T$ and we are using the mean square error, we can compute the loss for

$$L(\mathbf{x}, y) = \frac{1}{2}(0.7513 - 0.01)^2 + \frac{1}{2}(0.7729 - 0.99)^2 \approx 0.2748 + 0.0236 = 0.2984 \tag{4.18}$$

where we let $L_{o1}(\mathbf{x}, y) = \frac{1}{2}(0.7513 - 0.01)^2$ and $L_{o2}(\mathbf{x}, y) = \frac{1}{2}(0.7729 - 0.99)^2$, representing the loss of individual neurons o_1 and o_2, respectively.

4.3.2 Backward Computation

Once we have the loss $L(\mathbf{x}, y)$, we can start back-propagation by applying the chain rule.

Weights of output neurons First of all, for the weights of output neuron, such as w_5, we can compute as follows:

$$\frac{\partial L}{\partial w_5} = \frac{\partial L_{o_1}}{\partial v_{o_1}} * \frac{\partial v_{o_1}}{\partial u_{o_1}} * \frac{\partial u_{o_1}}{\partial w_5} \tag{4.19}$$

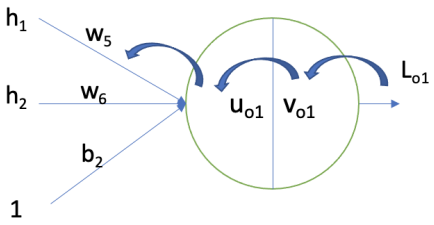


Figure 4.14: Backward propagation on the output neuron

Figure 4.14 presents an illustrative diagram for the Equation (4.19). Actually, the backpropagation goes from the loss L_{o_1} to the value v_{o_1}, u_{o_1}, until the weight w_5.

Concretely, for the running example, we have

$$\frac{\partial L_{o_1}}{\partial v_{o_1}} = \frac{\partial}{\partial v_{o_1}}(\frac{1}{2}(y^{(1)} - v_{o_1})^2) = -(y^{(1)} - v_{o_1}) = 0.7513 - 0.01 = 0.74 \tag{4.20}$$

where $y^{(1)}$ is the first component of y, and

$$\frac{\partial v_{o_1}}{\partial u_{o_1}} = \frac{\partial \sigma(u_{o_1})}{\partial u_{o_1}} = \sigma(u_{o_1})(1 - \sigma(u_{o_1})) = v_{o_1}(1 - v_{o_1}) \approx 0.7513 \times 0.2487 \approx 0.1868 \tag{4.21}$$

and

$$\frac{\partial u_{o_1}}{\partial w_5} = \frac{\partial}{\partial w_5}(w_5 v_{h_1} + w_6 v_{h_2} + b_2) = v_{h_1} \approx 0.5927 \tag{4.22}$$

Therefore, we have

$$\frac{\partial L}{\partial w_5} = \frac{\partial L_{o_1}}{\partial v_{o_1}} * \frac{\partial v_{o_1}}{\partial u_{o_1}} * \frac{\partial u_{o_1}}{\partial w_5} \approx 0.74 \times 0.1868 \times 0.5927 = 0.0819 \tag{4.23}$$

Weights of hidden neurons Now, the weight of hidden layer can be done recursively by applying the chain rules, e.g.,

$$\begin{aligned}
\frac{\partial L}{\partial w_1} &= \frac{\partial L}{\partial v_{h_1}} * \frac{\partial v_{h_1}}{\partial u_{h_1}} * \frac{\partial u_{h_1}}{\partial w_1} \\
&= (\frac{\partial L_{o_1}}{\partial v_{h_1}} + \frac{\partial L_{o_2}}{\partial v_{h_1}}) * \frac{\partial v_{h_1}}{\partial u_{h_1}} * \frac{\partial u_{h_1}}{\partial w_1} \\
&= (\frac{\partial L_{o_1}}{\partial u_{o_1}}\frac{\partial u_{o_1}}{\partial u_{h_1}} + \frac{\partial L_{o_2}}{\partial u_{o_2}}\frac{\partial u_{o_2}}{\partial u_{h_1}}) * \frac{\partial v_{h_1}}{\partial u_{h_1}} * \frac{\partial u_{h_1}}{\partial w_1} \\
&= (\frac{\partial L_{o_1}}{\partial u_{o_1}} w_5 + \frac{\partial L_{o_2}}{\partial u_{o_2}} w_7) * \frac{\partial v_{h_1}}{\partial u_{h_1}} * \frac{\partial u_{h_1}}{\partial w_1}
\end{aligned} \tag{4.24}$$

Note that, all the components of Equation (4.24) can now be computed as the method we used for the output layer. Figure 4.15 presents an illustration of backward propagation as in Equation (4.24).

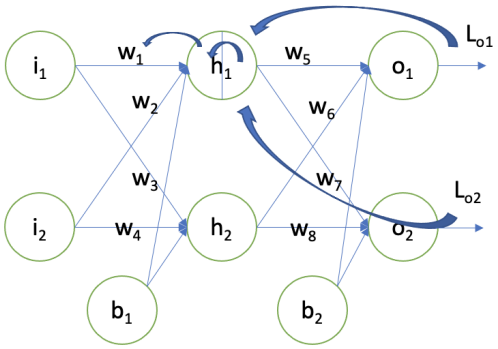


Figure 4.15: Backward propagation on the hidden neuron

Weight update Finally, once we compute the gradients $\frac{\partial L}{\partial w_5}$ or $\frac{\partial L}{\partial w_1}$, we can update the weights w_5 or w_1 by applying the gradient descent algorithm. We remark that, while the above computation is conducted for individual weights, the BP algorithm can work on a layer basis to significantly improve the efficiency.

4.3.3 Regularisation as Constraints

In general, regularisation is a set of methods to prevent overfitting or help the optimization. Typically, this is done by having additional terms in the training optimisation objective.

Overfitting Overfitting is a concept closely related to the generalisation error as introduced in Section 2.3.1, which is the gap between empirical loss and expected loss. It has been observed that, the following two reasons may contribute as the key to the overfitting.

- dataset is too small
- hypothesis space is too large

To understand the second point, we note that, the larger the hypothesis space, the easier is for a learning algorithm to find a hypothesis that has small training error. However, finding a small training error does not warrant the found hypothesis can be of a small test error, and so this may lead to large test error (overfitting). This observation suggests that it might be beneficial to leave out useless hypotheses, which is what regularization is for.

Regularization as hard constraint Assume that, for a dataset $D = (\mathbf{X}, \mathbf{y})$ of n training instances, we have the following optimising objective:

$$\begin{aligned} \min_{f} \quad & L(f, D) = \frac{1}{n} \sum_{i=1}^{n} L(f, \mathbf{x}_i, y_i) \\ \text{subject to} \quad & f \in \mathcal{H} \end{aligned} \tag{4.25}$$

Considering that for a deep learning model, f is parameterised over the weights W, we have

$$\begin{aligned} \min_{W} \quad & L(W, D) = \frac{1}{n} \sum_{i=1}^{n} L(W, \mathbf{x}_i, y_i) \\ \text{subject to} \quad & W \in \mathbb{R}^{|\mathcal{W}|} \end{aligned} \tag{4.26}$$

where $|W|$ is the number of weights.

The regularisation is to add further constraints. For example, if we ask for L_2 regularisation, we have

$$\begin{aligned} \min_{W} \quad & L(W, D) = \frac{1}{n} \sum_{i=1}^{n} L(W, \mathbf{x}_i, y_i) \\ \text{subject to} \quad & W \in \mathbb{R}^{|\mathcal{W}|} \\ & ||W||_2^2 \leq r^2 \end{aligned} \tag{4.27}$$

for some pre-specified $r > 0$.

Regularization as soft constraint While the hard constraints limit the selection of hypothesis, it might not be easy to be integrated with the backpropagation algorithm, which does not consider the constraints directly. This can be done through a soft constraint, e.g.,

$$\min_{W} \quad L(W, D) = \frac{1}{n} \sum_{i=1}^{n} L(W, \mathbf{x}_i, y_i) + \lambda ||W||_2^2 \tag{4.28}$$

where λ is a hyper-parameter to balance between the loss term and the constraint/penalty term. Alternatively, this can be done through Lagrangian multiplier method:

$$\min_{W} \quad L(W, D) = \frac{1}{n} \sum_{i=1}^{n} L(W, \mathbf{x}_i, y_i) + \lambda(||W||_2^2 - r) \tag{4.29}$$

4.3.4 Practicals

The following code is to save the weights of a trained model and load the weights from file.

```
# Save model
torch.save(model.state_dict(), 'model.pt')

# Load model
model.load_state_dict(torch.load('model.pt'))
```

4.4 Convolutional Neural Networks

This section introduces a specific class of neural networks that has been shown very effective in processing images. In 1998, Yann LeCun and his collaborators developed a neural network for handwritten digits called LeNet [9]. It is a feedforward network with several hidden layers, trained with the backpropagation algorithm. It is later formalised with the name convolutional neural networks (CNNs). Since LeNet, there are many other variants of CNNs, such as AlexNet, VGG16, ResNet, GoogLeNet, and so on. These variants introduce new functional layers or training methods that help improve the performance of the CNNs in pattern recognition tasks.

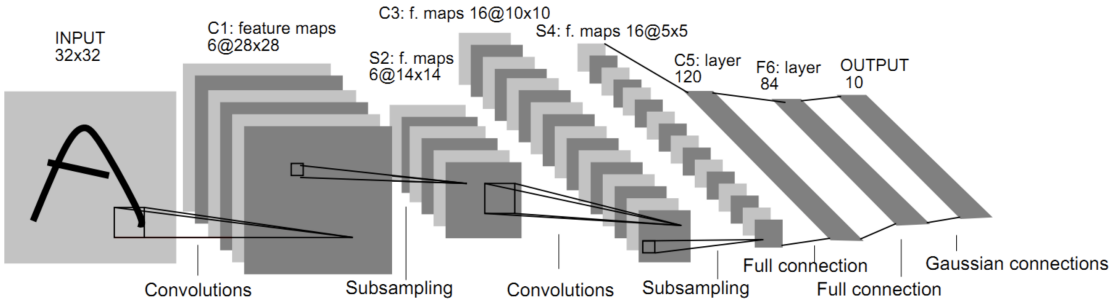


Figure 4.16: Architecture of LeNet-5 [9], a convolutional neural network for digits recognition.

As shown in Figure 4.16, LeNet has an input layer, 6 hidden layers, and an output layer. Among the 4 hidden layers, there are 2 convolutional layers, 2 subsampling (or pooling) layers, and 2 fully connected layers. Actually, common functional layers of a CNN can be e.g., fully-connected layers, convolutional layers, pooling layers, etc. It is very often that a functional layer is followed by an activation layer, such as ReLU layer, Sigmoid layer, Tanh layer, etc. After a sequence of functional and activation layers, we need a softmax layer to convert the output into a probability distribution.

In the following, we first introduce functional layers, activation functions, and softmax layer that have been widely used in various CNNs, and then present a few common practices that have been used to either prepare data for training in or support the training.

4.4.1 Functional Layers

As suggested earlier, each layer function f_i is a mapping from a high-dimensional space $\mathbb{R}^{k_{i-1}}$ (that associates with Layer-$(i-1)$) to another $\mathbb{R}^{k_i}$ (that associates with Layer-i). That is, given $\mathbf{v}_{i-1} \in \mathbb{R}^{k_i}$, we have $\mathbf{v}_i = f_i(\mathbf{v}_{i-1}) \in \mathbb{R}^{k_i}$.

Actually, in most CNN layers, the transformation f_i is conducted in two steps. For the first step, it is transformed with a linear transformation, and in the second step, every neuron passes through an activation function. Formally,

$$\mathbf{u}_i = \mathbf{W}_i \mathbf{v}_{i-1} + \mathbf{b}_i \quad \text{and} \quad \mathbf{v}_i = \sigma_i(\mathbf{u}_i) \tag{4.30}$$

where σ_i is an activation function. In the following, we introduce functional layers, followed by activation functions.

Fully-connected Layer

In a fully connected layer, every neuron receives inputs from all neurons of the previous layer, and the output of the neuron is the result of the linear combination of the inputs. As shown in Figure 4.17, Layer 2 is a fully-connected layer. The neuron n_{21} receives inputs from all neurons $n_{11}, ..., n_{16}$ in Layer 1, and weighted them with the learnable weights $\mathbf{W}_{21} = (w_{21,11}, ..., w_{21,16})$. Therefore, its value

$$u_{21} = \mathbf{W}_{21} \times \mathbf{v}_1 + b_2 = w_{21,11}v_{11} + ... + w_{21,16}v_{16} + b_2 \tag{4.31}$$

where b_2 is the bias of layer 2. Similar for the neuron n_{22}.

Note that, in the above, we use symbol u, instead of v, to denote the value of the neuron, because the output of this neuron normally needs to pass through an activation function, i.e.,

$$v_{21} = \alpha(u_{21}) \tag{4.32}$$

We will introduce activation functions α later.

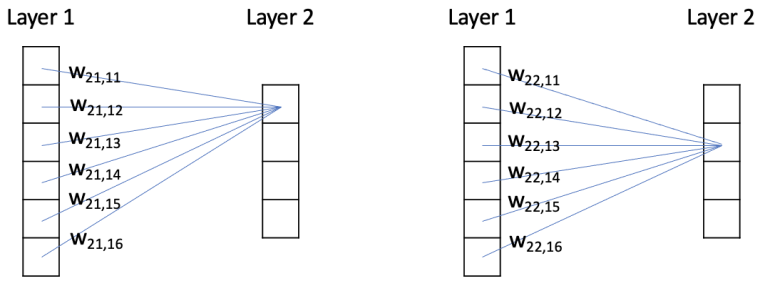


Figure 4.17: Illustration of fully-connected layer

In a CNN, fully connected layers often appear in the last few layers, after the convolutional layers. The main functionality of those fully-connected layers are to implement classification over those features extracted by the convolutional layers.

Convolutional Layer

Given an $n \times n$ matrix $\mathbf{x}$ and an $m \times m$ filter $\mathbf{f}$, we can compute the resulting matrix $\mathbf{z}$ by repeatedly (1) overlapping the filter over the matrix, as illustrated in Figure 4.18 with the red dashed lines, and (2) computing an element $z_{i,j}$ with element-wise multiplication, as illustrated in Figure 4.18 with the blue dashed lines.

The overlapping usually starts from the element (1,1) of the matrix $\mathbf{x}$. Therefore, the (1,1) element in the resulting matrix $\mathbf{z}$ is computed as follows with the element-wise multiplication:

$$z_{1,1} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k,1+l} \times f_{1+k,1+l} \tag{4.33}$$

Afterwards, it depends on a parameter $stride$ to determine the next element on $\mathbf{x}$. For example, if $stride = 1$, then one of the next elements, along the horizontal direction, is $(1, 1 + stride) = (1, 2)$ such that

$$z_{1,2} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k,1+l+stride} \times f_{1+k,1+l} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k,1+l+1} \times f_{1+k,1+l} \tag{4.34}$$

The other next element, along the vertical direction, is $(1 + stride, 1) = (2, 1)$ such that

$$z_{2,1} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k+stride,1+l} \times f_{1+k,1+l} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k+1,1+l} \times f_{1+k,1+l} \tag{4.35}$$

Figure 4.18 presents the case where we move horizontally with $stride = 1$.

If $stride = 2$, the next horizontal element on $\mathbf{x}$ will be $(1, 1 + stride) = (1, 3)$ and we are computing the $(1, 2)$ element for the resulting matrix $\mathbf{z}$, i.e.,

$$z_{1,2} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k,1+l+stride} \times f_{1+k,1+l} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k,1+l+2} \times f_{1+k,1+l} \tag{4.36}$$

Similarly, the next vertical element $(2, 1)$ is computed as follows:

$$z_{2,1} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k+stride,1+l} \times f_{1+k,1+l} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k+2,1+l} \times f_{1+k,1+l} \tag{4.37}$$

Note that, no matter what the $stride$ is, the incremental to the element on $\mathbf{z}$ is always 1, to make sure that we are constructing $\mathbf{z}$ one element by one element.

Zero-Padding

As we can see from the previous discussion on convolutional layer, the shapes of the matrices $\mathbf{x}$ and $\mathbf{z}$ are not the same. It is possible that we might be interested in maintaining the shape of the matrix along a sequence of convolutional operations. In this case, it is useful to consider a pre-processing on $\mathbf{x}$ before the convolutional filter is applied. Zero-padding, a typical pre-processing operation, is to use 0 to pad the input with 0-cells, as shown in Figure 4.19.

We can see that,

Pooling Layer

A pooling layer is to reduce information in a matrix by collapsing elements with operations.

4.4.2 Activation Functions

As mentioned earlier, for most functional layers, they are followed by an activation layer where

ReLU

$$ReLU(x) = max(0, x) \tag{4.38}$$

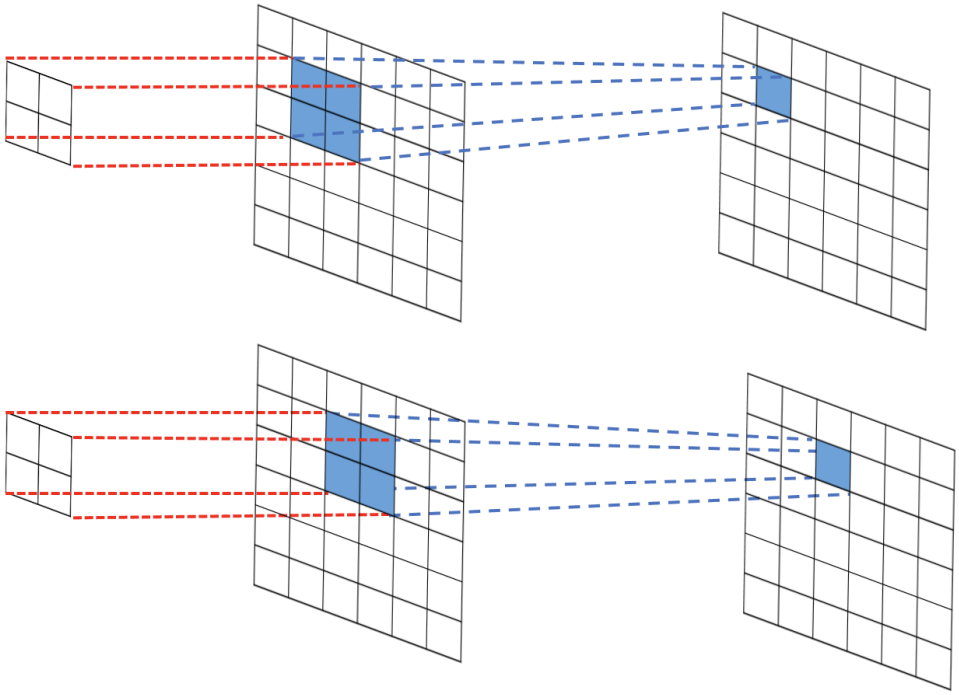


Figure 4.18: Illustration of convolutional layer

4	0	1	7
5	6	9	-5
-3	8	3	6
2	-2	-1	4

0	0	0	0	0	0
0	4	0	1	7	0
0	5	6	9	-5	0
0	-3	8	3	6	0
0	2	-2	-1	4	0
0	0	0	0	0	0

Figure 4.19: Zero-padding: pad the input with 0-cells around it

ReLU

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{4.39}$$

such that $\sigma'(x) = \sigma(x)(1-\sigma(x))$.

4.4.3 Softmax

$$\delta(\mathbf{v}) = (\frac{e^{v_1}}{\sum_{i=1}^{|\mathbf{v}|} e^{v_i}}, ..., \frac{e^{v_{|\mathbf{v}|}}}{\sum_{i=1}^{|\mathbf{v}|} e^{v_i}}) \tag{4.40}$$

4.4.4 Data Preprocessing

4.4.5 Practicals

[xiaowei: here we need a code piece to train a convolutional neural networks and retrieve the weights.]

First, we setup hyper-parameters (e.g., batchsize, epoch, learning rate), device (e.g., CPU or GPU), and load training dataset (MNIST).

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
import argparse
import time
import os

# Setup training parameters
parser = argparse.ArgumentParser(description='PyTorch MNIST Training')
parser.add_argument('--batch-size', type=int, default=128, metavar='N',
                    help='input batch size for training (default: 128)')
parser.add_argument('--test-batch-size', type=int, default=128, metavar='N',
                    help='input batch size for testing (default: 128)')
parser.add_argument('--epochs', type=int, default=5, metavar='N',
                    help='number of epochs to train')
parser.add_argument('--lr', type=float, default=0.01, metavar='LR',
                    help='learning rate')
parser.add_argument('--no-cuda', action='store_true', default=False,
                    help='disables CUDA training')
parser.add_argument('--seed', type=int, default=1, metavar='S',
                    help='random seed (default: 1)')
parser.add_argument('--model-dir', default='./model-mnist-cnn',
                    help='directory of model for saving checkpoint')
parser.add_argument('--load-model', action='store_true', default=False,
                    help='load model or not')

args = parser.parse_args(args=[])

if not os.path.exists(args.model_dir):
    os.makedirs(args.model_dir)

```

```
# Judge cuda is available or not
use_cuda = not args.no_cuda and torch.cuda.is_available()
#device = torch.device("cuda" if use_cuda else "cpu")
device = torch.device("cpu")

torch.manual_seed(args.seed)
kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}

# Setup data loader
transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
        ])
trainset = datasets.MNIST('../data', train=True, download=True,
                    transform=transform)
testset = datasets.MNIST('../data', train=False,
                    transform=transform)
train_loader = torch.utils.data.DataLoader(trainset,batch_size=args.
    batch_size, shuffle=True,**kwargs)
test_loader = torch.utils.data.DataLoader(testset,batch_size=args.
    test_batch_size, shuffle=False, **kwargs)
```

We can define a convolutional neural network as follows, with 2 convolutional layers and 2 fully connected layers.

```
# Define CNN
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # in_channels:1  out_channels:32  kernel_size:3  stride:1
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        # in_channels:32  out_channels:64  kernel_size:3  stride:1
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output
```

```
# Train function
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
```

```
        #clear gradients
        optimizer.zero_grad()

        #compute loss
        loss = F.cross_entropy(model(data), target)

        #get gradients and update
        loss.backward()
        optimizer.step()

# Predict function
def eval_test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.cross_entropy(output, target, size_average=False).
    item()
            pred = output.max(1, keepdim=True)[1]
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader.dataset)
    test_accuracy = correct / len(test_loader.dataset)
    return test_loss, test_accuracy
```

Finally, we define the main function, which can load the trained model, or train the initial model and save the trained model.

```
# Main function, train the initial model or load the model
def main():
    model = Net().to(device)
    optimizer = optim.SGD(model.parameters(), lr=args.lr)

    if args.load_model:
        # Load model
        model.load_state_dict(torch.load(os.path.join(args.model_dir, '
    final_model.pt')))
        trnloss, trnacc = eval_test(model, device, train_loader)
        tstloss, tstacc = eval_test(model, device, test_loader)
        print('trn_loss: {:.4f}, trn_acc: {:.2f}%'.format(trnloss, 100. *
    trnacc), end=', ')
        print('test_loss: {:.4f}, test_acc: {:.2f}%'.format(tstloss, 100. *
    tstacc))

    else:
        # Train initial model
        for epoch in range(1, args.epochs + 1):
            start_time = time.time()

            #training
            train(args, model, device, train_loader, optimizer, epoch)
```

```
21
22              #get trnloss and testloss
23              trnloss, trnacc = eval_test(model, device, train_loader)
24              tstloss, tstacc = eval_test(model, device, test_loader)
25
26              #print trnloss and testloss
27              print('Epoch '+str(epoch)+': '+str(int(time.time()-start_time))+'
    s', end=', ')
28              print('trn_loss: {:.4f}, trn_acc: {:.2f}%'.format(trnloss, 100. *
     trnacc), end=', ')
29              print('test_loss: {:.4f}, test_acc: {:.2f}%'.format(tstloss, 100.
     * tstacc))
30
31          #save model
32          torch.save(model.state_dict(), os.path.join(args.model_dir, '
    final_model.pt'))
33
34  if __name__ == '__main__':
35      main()
```

4.5 Adversarial Examples

4.5.1 Fast Gradient Sign Method

Fast Gradient Sign Method [4] is able to find adversarial perturbations with a fixed L_∞-norm constraint. FGSM conducts a one-step modification to all pixel values so that the value of the loss function is increased under a certain L_∞-norm constraint. The authors claim that the linearity of the neural network classifier leads to the adversarial images because the adversarial examples are found by moving linearly along the reverse direction of the gradient of the cost function. Based on this linear explanation, [4] proposes an efficient linear approach to generate adversarial images. Let θ represents the model parameters, x, y denote the input and the label and $J(\theta, x, y)$ is the loss function. We can calculate adversarial perturbation r by

$$r = \epsilon \operatorname{sign}\left(\nabla_x J(\theta, x, y)\right) \tag{4.41}$$

A larger ϵ leads to a higher success rate of attacking, but potentially results in a bigger human visual difference. This attacking method has since been extended to a targeted and iterative version [8].

4.5.2 Practicals

First, we setup hyper-parameters (e.g., epsilon, steps, step size), device (e.g., CPU or GPU), and load training dataset (MNIST). Note that for FGSM: num-steps = 1 and step-size = 0.031; for PGD-20: num-steps = 20 and step-size = 0.003.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
import argparse
import time
import os
from torch.autograd import Variable

# Setup training parameters
parser = argparse.ArgumentParser(description='PyTorch MNIST Training')
parser.add_argument('--batch-size', type=int, default=128, metavar='N',
                    help='input batch size for training (default: 128)')
parser.add_argument('--test-batch-size', type=int, default=128, metavar='N',
                    help='input batch size for testing (default: 128)')
parser.add_argument('--lr', type=float, default=0.01, metavar='LR',
                    help='learning rate')
parser.add_argument('--no-cuda', action='store_true', default=False,
                    help='disables CUDA training')
parser.add_argument('--seed', type=int, default=1, metavar='S',
                    help='random seed (default: 1)')
parser.add_argument('--model-dir', default='./model-mnist-cnn',
                    help='directory of model for saving checkpoint')
parser.add_argument('--random', default=True,
                    help='random initialization for PGD')
```

```

# FGSM: num-steps:1 step-size:0.031   PGD-20: num-steps:20 step-size:0.003
parser.add_argument('--epsilon', default=0.031,
                    help='perturbation')
parser.add_argument('--num-steps', default=1,
                    help='perturb number of steps, FGSM: 1, PGD-20: 20')
parser.add_argument('--step-size', default=0.031,
                    help='perturb step size, FGSM: 0.031, PGD-20: 0.003')

args = parser.parse_args(args=[])

if not os.path.exists(args.model_dir):
    os.makedirs(args.model_dir)

# Judge cuda is available or not
use_cuda = not args.no_cuda and torch.cuda.is_available()
#device = torch.device("cuda" if use_cuda else "cpu")
device = torch.device("cpu")

torch.manual_seed(args.seed)
kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}

# Setup data loader
transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
        ])
trainset = datasets.MNIST('../data', train=True, download=True,
                    transform=transform)
testset = datasets.MNIST('../data', train=False,
                    transform=transform)
train_loader = torch.utils.data.DataLoader(trainset,batch_size=args.
    batch_size, shuffle=True,**kwargs)
test_loader = torch.utils.data.DataLoader(testset,batch_size=args.
    test_batch_size, shuffle=False, **kwargs)
```

```
# Define CNN
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # in_channels:1  out_channels:32  kernel_size:3  stride:1
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        # in_channels:32  out_channels:64  kernel_size:3  stride:1
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
```

```
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output
```

Then we define the function for FGSM/PGD attack.

```
def _pgd_whitebox(model,
                  X,
                  y,
                  epsilon=args.epsilon,
                  num_steps=args.num_steps,
                  step_size=args.step_size):
    out = model(X)
    err = (out.data.max(1)[1] != y.data).float().sum()
    X_pgd = Variable(X.data, requires_grad=True)
    if args.random:
        random_noise = torch.FloatTensor(*X_pgd.shape).uniform_(-epsilon, epsilon).to(device)
        X_pgd = Variable(X_pgd.data + random_noise, requires_grad=True)

    for _ in range(num_steps):
        opt = optim.SGD([X_pgd], lr=1e-3)
        opt.zero_grad()

        with torch.enable_grad():
            loss = nn.CrossEntropyLoss()(model(X_pgd), y)
        loss.backward()
        eta = step_size * X_pgd.grad.data.sign()
        X_pgd = Variable(X_pgd.data + eta, requires_grad=True)
        eta = torch.clamp(X_pgd.data - X.data, -epsilon, epsilon)
        X_pgd = Variable(X.data + eta, requires_grad=True)
        X_pgd = Variable(torch.clamp(X_pgd, 0, 1.0), requires_grad=True)
    err_pgd = (model(X_pgd).data.max(1)[1] != y.data).float().sum()
    return err, err_pgd

def eval_adv_test_whitebox(model, device, test_loader):
    # Ealuate model by white-box attack
    model.eval()
    robust_err_total = 0
    natural_err_total = 0

    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        # fgsm/pgd attack
        X, y = Variable(data, requires_grad=True), Variable(target)
        err_natural, err_robust = _pgd_whitebox(model, X, y)
        robust_err_total += err_robust
        natural_err_total += err_natural
    print('natural_accuracy: {:.2f}%'.format(0.01 * (10000-natural_err_total)))
    print('robust_accuracy: : {:.2f}%'.format(0.01 * (10000-robust_err_total)))
```

Finally, we load and attack the model.

```
def main():
    model = Net().to(device)
    model.load_state_dict(torch.load(os.path.join(args.model_dir, '
    final_model.pt')))
    eval_adv_test_whitebox(model, device, test_loader)
if __name__ == '__main__':
    main()
```

4.6 Exercise

Question 6. *Implement the perceptron learning algorithm in Section 4.1, and compare the obtained binary classifier with the one obtained from logistic regression.*

Question 7. *Understand how learning rate in the perceptron learning algorithm affects the learning results. Draw a curve to exhibit the change of accuracy with respect to the learning rate.*

Question 8. *Use the perceptron learning algorithm to work with the XOR dataset in Example 27, and check the accuracy.*

Question 9. *Assuming that all weights in Figure 4.5, i.e., those numbers 1 and -1, need to be learned. Can you adapt the perceptron learning algorithm to learn the weights?*

Question 10. *Understand the equivalence of applying matrix expression to compute the outputs for a dataset and the computation of outputs for individual inputs.*

5 Chapter 5: Loss Functions and Regularisation

This chapter introduces variants of loss functions, which are learning objectives, and several regularisation techniques, aiming to introduce inductive bias to the learning process. Based on them, we will also present some recent progress on adversarial training, aiming to train a deep learning model that is more robust to input perturbations.

Assume that, we have a model $f_{\mathbf{W}}$, being it a model whose parameters are just initialised or a model who appears during the training process. The dataset is $D = \{(\mathbf{x}_i, y_i) \mid i \in \{1..n\}\}$ is a labelled dataset. We are considering the classification task.

5.1 Loss Functions

In previous sections, we have introduced mean squared error (MSE), repeated as below, for linear regression. MSE is one of the most widely used loss functions.

$$\hat{L}(f_{\mathbf{W}}) = \frac{1}{m}\sum_{i=1}^{m}(f_{\mathbf{W}}(\mathbf{x}^{(i)}) - y^{(i)})^2 \tag{5.1}$$

Intuitively, MSE measures the average areas of the square created by the predicted and ground-truth points.

Mean Absolute Error

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}|f_{\mathbf{W}}(\mathbf{x}^{(i)}) - y^{(i)}| \tag{5.2}$$

Unlike MSE which concerns the areas of square, MAE concerns the geometrical distance between the predicted and ground-truth points. Comparing to MSE whose derivative can be easily computed, it is harder to compute derivative for MAE.

Root Mean Squared Error (RMSE) RMSE is very similar to MSE, except for the square root operation.

$$\hat{L}(f_{\mathbf{w}}) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(f_{\mathbf{W}}(\mathbf{x}^{(i)}) - y^{(i)})^2} \tag{5.3}$$

Binary Cross Entropy Cost Function When considering binary classification, i.e., $C = \{0, 1\}$, we may utilise information theoretical concepts, cross entropy, which measures the difference between two distributions for the predictions and the ground truths.

$$\hat{L}(f_{\mathbf{w}}) = \sum_{i=1}^{m} -y^{(i)} \log f_{\mathbf{W}}(\mathbf{x}^{(i)}) - (1 - y^{(i)}) \log(1 - f_{\mathbf{W}}(\mathbf{x}^{(i)})) \tag{5.4}$$

where Cross entropy loss works better after the softmax layer, because the output of the softmax layer represents a distribution.

Categorical Cross Entropy Cost Function Extending the above to multiple classes, we may have

$$\hat{L}(f_{\mathbf{w}}) = -\sum_{i=1}^{m} \sum_{c \in C} \mathbf{y}_c^{(i)} \log[f_{\mathbf{W}}(\mathbf{x}^{(i)})]_c \tag{5.5}$$

where $\mathbf{y}^{(i)}$ is the one-hot representation of the ground truth $y^{(i)}$ and $\mathbf{y}_c^{(i)}$ denotes the component of $\mathbf{y}^{(i)}$ that is for the class c. Also, unlike the previous notations, $f_{\mathbf{W}}(\mathbf{x}^{(i)})$ is a probability distribution of the prediction over $\mathbf{x}^{(i)}$, and $[f_{\mathbf{W}}(\mathbf{x}^{(i)})]_c$ denotes the component of $f_{\mathbf{W}}(\mathbf{x}^{(i)})$ that is for the class c.

5.2 Regularisation Techniques

As we have seen in the previous chapters that most machine learning algorithms are to optimise the loss between ground truths and predictions. For example, as suggested in Equation (3.24), the linear regression is to minimise

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{w}^T\mathbf{x}^{(i)} - y^{(i)})^2 \tag{5.6}$$

and, as suggested in Equation (4.8), the convolutional neural network is to minimise

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(f_{\mathbf{w}}(\mathbf{x}^{(i)}) - y^{(i)})^2 \tag{5.7}$$

when taking the mean square error as the loss function. Based on such optimisation objectives, stochastic gradient descent based methods are applied to search for the optimal solutions. When the problem is relatively simple, e.g., the number of parameters is small, this may lead to optimal solution. However, this might not work well and it is very easy to over-fit the model when the problem is complex.

For the complex cases, it is needed to reduce the model complexity by applying regularisation techniques. In the following, we introduce a few regularisation techniques that have been widely used.

5.2.1 Ridge Regularisation

For ridge regularisation, the loss function is updated by having a penalty term, i.e.,

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(f_{\mathbf{w}}(\mathbf{x}^{(i)}) - y^{(i)})^2 + \lambda\sum_{w\in\mathbf{W}} w^2 \tag{5.8}$$

where the term $\sum_{w\in\mathbf{W}} w^2$ is the square of the magnitude of the coefficients, and λ is a hyper-parameters balancing between learning loss and the penalty term. According to the definition, the ridge regularisation reduces the model complexity and multicollinearity.

5.2.2 Lasso Regularisation

For lasso (least absolute shrinkage and selection operator) regularisation, the loss function is updated by having a penalty term, i.e.,

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(f_{\mathbf{w}}(\mathbf{x}^{(i)}) - y^{(i)})^2 + \lambda\sum_{w\in\mathbf{W}} |w| \tag{5.9}$$

that is, instead of taking squared coefficients, we consider the absolute value of the coefficients. According to the definition, the lasso regularisation encourages the selectivity of features, i.e., make the weight matrix sparser.

5.2.3 Dropout

Dropout [12] is a regularisation technique to reduce the complex co-adaptations of training data. Essentially, it randomly ignores, or drop out, a certain percentage of the layer output during the training.

Dropout can be used on most types of layers, such as fully connected layers, convolutional layers, and the long short-term memory network layers. It may be applied to any or all hidden layers as well as the input layer, but not on the output layer. Dropout is a training technique, and is not used when making a prediction, i.e., after training.

5.2.4 Early Stopping

Early stopping is to use a holdout validation dataset to evaluate whether the training procedure should be terminated to prevent the increase of generalisation error. In general, it is applied when the performance of the model on the validation dataset starts to degrade (e.g. loss begins to increase or accuracy begins to decrease).

5.2.5 Batch-Normalisation

a normalization step that fixes the means and variances of each layer's inputs

5.3 Adversarial Training

Adversarial training is an effective approach to improve the robustness of a deep learning model. The general idea is to create and incorporate adversarial examples into the training process. Surely, given there are many different ways of generating adversarial examples and many different ways of incorporating them in training, this general idea can be implemented in many different ways. The following is a typical format on how to conduct adversarial training.

$$\min_{\mathbf{W}} \sum_{(\mathbf{x},y)\in D} \max_{\boldsymbol{\delta}\in\Delta(\mathbf{x})} L(f_{\mathbf{W}}(\mathbf{x}+\boldsymbol{\delta}), y) \tag{5.10}$$

Intuitively, it is a min-max process, where each learning batch is conducted by first selecting the worst-case adversarial perturbation during the *inner maximisation*, and then adapting weights to reduce the loss by the adversarial perturbation in *outer minimisation*.

5.3.1 Training with Adversarial Examples

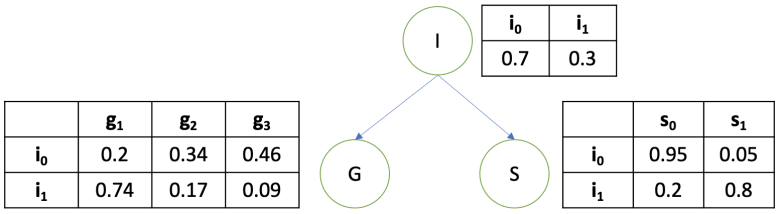
5.3.2 Label Smoothing

6 Chapter 6: Probabilistic Graph Models

Up to now, we have known that machine learning algorithms can be used to effectively learn a function f from a set of input-output pairs. The function f approximates the relation between two random variables X and Y, and actually expresses the conditional probability $P_f(Y|X)$. However, in a complex, real world system, there might be more than two random variables and it is useful to not only understand the the conditional probability between random variables but also be able to infer more intriguing information from the conditional dependence relations between random variables. It is also possible that, a complex machine learning model, such as a convolutional neural network, can be approximated by constructing the conditional dependencies between a set of random variables representing the features learned by the neural networks, see e.g., [1] for an example. Therefore, while it is agreeable that machine learning has been able to support human operators in dealing with some long-standing tasks such as object detection and recognition with accuracy and efficiency, there is still a need to infer useful knowledge from a set of conditional probabilities.

Probabilistic graphical models are a formalism for the above purpose. They use a structure, or more specifically a graph, to represent conditional dependence relations between random variables, and use a probability table for every random variable to express the local dependence relation of the random variable. There are two major branches of graphical models, namely, Bayesian networks and Markov random fields, and in this chapter, we focus on Bayesian network. In the following, we will use (probabilistic) graphical models and Bayesian network interchangeably.

Depending on the dependence relation of individual random variable, the probability tables in a graphical model can be a marginal probability table, which shows that the random variable does not depend on any other variables in the graph, or a conditional probability table, which represents the conditional probability distribution of the current random variable over other random variables in the graph.



	i_0	i_1
	0.7	0.3

	g_1	g_2	g_3
i_0	0.2	0.34	0.46
i_1	0.74	0.17	0.09

	s_0	s_1
i_0	0.95	0.05
i_1	0.2	0.8

Figure 6.1: A simple graphical model of three nodes

Figure 6.1 presents a simple graphical model with 3 nodes: I, G, S. We can see that one of the nodes I has a marginal probability table while the remaining two nodes have conditional

probability tables. Summarising,

$$\text{Probabilistic Graphical Model} = \text{Graphical Structure} + \text{Multivariate Statistics} \tag{6.1}$$

Formally, a probabilistic graphical model $G = (\mathcal{V}, \mathcal{E}, P)$ where $\mathcal{V}$ is a set of nodes, representing the random variables, $\mathcal{E}$ is the set of edges between nodes, and P is a set of probability tables, one for each node in $\mathcal{V}$.

6.1 A Running Example

Assume that, on a self-driving car, there are two sensors, $Camera$ and $Radar$, that are used to detect pedestrian collectively. The precision of the camera may be affected by weather conditions, such as the Fog as we consider in this example. The $Radar$ may be affected by the distance of the object from the car, i.e., it can be very precise when the object is close but may become less precise when the object is $Away$. Once a pedestrian is detected and it is not away, the car will need to stop.

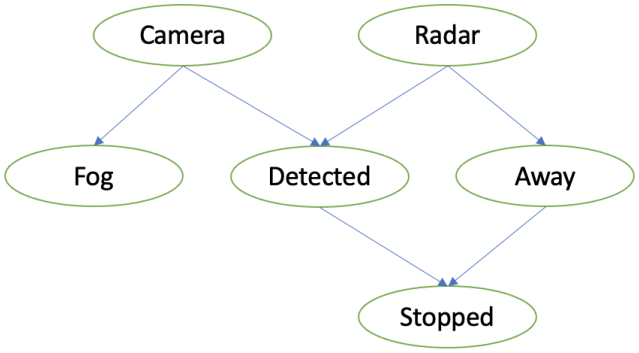


Figure 6.2: A simple Bayesian network for safety analysis on vehicle stopping upon pedestrian detection

Figure 6.2 presents a probabilistic graphical model for this example. In the graph G, there are six random variables: $Camera$, $Radar$, Fog, $Detected$, $Away$, and $Stopped$. Every node is associated with either a marginal probability table or a conditional probability table, depending on whether they have incoming edges. The information about the probability tables are given in Figure 6.3. For example, the nodes $Camera$ and $Radar$ do not have incoming edges, so each of them is associated with a marginal probability table. Intuitively, the two tables suggest that the probability of a pedestrian appearing in the imagery input of the camera is 0.4, and in the signal input of the radar is 0.5. Note that, the "appearing" is for ground truth (through human's eyes), not for the result of a detection system. The detection is implemented through the $Detected$ node to be explained below.

Other nodes are associated with conditional probability tables. For example, the table for Fog shows that, when there is no pedestrian appearing in the imagery input, the probability of the foggy weather condition is 0.5. This probability is lowered when there is a pedestrian

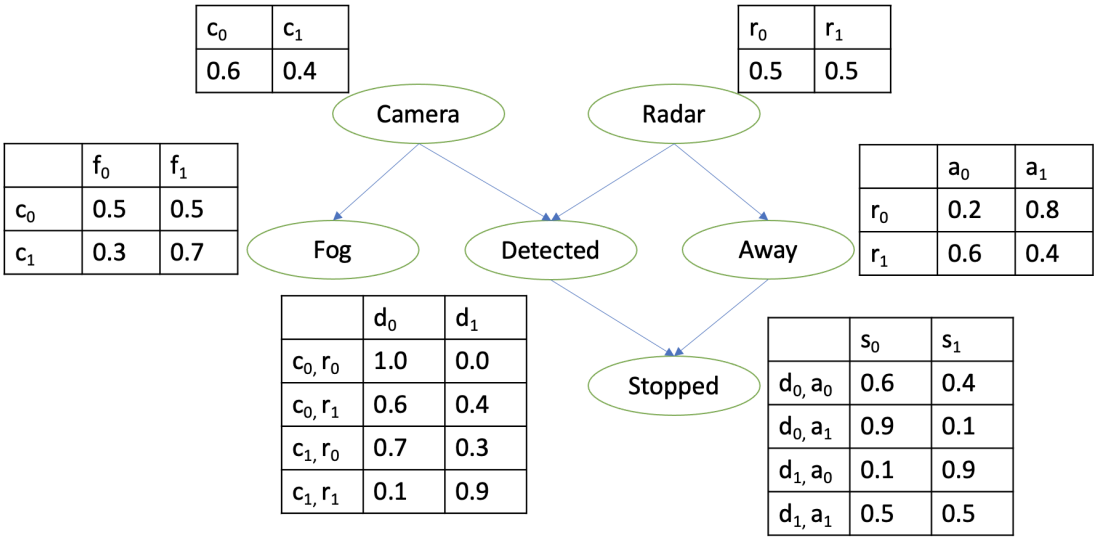


Figure 6.3: Probabilistic table of the graphical model in Figure 6.2

appearing in the imagery input. This is intuitive, because the foggy condition may affect the ability of camera capturing the pedestrian. Similar for the $Away$ node. When there is no pedestrian appearing in the signal input, the probability of its away from a pedestrian is 0.8. This probability is lowered to 0.4 when there is a pedestrian appearing in the signal input.

The detection result is a fusion of both camera and radar's results. Note that, even if a pedestrian appears in the imagery input, it does not mean that the pedestrian can be detected (Recall the generalisation error and robustness error of deep learning). We note that, if neither of the sensors has a pedestrian appeared, no detection can be made at all. If one of the sensors has a pedestrian, there is a non-trivial chance that it can be detected. The detection becomes significantly better when both sensors captured the pedestrian.

Finally, the decision making on whether the car should be stopped is based on both the detection result and the distance. If it is detected (i.e., $Detected = d_1$) and not far away (i.e., $Away = a_0$) then this probability is high (0.9). Otherwise, the probability is low (0.1). The lowest probability appears when no pedestrian is detected (i.e., $Detected = d_0$) and it is away (i.e., $Away = a_1$).

Where does machine learning play a role? Machine learning can be used to generate those conditional probability tables. For example, a deep learning model can be designed and trained to get the table for $Detected$ node, i.e., classify whether a pedestrian is detected or not on both the camera input and the radar input. Similarly, other nodes such as Fog, $Away$, and $Stopped$ may also be implemented with a machine learning model.

6.2 I-Maps

This section studies the conditional independences in G.

6.2.1 Naive Bayes and Joint Probability

First of all, we explain the difference between usual classification and Naive Bayes classifier. As shown in Figure 6.4, usual classification can be represented as a graphical model G where the class label Y receives incoming connections from the feature variables $X_1, ..., X_n$. Therefore, Y has a conditional probabilistic table $P(Y|X_1, ..., X_n)$ and the feature variables X_i has a marginal probability table $P(X_i)$. However, for Naive Bayes classifier, it can be represented

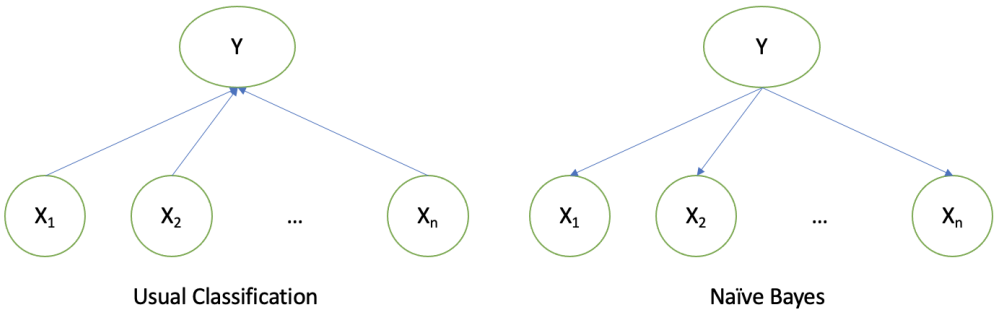


Figure 6.4: Naive Bayes as a probabilistic graphical model

as another graphical model G' where the label Y has outgoing arrows to the feature variables $X_1, ..., X_n$. Therefore, each feature variable X_i has a conditional probability table $P(X_i|Y)$ and the class label has a marginal probability table $P(Y)$. As we explained earlier, for naive Bayes, we have

$$P(X_1, ..., X_n, Y) = P(Y) \prod_{i \in \{1..n\}} P(X_i|Y) \tag{6.2}$$

That is, the joint probability is the product of probability tables for the nodes on the graph. Generalising the case for Naive Bayes classifier, we conjecture that for each probabilistic graphical model $G = (\mathcal{V}, \mathcal{E}, P)$ represent the joint probability distribution between random variables, we may have

$$P(\mathcal{V}) = \prod_{V \in \mathcal{V}} P(V|Pa(V)) \tag{6.3}$$

where $Pa(V)$ is the set of parent nodes of node V. Note that, we let $P(V|\emptyset) = P(V)$ when the node V does not have incoming edges.

6.2.2 Independencies in a Distribution

Before proceeding to the conditional probabilities in G, we may need to know how to find conditional independencies from a joint probability. For this, Section 1.2.3 has presented a few examples on how to do the calculation.

6.2.3 Markov Assumption

In addition to the notation $Pa(X)$ for the set of parents of a node X, we need $NonDesc(X)$ for the set of non-descendents of X. We have that

each random variable X is independent of its non-descendents, given its parents, i.e.,

$$X \perp NonDesc(X) | Pa(X) \tag{6.4}$$

Intuitively, parents of a variable shield it from probabilistic influence. Once values of parents are known, no influence of ancestors can be made. On the other hand, information about descendants can change beliefs about a node .

For the running example in Figure 6.2, we have the following local conditional independences that can be read directly from the graph:

$$\begin{aligned} I(G) \quad = \{ \quad & (Fog \perp Detected, Radar, Away, Stopped | Camera), \\ & (Camera \perp Radar, Away), \\ & (Radar \perp Camera, Fog), \\ & (Detected \perp Away, Fog | Camera, Radar), \\ & (Away \perp Camera, Fog, Detected | Radar), \\ & (Stopped \perp Fog, Camera, Radar | Detected, Away)\} \end{aligned} \tag{6.5}$$

Intuitively, to understand $(Fog \perp Detected, Radar, Away, Stopped | Camera)$, we note that, once we are able to observe the camera's result, the weather conditions such as Fog can be directly obtained and are independent from other random variables. Moreover, to understand $(Camera \perp Radar, Away)$, we note that camera's precision – as a quality of hardware – is absolutely independent of both radar's result and whether or not the pedestrian is away. Other (conditional) independencies can be explained in a similar way.

We remark that, the conditional independencies obtained through this way are local ones, and do not necessarily include all conditional independencies that can be inferred from the graph. In Section 6.4, we will introduce a comprehensive method – d-separation – that is able to infer all possible conditional independencies from a graph.

6.2.4 I-Map of Graph and Factorisation of Joint Distribution

Let G be a graph associated with a set of independencies $I(G)$, and P a probability distribution with a set of independencies $I(P)$. Note that, $I(P)$ can be obtained by the way shown in Section 6.2.2. Then, we define

Definition 12. *G is an I-Map of P if $I(G) \subseteq I(P)$.*

By this definition, I-Map requires that a joint distribution P can have more independencies than the graph G, but graph G cannot mislead by containing independencies that do not exist in P.

Example 28. *Consider the joint probability table P as in Table 1.3, and the three graphs in Figure 6.5. we note that $I(P) = \{X \perp Y\}$, $I(G_0) = \{X \perp Y\}$, and $I(G_1) = I(G_2) = \emptyset$.*

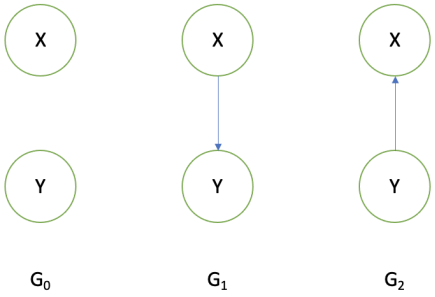


Figure 6.5: Three simple, two-node graphs

Therefore, all three graphs G_0, G_1, G_2 are I-maps of P. On the other hand, if consider the joint probability table P as in Table 1.4, we have that $I(P) = \emptyset$. Then, while G_1 and G_2 are still I-maps of P, G_0 is not any more.

In the following, we introduce the relationship between I-map and factorisation. Factorization is to write a mathematical object as a product of several, usually smaller or simpler, objects of the same kind. For example, in the Naive Bayes classifier, the joint distribution $P(X_1, ..., X_n, Y)$ is rewritten as the production of $P(Y)$ and conditional probabilities $P(X_i|Y)$, where $P(Y)$ and $P(X_i|Y)$ are much smaller than $P(X_1, ..., X_n, Y)$. For the graphical model in general, we have

Theorem 1. *If G is an I-map of P, then*

$$P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i|Pa(X_i)) \tag{6.6}$$

This justifies our conjecture at Equation (6.3).

6.2.5 Perfect Map

Similar as I-map, we may define D-map, which requires that $I(P) \subseteq I(G)$. The intersection of I-map and D-map leads to perfect map, which requires that the conditional independencies in G and P. Interestingly, but not surprisingly, not all distributions P over a given set of variables can be represented as a perfect map.

6.3 Reasoning Patterns

As explained earlier, the construction of graphical models will enable our reasoning about a more complex relation between a set of random variables. In this section, we introduce a few typical reasoning patterns.

6.3.1 Causal Reasoning

Causal reasoning considers how the changes of up-stream variables may affect the values of the down-stream variables. For our running example as in Figure 6.3, it is for an engineer to concern about how likely the car will stop given e.g., the quality of sensors (i.e., camera and/or radar).

A typical process is to first compute a marginal probability such as

$$P(s_0) \tag{6.7}$$

which is the probability of the car does not stop. This can be computed by having

$$\begin{aligned} P(s_0) = & \sum_{C,R,F,D,A} P(C,R,F,D,A,S=s_0) \\ = & \sum_{i=0}^{1}\sum_{j=0}^{1}\sum_{k=0}^{1}\sum_{l=0}^{1}\sum_{m=0}^{1} P(C=c_i,R=r_j,F=f_k,D=d_l,A=a_m,S=s_0) \\ \approx & \ 0.62 \end{aligned} \tag{6.8}$$

Then, we may consider a conditional probability such as

$$P(s_0|c_1) = \frac{P(s_0,c_1)}{P(c_1)} \approx 0.51 \tag{6.9}$$

which says that if we know that the camera captured the pedestrian, the probability of not stopped decreased to 0.33. Similarly, if consider radar, we have

$$P(s_0|r_1) = \frac{P(s_0,c_1)}{P(r_1)} \approx 0.44 \tag{6.10}$$

If both the camera and the radar are considered, we have

$$P(s_0|c_1,r_1) = \frac{P(s_0,c_1,r_1)}{P(c_1,r_1)} \approx 0.31 \tag{6.11}$$

6.3.2 Evidential Reasoning

A driver may want to know, subject to her own experience on e.g., car stopping and foggy weather, about the quality of the sensors. For example, first of all, she may have the following statistics from the vendor of the camera:

$$P(c_1) = 0.4 \tag{6.12}$$

After observing that the car stopped, she might infer as follows, which shows that the probability increased to 0.51.

$$P(c_1|s_1) = \frac{P(c_1, s_1)}{P(s_1)} \approx 0.51 \tag{6.13}$$

Intuitively, this may suggest that the specific camera installed on this car may perform above average.

6.3.3 Inter-causal Reasoning

In addition to the above, it might be interested to understand how the causes of an event may affect each other. For example, consider the following

$$P(r_1|d_1) \approx 0.83 \tag{6.14}$$

which suggests that once we know that the pedestrian is detected, the chance of radar captured the pedestrian is high, i.e., the quality of the radar is good. However, by having the following

$$P(r_1|d_1, c_1) \approx 0.75 \tag{6.15}$$

which is lower than $P(r_1|d_1)$, it suggests that the quality of the radar may not be as optimistic as it seems when only observing the detection result. The quality of the camera may also contribute well to the excellent detection result.

6.3.4 Practicals

First of all, we install a software package that can support the inference of probabilistic graphical models:

```
$ conda install pomegranate
```

To work with the package, we create a script **pedestrian_detection.txt**. In the script, first of all, we import the package:

```
from pomegranate import *
```

Then, we encode a graphical model

```
camera = DiscreteDistribution({'0': 0.6, '1': 0.4})
radar = DiscreteDistribution({'0': 0.5, '1': 0.5})
fog = ConditionalProbabilityTable(
        [['0', '0', 0.5],
         ['0', '1', 0.5],
         ['1', '0', 0.3],
         ['1', '1', 0.7]], [camera])
away = ConditionalProbabilityTable(
        [['0', '0', 0.2],
         ['0', '1', 0.8],
         ['1', '0', 0.6],
         ['1', '1', 0.4]], [radar])
detected = ConditionalProbabilityTable(
        [['0', '0', '0', 1.0],
```

```
         ['0', '0', '1', 0.0],
         ['0', '1', '0', 0.6],
         ['0', '1', '1', 0.4],
         ['1', '0', '0', 0.7],
         ['1', '0', '1', 0.3],
         ['1', '1', '0', 0.1],
         ['1', '1', '1', 0.9]], [camera, radar])
stopped = ConditionalProbabilityTable(
        [['0', '0', '0', 0.6],
         ['0', '0', '1', 0.4],
         ['0', '1', '0', 0.9],
         ['0', '1', '1', 0.1],
         ['1', '0', '0', 0.1],
         ['1', '0', '1', 0.9],
         ['1', '1', '0', 0.5],
         ['1', '1', '1', 0.5]], [detected, away])

s1 = Node(camera, name="camera")
s2 = Node(radar, name="radar")
s3 = Node(fog, name="fog")
s4 = Node(away, name="away")
s5 = Node(detected, name="detected")
s6 = Node(stopped, name="stopped")

model = BayesianNetwork("Pedestrian Detection Problem")
model.add_states(s1, s2, s3, s4, s5, s6)
model.add_edge(s1, s3)
model.add_edge(s1, s5)
model.add_edge(s2, s4)
model.add_edge(s2, s5)
model.add_edge(s5, s6)
model.add_edge(s4, s6)
model.bake()
```

After the above, we can start computing the probability values:

```
#### P(s0,c1)
query = ['1', None, None, None, None, '0']
ps0c1 = 0
for j1 in range(2):
    for j2 in range(2):
        for j3 in range(2):
            for j4 in range(2):
                ps0c1 += model.probability([['1', str(j1), str(j2), str(j3),
    str(j4), '0']])
print("the probability of the car does not stop but the camera captured the
    pedestrian P(s0,c1): %s\n"%(ps0c1))

### P(c1)
query = ['1', None, None, None, None, None]
pc1 = 0
for j1 in range(2):
    for j2 in range(2):
        for j3 in range(2):
            for j4 in range(2):
```

```
                for j5 in range(2):
                    pc1 += model.probability([['1', str(j1), str(j2), str(j3)
    , str(j4), str(j5)]])
print("the probability of camera captured the pedestrian P(c1): %s\n"%pc1)

#### P(s0|c1)
print("the conditional probability of the car does not stop when the camera
    captured the pedestrian P(s0|c1)): %s\n"%(ps0c1/pc1))
```

6.4 D-Separation

From Section 6.2, we know that a graph structure G encodes a set of conditional independence assumptions $I(G)$ and we can read a set of independencies directly according to the Markov assumption. However, we may be interested to infer all possible conditional independence from a graph G.

Definition 13. *D-separation is a procedure d-sep$_G(X \perp Y|Z)$ that, given a graph G and three sets X, Y, and Z of nodes in G, returns Yes or No, such that d-sep$_G(X \perp Y|Z) = Yes$ iff $(X \perp Y|Z)$ follows from $I(G)$.*

First of all, we note that if X and Y are connected directly, they are co-related regardless of any evidence about any other variables. So, in the following, we consider X and Y that are not directly connected.

6.4.1 Four Local Triplets

Before considering more complex cases, we consider four local cases where X and Y are indirectly connected with another variable Z in the middle. The four possible cases are shown in Figure 6.6.

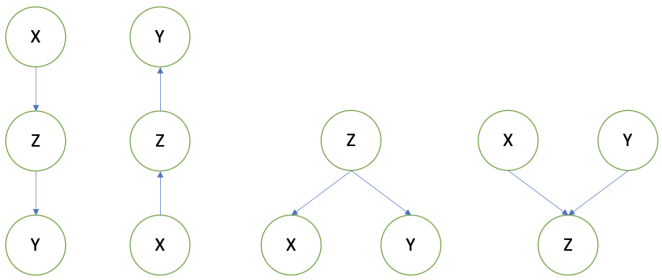


Figure 6.6: Four patterns

Indirect Causal Effect $X \rightarrow Z \rightarrow Y$ Cause X cannot influence effect Y if Z is observed, i.e., observed Z blocks the influence of X over Y. For the running example, as shown in Figure 6.7,

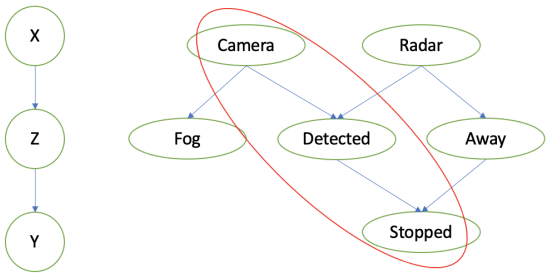


Figure 6.7: Indirect Causal Effect

Indirect Evidential Effect $Y \to Z \to X$ Similarly, evidence X cannot influence the cause Y if Z is observed.

Common Cause $X \leftarrow Z \to Y$ Once Z is observed, one of the causes cannot influence the other. Figure 6.8 presents a case of common cause in our running example.

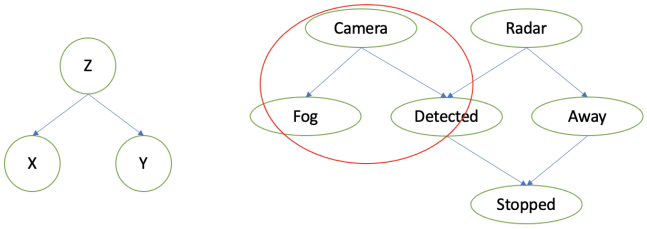


Figure 6.8: Common Cause

Common Effect $X \to Z \leftarrow Y$ Unlike the above three cases where observing the middle variable Z blocks the influence, the case of common effect is on the opposite, i.e., the influence is blocked when the common effect Z and its descendants are not observed. Figure 6.9 presents a case of common effect in our running example.

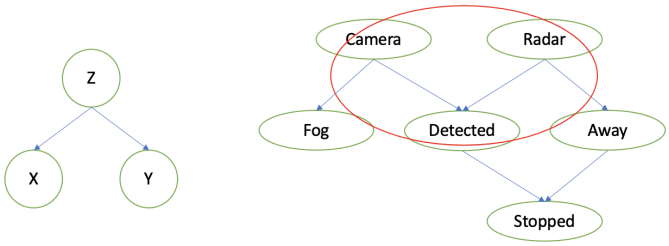


Figure 6.9: Common Effect

6.4.2 General Case: Active Trail and D-Separation

Summarising the above discussion on local canonical cases (or v-structures), we have that

- Causal trail, $X \to Z \to Y$, is active if and only if Z is not observed.
- Evidential trail, $X \leftarrow Z \leftarrow Y$, is active if and only if Z is not observed.
- Common cause, $X \leftarrow Z \to Y$, is active if and only if Z is not observed.
- Common effect, $X \to Z \leftarrow Y$, is active if and only if either Z or one of its descendants is observed.

Now we can consider the general case of d-sep$_G(X \perp Y | Z)$, where X, Y and Z may not be connected to each other. Actually, the graph can be large and the variables may be far away

from each other. Fortunately, as we will introduce below, any complex case can be broken into repetitions of the local canonical cases.

The D-separation algorithm is given in Algorithm 6. It collects all paths between X and Y, without considering the directions of the edges. Then, as long as no paths are active given the observations $\{Z_1, ..., Z_k\}$, the two variables X and Y are conditionally independent.

Algorithm 6: ***d-sep***$_G(X, Y, \{Z_1, ..., Z_k\})$, where X, Y, Z_i are nodes on a graph G

```
Path = all (undirected) paths from X to Y
for path in Path do
    if isActive(path, {Z_1, ..., Z_k}) then
        return X ⫫̸ Y | {Z_1, ..., Z_k}
    else
end
return X ⊥ Y | {Z_1, ..., Z_k}
```

Now, we need to determine whether a path is active or not, i.e., ***isActive***(path, $\{Z_1, ..., Z_k\}$), which can be done with Algorithm 7. Simply speaking, it requires all local triplets on the path to be inactive to make the path inactive.

Algorithm 7: ***isActive***$(path, \{Z_1, ..., Z_k\})$, where $path$ is a path on the graph and Z_i are nodes on G

```
Let path = X_1, ..., X_k
for all triplets X_{i-1}, X_i, X_{i+1} on path do
    if (X_{i-1}, X_i, X_{i+1}) is inactive then
        return False
    else
end
return True
```

I-equivalence Conditional independence assertions can be the same with different graphical structures, and I-equivalence is to capture such equivalence relation.

Definition 14. *Two graphs G_1 and G_2 are I-equivalent if $I(G_1) = I(G_2)$.*

Let skeleton of a graph G be an undirected graph with an edge for every edge in G. We have that, if two graphs have the same set of skeletons and v-structures then they are I-equivalent.

6.5 Structure Learning

There are mainly two approaches to acquire a graphical model. The first is through knowledge engineering, where the graphical model is constructed by hand with expert's help. The second is through machine learning, by which the graphical model is learned from a set of instances.

First of all, the following is the problem statement for structure learning:

> Assume dataset D is generated i.i.d. from distribution $P^*(X)$, and $P^*(X)$ is induced by an underlying graph G^*. The goal is to construct a graphical model G such that it is as close as possible to G^*.

Considering that there may be many I-maps for P^* and we cannot distinguish them from D, we know that G^* is not identifiable. Nevertheless, in most cases, it is sufficient to recover G^*'s equivalence class.

6.5.1 Criteria of Structure Learning

While our goal is to recover G^*'s equivalence class, this goal is nontrivial. One of the key issues is that the data instances sampled from the distribution $P^*(X)$ may be noisy. Therefore, we need to make decisions about including edges we are less sure about. Actually, too few edges means the possibility of missing out on dependencies, and too many edges means the possibility of spurious dependencies.

Example 29. *In a coin tossing example, we have two coins X and Y, who are tossed independently. Assume that they are tossed 100 times, with the statistics shown in Table 6.1. Are X and Y independent? Actually, if we follow the exact computation, they are dependent. But we suspect that*

X	Y	times
head	head	27
head	tail	22
tail	head	25
tail	head	26

Table 6.1: A simple coin tossing example

they should be independent, because the probability of getting exactly 25 in each category is small (approx. 1 in 1,000).

Actually, in a conditional probability table, the number of entries grows exponentially with the of parent nodes. Therefore, the cost of adding a parent node can be very large. According to this, it is better to obtain a sparser, simpler structure. Actually, we can sometimes learn a better model by learning a model with fewer edges even if it does not represent the true distribution.

6.5.2 Overview of Structure Learning Algorithms

Basically, there are two classes of structural learning algorithms:

- Constraint-based algorithms
- Score-based algorithms

Constraint-based algorithms are to find a graphical model whose implied independence constraints match those found in the data. On the other hand, score-based algorithms are to find a graphical model that can represent distributions that match the data (i.e., could have generated the data).

In the following, we introduce two different directions of structural learning, by the consideration of learning local relations and global structure, respectively.

Local: Independence Tests For testing the independence between variables, we need measures of "deviance-from-independence" and rules for accepting/rejecting hypothesis of independence. For the measures, we may consider e.g., mutual Information (K-L divergence) between joint and product of marginals, by having

$$d_I(D) = \frac{1}{|D|} \sum_{x_i, x_j} P(x_i, x_j) \log \frac{P(x_i, x_j)}{P(x_i)P(x_j)} \tag{6.16}$$

for any two variables X_i and X_j. Theoretically, $d_I(D) = 0$ if X_i and X_j are independent, and $d_I(D) > 0$ otherwise. In addition to $d_I(D)$, there may be other means to define the measures, such as Pearson's Chi-squared test.

Based on the measure, we can define the acceptance rule as

$$R_{d,t}(D) = \begin{cases} \text{Accept} & d(D) \leq t \\ \text{Reject} & d(D) > t \end{cases} \tag{6.17}$$

where t is a pre-specified threshold. We remark that false rejection probability due to choice of t is its p-value (refer to hypothesis testing). Alternatively, we may take Chow-Liu algorithm to construct a tree-like graphical model as follows:

1. find maximum weight spanning tree based on the values we compute in Equation (6.16); there are existing algorithms for this purpose, such as the Kruskal's algorithm and the Prim's algorithm.
2. pick a root node and assign edge directions.

Global: Structure Scoring Similarly as the local case, we need measures to evaluate the goodness of a graphical model and rules for accepting/rejecting hypothesis of goodness. There are different ways to define measures, including

- Log-likelihood Score for G with n variables

$$Score_L(G, D) = \sum_{D} \sum_{i=1}^{n} \log P(x_i | pa(x_i)) \tag{6.18}$$

  which can be seen as the loss of using graph G to predict D.

- Bayesian Score

$$Score_B(G, D) = \log P(D|G) + \log P(G) \tag{6.19}$$

  which, in addition to the log-likelihood score $\log P(D|G)$, it also consider the prior $P(G)$.
- Bayes score with penalty term

$$Score_{BIC}(G, D) = L(G, D) - \frac{\log |D|}{2}||G|| \tag{6.20}$$

  where $L(G, D)$ is the loss of using G to predict D, and $||G||$ is the complexity of the graph G.

Once defined the scores, the structure learning is to search for a graphical structure with the highest score. It is known that finding the optimal one among those structures with at most k parents is NP-hard for $k > 1$. To deal with the high complexity, there are multiple methods including e.g.,

- Greedy search
- Greedy search with restarts
- MCMC methods

For the greedy search, it repeatedly does the following:

1. score all possible single changes, and
2. select the best change to apply if there are any changes that lead to better performance than the existing structure.

6.5.3 Practicals

The following code uses the **pomegranate** package to automatically learn structures from the **digits** dataset.

```
from pomegranate import BayesianNetwork
import seaborn, time, numpy, matplotlib
seaborn.set_style('whitegrid')

from sklearn.datasets import load_digits
X, y = load_digits(10, True)

tic = time.time()
model = BayesianNetwork.from_samples(X)
t = time.time() - tic
p = model.log_probability(X).sum()
print("Greedy")
print("Time (s): ", t)
print("P(D|M): ", p)
model.plot('model-greedy')

tic = time.time()
model = BayesianNetwork.from_samples(X, algorithm='exact-dp')
t = time.time() - tic
```

```
p = model.log_probability(X).sum()
print("exact-dp")
print("Time (s): ", t)
print("P(D|M): ", p)
model.plot('model-exact-dp')

tic = time.time()
model = BayesianNetwork.from_samples(X, algorithm='exact')
t = time.time() - tic
p = model.log_probability(X).sum()
print("exact")
print("Time (s): ", t)
print("P(D|M): ", p)
model.plot('model-exact')
```

6.6 Sensitivity Analysis

As suggested earlier, it is possible that some probability tables of a graphical model can be obtained through machine learning. However, considering that a machine learning may be subject to various safety risks, it is useful to study how such safety risks on machine learning learning may affect the inference on the graphical model. In particular, we consider the following problem:

> Given a certain constraint on the perturbations and an MAP query, it is to determine whether or not the perturbed values may lead to a different decision about the MAP query.

The following is an example from [2].

Example 30. *Consider a simple two-node graphical model* $G = (\{X_1, X_2\}, \{X_1 \to X_2\}, P)$*, where* $P(X_1 = 1) = 0.45$ *and* $P(X_2 = 1|X_1 = 1) = 0.2$ *and* $P(X_2 = 1|X_1 = 0) = 0.9$*. We have that*

$$MAP(G) = (X_1 = 1, X_2 = 0) \tag{6.21}$$

Now, if consider the following perturbation constraint

$$|P(X_1 = 1) - 0.45| \leq 0.06 \tag{6.22}$$

does the assignment $(X_1 = 1, X_2 = 0)$ *continue to be the* $MAP(G)$*?*

6.7 Exercise

Question 11. *Try to explain another inter-causal reasoning in Figure 6.3, over the three variables* $Detected$, $Stopped$, *and* $Away$.

Question 12. *Implement the Chow-Liu algorithm for the* ***iris*** *dataset.*

Question 13. *Can you propose a solution for the sensitivity analysis?*

7 Chapter 7: Competitions

7.1 Competition 1: Resilience to Adversarial Attack

This is a student competition to address two key issues in modern deep learning, i.e.,

- O1 how to find better adversarial attacks, and
- O2 how to train a deep learning model with better robustness to the adversarial attacks.

We provide a template code (**Chapter_7.py**), where there are two code blocks corresponding to the training and the attack, respectively. The two code blocks are filled with the simplest implementations representing the baseline methods, and the participators are expected to replace the baseline methods with their own implementations, in order to achieve better performance regarding the above O1 and O2.

7.1.1 Submissions

In the end, we will collect submissions from the students and rank them according to a pre-specified metric taking into consideration both O1 and O2. Assume that we have n students participating in this competition, and we have a set S of submissions.

Every student with student number i will submit a package i**.zip**, which includes two files:

1. i**.pt**, which is the file to save the trained model, and
2. competition_i**.py**, which is your script after updating the two code blocks in **Chapter_7.py** with your implementations.

NB: Please carefully follow the naming convention as indicated above, and we will not accept submissions which do not follow the naming convention.

7.1.2 Source Code

The template source code of the competition is available at

https://github.com/xiaoweih/AISafetyLectureNotes/tree/main/Chapter-7

In the following, we will explain each part of the code.

Load packages First of all, the following code piece imports a few packages that are needed.

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import torch.optim as optim
import torchvision
from torchvision import transforms
from torch.autograd import Variable
import argparse
import time
```

Note: You can add necessary packages for your implementation.

Define competition ID The below line of code defines the student number. By replacing it with your own student number, it will automatically output the file i**.pt** once you trained a model.

```
# input id
id_ = 1000
```

Set training parameters The following is to set the hyper-parameters for training. It considers e.g., batch size, number of epochs, whether to use CUDA, learning rate, and random seed. You may change them if needed.

```
# setup training parameters
parser = argparse.ArgumentParser(description='PyTorch MNIST Training')
parser.add_argument('--batch-size', type=int, default=128, metavar='N',
                    help='input batch size for training (default: 128)')
parser.add_argument('--test-batch-size', type=int, default=128, metavar='N',
                    help='input batch size for testing (default: 128)')
parser.add_argument('--epochs', type=int, default=10, metavar='N',
                    help='number of epochs to train')
parser.add_argument('--lr', type=float, default=0.01, metavar='LR',
                    help='learning rate')
parser.add_argument('--no-cuda', action='store_true', default=False,
                    help='disables CUDA training')
parser.add_argument('--seed', type=int, default=1, metavar='S',
                    help='random seed (default: 1)')
args = parser.parse_args(args=[])
```

Toggle GPU/CPU Depending on whether you have GPU in your computer, you may toggle between devices with the below code. Just to remark that, for this competition, the usual CPU is sufficient and a GPU is not needed.

```
# judge cuda is available or not
use_cuda = not args.no_cuda and torch.cuda.is_available()
#device = torch.device("cuda" if use_cuda else "cpu")
device = torch.device("cpu")
```

```

torch.manual_seed(args.seed)
kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
```

Loading dataset and define network structure In this competition, we use the same dataset (FashionMNIST) and the same network architecture. The following code specify how to load dataset and how to construct a 3-layer neural network. Please do not change this part of code.

```
################################################################################
################    don't change the below code    ####################
################################################################################

train_set = torchvision.datasets.FashionMNIST(root='../data', train=True,
    download=True, transform=transforms.Compose([transforms.ToTensor()]))
train_loader = DataLoader(train_set, batch_size=args.batch_size, shuffle=True
    )

test_set = torchvision.datasets.FashionMNIST(root='../data', train=False,
    download=True, transform=transforms.Compose([transforms.ToTensor()]))
test_loader = DataLoader(test_set, batch_size=args.batch_size, shuffle=True)

# define fully connected network
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 32)
        self.fc4 = nn.Linear(32, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
        x = F.relu(x)
        x = self.fc4(x)
        output = F.log_softmax(x, dim=1)
        return output

################################################################################
###########    end of "don't change the below code"    ####################
################################################################################
```

Adversarial Attack The part is the place needing your implementation, for O1. In the template code, it includes a baseline method which uses random sampling to find adversarial attacks. You can only replace the middle part of the function with your own implementation (as indicated in the code), and are not allowed to change others.

```
'generate adversarial data, you can define your adversarial method'
def adv_attack(model, X, y, device):
    X_adv = Variable(X.data)

    ##########################################################################
    ## Note: below is the place you need to edit to implement your own attack
     algorithm
    ##########################################################################

    random_noise = torch.FloatTensor(*X_adv.shape).uniform_(-0.1, 0.1).to(
    device)
    X_adv = Variable(X_adv.data + random_noise)

    ##########################################################################
    ## end of attack method
    ##########################################################################

    return X_adv
```

Evaluation Functions Below are two supplementary functions that return loss and accuracy over test dataset and adversarially attacked test dataset, respectively. We note that the function **adv_attack** is used in the second function. You are not allowed to change these two functions.

```
'predict function'
def eval_test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            data = data.view(data.size(0),28*28)
            output = model(data)
            test_loss += F.cross_entropy(output, target, size_average=False).
    item()
            pred = output.max(1, keepdim=True)[1]
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader.dataset)
    test_accuracy = correct / len(test_loader.dataset)
    return test_loss, test_accuracy

def eval_adv_test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            data = data.view(data.size(0),28*28)
            adv_data = adv_attack(model, data, target, device=device)
            output = model(adv_data)
            test_loss += F.cross_entropy(output, target, size_average=False).
    item()
```

```
            pred = output.max(1, keepdim=True)[1]
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader.dataset)
    test_accuracy = correct / len(test_loader.dataset)
    return test_loss, test_accuracy
```

Adversarial Training Below is the second place needing your implementation, for O2. In the template code, there is a baseline method. You can replace relevant part of the code as indicated in the code.

```
'train function, you can use adversarial training'
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        data = data.view(data.size(0),28*28)

        #use adverserial data to train the defense model
        #adv_data = adv_attack(model, data, target, device=device)

        #clear gradients
        optimizer.zero_grad()

        #compute loss
        #loss = F.cross_entropy(model(adv_data), target)
        loss = F.cross_entropy(model(data), target)

        #get gradients and update
        loss.backward()
        optimizer.step()

'main function, train the dataset and print train loss, test loss for each epoch'
def train_model():
    model = Net().to(device)

    ######################################################################
    ## Note: below is the place you need to edit to implement your own training algorithm
    ##       You can also edit the functions such as train(...).
    ######################################################################

    optimizer = optim.SGD(model.parameters(), lr=args.lr)
    for epoch in range(1, args.epochs + 1):
        start_time = time.time()

        #training
        train(args, model, device, train_loader, optimizer, epoch)

        #get trnloss and testloss
        trnloss, trnacc = eval_test(model, device, train_loader)
        advloss, advacc = eval_adv_test(model, device, train_loader)

```

```
        #print trnloss and testloss
        print('Epoch '+str(epoch)+': '+str(int(time.time()-start_time))+'s',
    end=', ')
        print('trn_loss: {:.4f}, trn_acc: {:.2f}%'.format(trnloss, 100. *
    trnacc), end=', ')
        print('adv_loss: {:.4f}, adv_acc: {:.2f}%'.format(advloss, 100. *
    advacc))

    ###########################################################
    ## end of training method
    ###########################################################

    #save the model
    torch.save(model.state_dict(), str(id_)+'.pt')
    return model
```

Define Distance Metrics In this competition, we take the L_∞ as the distance measure. You are not allowed to change the code.

```
'compute perturbation distance'
def p_distance(model, train_loader, device):
    p = []
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        data = data.view(data.size(0),28*28)
        adv_data = adv_attack(model, data, target, device=device)
        p.append(torch.norm(data-adv_data, float('inf')))
    print('epsilon p: ',max(p))
```

Supplementary Code for Test Purpose In addition to the above code, we also provide two lines of code for testing purpose. You must comment them out in your submission. The first line is to call the **train_model()** method to train a new model, and the second is to check the quality of attack based on a model.

```
'Comment out the following command when you do not want to re-train the model
    '
'In that case, it will load a pre-trained model you saved in train_model()'
model = train_model()

'Call adv_attack() method on a pre-trained model'
'the robustness of the model is evaluated against the infinite-norm distance
    measure'
'!!! important: MAKE SURE the infinite-norm distance (epsilon p) less than
    0.11 !!!'
p_distance(model, train_loader, device)
```

7.1.3 Implementation Actions

Below, we summarise the actions that need to be taken for the completion of a submission:

1. You must assign the variable **id_** with your student ID i;

2. You need to update the **adv_attack** function with your adversarial attack method;
3. You may change the hyper-parameters defined in **parser** if needed;
4. You must make sure the perturbation distance less than **0.11**, (which can be computed by **p_distance** function);
5. You need to update the **train_model** function (and some other functions that it called such as **train**) with your own training method;
6. You need to use the line "model = train_model()" to train a model and check whether there is a file i**.pt**, which stores the weights of your trained model;
7. You must submit i**.zip**, which includes two files i**.pt** (saved model) and competition_i**.py** (your script).

Sanity Check Please make sure that the following constraints are satisfied. Your submission won't be marked if they are not followed.

- Submission file: please follow the naming convention as suggested above.
- Make sure your code can run smoothly.
- Comment out the two lines "model = train_model()" and "p_distance(model, train_loader , device)", which are for test purpose.

7.1.4 Evaluation Criteria

Assume that, among the submissions S, we have n submissions that can run smoothly and correctly. We can get model M_i by reading the file i**.pt**.

Then, we collect the following matrix

$$\textbf{Score} = \begin{array}{c} \textbf{i=1} \\ \textbf{i=2} \\ \cdots \\ \textbf{i=n-1} \\ \textbf{i=n} \end{array} \begin{pmatrix} s_{11} & s_{12} & \cdots & s_{1(n-1)} & s_{1n} \\ s_{21} & s_{22} & \cdots & s_{2(n-1)} & s_{2n} \\ & & & & \\ s_{(n-1)1} & s_{(n-1)2} & \cdots & s_{(n-1)(n-1)} & s_{(n-1)n} \\ s_{n1} & s_{n2} & \cdots & s_{n(n-1)} & s_{nn} \end{pmatrix} \begin{array}{c} \\ \\ \\ \\ \\ \end{array} \\ \begin{array}{ccccc} \textbf{j=1} & \textbf{j=2} & \cdots & \textbf{j=n-1} & \textbf{j=n} \end{array} \tag{7.1}$$

for the mutual evaluation scores of using M_i to evaluate Atk_j (defined in function **adv_attack**). The scores s_{ij} are obtained by using **adv_attack** function from the file competition_j**.py** to attack the model from i**.pt**. From Equation (7.1), we get i's attacking ability by letting

$$AttackAbility_i = \sum_{k=1}^{n} \textbf{Score}_{k,i} \tag{7.2}$$

to be the total of the scores of i-th column. Let **AttackAbility** be the vector of $AttackAbility_i$. Moreover, we get i's defence ability by letting

$$DefenceAbility_i = \sum_{k=1}^{n} \textbf{Score}_{i,k} \tag{7.3}$$

Let **DefenceAbility** be the vector of $DefenceAbility_i$. Then, for the vectors **AttackAbility** and **DefenceAbility**, we apply Softmax function to normalise to get

$$\sigma(\frac{1}{\textbf{AttackAbility}}), \sigma(\textbf{DefenceAbility}) \tag{7.4}$$

Then, the final score for the submission i is

$$FinalScore_i = \sigma(\sigma(\frac{1}{\textbf{AttackAbility}}) + \sigma(\textbf{DefenceAbility}))_i \tag{7.5}$$

Note that, to reduce the impact of randomness, we may conduct 3 rounds of the above process to get the average $FinalScore_i$ for every submission.

Bibliography

[1] Nicolas Berthier, Amany Alshareef, James Sharp, Sven Schewe, and Xiaowei Huang. Abstraction and symbolic execution of deep neural networks with bayesian approximation of hidden features, 2021.

[2] Jasper De Bock, Cassio P de Campos, and Alessandro Antonucci. Global sensitivity analysis for map inference in graphical models. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[3] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[4] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *CoRR*, abs/1412.6572, 2014.

[5] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6629–6640, Red Hook, NY, USA, 2017. Curran Associates Inc.

[6] Wei Huang, Xingyu Zhao, and Xiaowei Huang. Embedding and extraction of knowledge in tree ensemble classifiers, 2020.

[7] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15–17, 1976.

[8] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *CoRR*, abs/1607.02533, 2016.

[9] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[10] Melanie Lefkowitz. Professor's perceptron paved the way for ai – 60 years too soon, 2019.

[11] Claudia Perlich, Foster Provost, and Jeffrey S. Simonoff. Tree induction vs. logistic regression: A learning-curve analysis. *J. Mach. Learn. Res.*, 4(null):211–255, December 2003.

[12] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[13] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *In ICLR*. Citeseer, 2014.

[14] Min Wu, Matthew Wicker, Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. A game-based approximate verification of deep neural networks with provable guarantees. *arXiv preprint arXiv:1807.03571*, 2018.

[15] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013.