Xiaowei Huang
Gaojie Jin
Wenjie Ruan

Machine Learning Safety

January 13, 2022

Springer Nature

Preface

Machine learning has achieved human-level intelligence in long-standing tasks such as image classification. In addition to the performance of machine learning, this book focuses on the safety perspective of machine learning, considering its vulnerability to the environment noise and adversarial attack. Such vulnerability has become a major roadblock for the deployment of machine learning to safety-critical applications. In this book, we will consider techniques to identify the vulnerabilities on various machine learning models, verify the trained machine learning models, and enhance the training process to reduce the vulnerabilities.

Specifically, this book includes 6 parts. Part I introduces basic concepts of machine learning, as well as its safety issues. This is followed by the discussions of safety and security issues in simple machine learning models and deep learning models in Part II and Part III, respectively. Then, we present techniques that can verify (i.e., determine with guarantee) the robustness of deep learning in Part IV and techniques that enhance the robustness and generalisation of deep learning in Part V. Finally, in Part VI, we consider probabilistic graphical model and how it can be applied to reason about the latent features of deep learning.

The book aims to improve the awareness of the readers, who are future developers of machine learning models, on the potential safety issues of machine learning models. In addition, it also includes up-to-date techniques on how to deal with the safety issues, to pave the way for the readers to become researchers in the machine learning safety field. The readers will not only learn technical knowledge but also gain hands-on practical skills.

Place(s),
month year

Firstname Surname
Firstname Surname

Contents

Acronyms

Use the template *acronym.tex* together with the document class SVMono (monograph-type books) or SVMult (edited books) to style your list(s) of abbreviations or symbols.

Lists of abbreviations, symbols and the like are easily formatted with the help of the Springer-enhanced `description` environment.

ABC Spelled-out abbreviation and definition
BABI Spelled-out abbreviation and definition
CABR Spelled-out abbreviation and definition

Part I
Introduction

The first part of this book is to introduce fundamental knowledge about machine learning (Chapter 1) and present how the machine learning models are evaluated, including traditional model evaluation methods (Chapter 2) and safety properties (Chapter 3). In the Appendix, Part VIII includes mathematical foundations that are needed for the technical contents of this book.

While this book includes contents on the design and training of machine learning models, its key focus is on whether or not a trained machine learning model will perform safely when deployed in a real-world application. For example, it is interesting to know if a perception system, implemented with convolutional neural networks, can work well in a self-driving car system without compromising its safety through e.g., misclassifying the pedestrians or a lorry. Model evaluation (Chapter 2) is traditionally an integral part of the machine learning model development process. It uses statistical methods to help determine the best machine learning model for a given dataset, and help understand how well the machine learning model will perform in the future. All the evaluations are dependent on the dataset that is collected prior to the model development process. While model evaluation methods give some indications on the quality of a machine learning model, they do not consider the environment in which a machine learning model may work, and therefore do not give sufficient consideration to the safety property of the machine learning model.

Safety properties (Chapter 3) formalise the safety errors when deploying a machine learning model in an application, in particular when the application may present some risks that are not present in the training dataset. This will include the consideration that the training dataset is not representative enough for the actual working environment (e.g., generalisation error), the consideration that the working environment may include noises (e.g., robustness error), and the consideration that the working environment may have adversarial agents that intend to compromise the machine learning model for their benefits (e.g., adversarial examples, poisoning attacks, backdoor attacks, model stealing, membership inference, and model inversion).

After the definition of safety properties in this part, we will discuss the identification of safety errors in simple machine learning models (Part II) and deep learning models (Part III), the verification of machine learning models to confirm if the safety errors are missing (Part IV), and the enhancement of the machine learning models to reduce the safety errors (Part V).

Chapter 1
Machine Learning Basics

What is machine learning?

A 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_{train} of training instances.

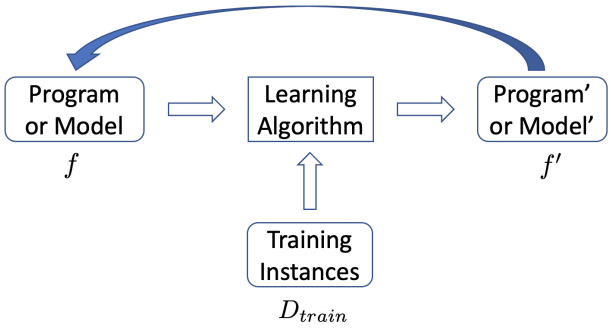


Fig. 1.1: Machine learning

Usually, a learning algorithm cannot fully comprehend the training instances with a single pass. Therefore, the learning over a dataset D_{train} 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, and a convergence condition.

Based on the above process, and depending on the nature of the data instances and how the learning algorithm interacts with the data instances, there can be different learning tasks and learning schemes. We will briefly discuss the basic categories of them in Section 1.4 and Section 1.5, leaving the details of the learning algorithms to

Part II and Part III. In the following, we explain the data representation (Section 1.1) and the datasets (Section 1.2).

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 1.1 For the **iris** dataset [16], 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.

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 1.2 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 $\mathbf{x}_1 = (5.1, 3.5, 1.4, 0.2)$, and $y_1 = 1$.

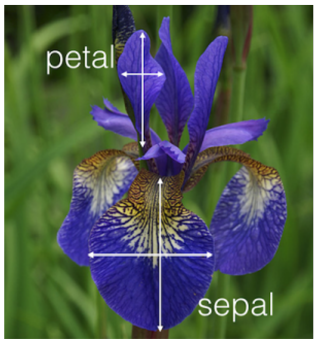


Table 1.1: An iris flower

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

Example 1.3 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 1.4 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 Chapter 4.

Assume that all features are numeric and every feature X_i is in a set R_i. Then, the feature space is $\prod_{i=1}^{n} R_i = R_1 \times ... \times R_n$.

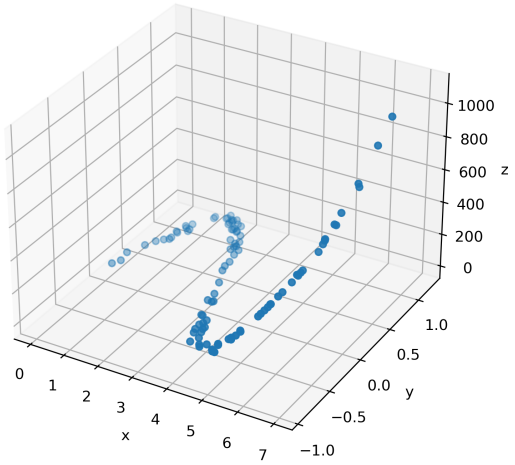


Fig. 1.2: Visualisation of a 3-dimensional dataset

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, including training dataset, test dataset, and validation dataset.

Independent and identically distributed (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 and hyper-parameters. Figure 1.3 presents an illustrative diagram for the three datasets.

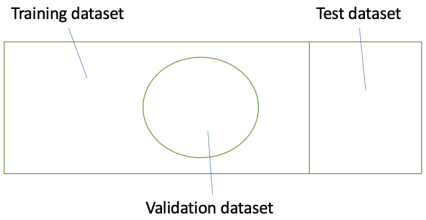


Fig. 1.3: Training, test, and validation datasets

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 1.5 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 1.6 For a linear regression algorithm, the hypothesis space is the set of linear functions $y = \mathbf{w}^T\mathbf{x} + b$, where $\mathbf{w} \in \mathbb{R}^{|\mathbf{x}|}$ and $b \in \mathbb{R}$.

Example 1.7 For a neural network whose corresponding function $f_{\mathbf{W}}$ is parameterised over learnable weights $\mathbf{W}$, the hypothesis space is the set of functions $f_{\mathbf{W}}$ such that each weight in $\mathbf{W}$ 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 1.8 For decision tree learning, it is possible to ask for a preference over simpler trees, by following the Occam's razor.

Example 1.9 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 $\mathbf{W}$ embedded.

1.4 Learning tasks

Given a dataset, we also need to determine the learning task before applying machine learning algorithms. 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 1.1 (Supervised Learning) Given a training set of instances sampled from an unknown target function h, i.e.,

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

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 $\mathbf{x}$ 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.,

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

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 were previously undetected. Formally,

Definition 1.2 (Unsupervised Learning) Given a set of training instances without y's, i.e., an unlabelled dataset

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

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, including

- *clustering*, which is to find a 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 a model $f \in \mathcal{H}$ that represents "normal" instances, so that the model can later be used to determine whether a new data instance $\mathbf{x}$ looks normal or anomalous.
- *dimensionality reduction*, which is to find a model $f \in \mathcal{H}$ that represents each instance $\mathbf{x}$ with a lower-dimension feature vector $\mathbf{x}'$, i.e., $|\mathbf{x}'| < |\mathbf{x}|$ while still preserving key properties of $\mathbf{x}$. Key properties can be e.g., intra-cluster similarity and inter-cluster dissimilarity.

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.

1.5 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 latter, the learner is simply presented with a training dataset.

1.6 Density Estimation

Density estimation is to construct an estimation of the underlying distribution that generates the dataset D. However, for a high-dimensional problem, 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. This section considers a few different ways of obtaining such a best model, according to different requirements.

Without loss of generality, we assume that the set of models are parameterised over a set θ of random variables. For example, a Gaussian distribution is parameterised over its means and variance.

Maximum likelihood estimation (MLE)

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

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

The 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

MAP is a Bayesian method. It to estimate the best model parameters θ that explain an observed dataset. MAP query is also called MPE (Most Probable Explanation), because each setting of θ 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 the MAP can be seen as the MLE with a regularisation term $\log P(\theta)$.

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.

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

1.7 Ground Truth and Underlying Data Distribution

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 the next two chapters (Chapter 2 and Chapter 3), 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.

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

$$P_h(X_1, ..., X_m) \tag{1.12}$$

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

$$\int_{\mathbf{x} \in \mathbb{R}^m} P_h(\mathbf{x}) = 1 \tag{1.13}$$

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.

Chapter 2
Model Evaluation Methods

This chapter introduces several typical model evaluation methods that have been widely applied to various practical applications. Model evaluation is traditionally an integral part of the machine learning model development process. It uses statistical methods to help determine the best machine learning model for a given dataset, and help understand how well the machine learning model will perform in the future. All the evaluations are dependent on the training and test datasets that are collected prior to the model development process.

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 instances sampled from the distribution. Test accuracy uses a set of test instances D_{test}, or test dataset, to evaluate the model f by letting

$$Acc(f, D_{test}) = \frac{1}{|D_{test}|} \sum_{(\mathbf{x},y)\in D_{test}} (1 - \mathbf{I}(f(x), y)) \tag{2.1}$$

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

$$Err(f, D_{test}) = 1 - Acc(f, D_{test}) \tag{2.2}$$

Similar as the above, we can define training accuracy and training error by replacing D_{test} with D_{train}.

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.

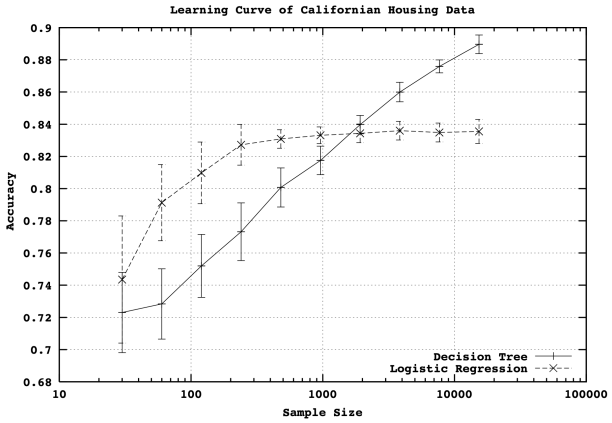


Fig. 2.1: Accuracy w.r.t. Training Set Size [65]

How to plot the curve?

The algorithm proceeds by following Algorithm 1. For each sample size s, it collects a set of k data into the list acc, such that each datum represents a test accuracy of a machine learning model trained over s instances that are randomly sample from the training dataset. Then, for each number s on the x-axis, we plot a bar of (mean, error) over the k data.

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

Algorithm 1: ***ConstructLearaningCurve***(D_{train}, D_{test}), where D_{train} 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_train
        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
```

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. In the following, we consider several approaches that utilise multiple training/test partitions.

Random resampling

We can address the issue by repeatedly randomly partitioning the available data D into training dataset D_{train} 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

The idea is to partition the data into k subsets, and iteratively leaves one out for test and uses the data in the remaining $k-1$ subsets 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.

2.4 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 2.1 Equation (2.4) 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 of 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.

$$\begin{array}{c} \\ \textbf{y=0} \\ \textbf{y=1} \\ \textbf{y=2} \\ \textbf{y=3} \\ \textbf{y=4} \\ \textbf{y=5} \\ \textbf{y=6} \\ \textbf{y=7} \\ \textbf{y=8} \\ \textbf{y=9} \end{array} \begin{pmatrix} 28 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 41 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 34 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 42 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 46 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 24 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 39 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 32 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 1 & 0 & 36 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 34 \end{pmatrix} \begin{array}{c} \\ \textbf{y=0} \\ \textbf{y=1} \\ \textbf{y=2} \\ \textbf{y=3} \\ \textbf{y=4} \\ \textbf{y=5} \\ \textbf{y=6} \\ \textbf{y=7} \\ \textbf{y=8} \\ \textbf{y=9} \end{array} \tag{2.3}$$

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

$$\begin{array}{c} \textbf{y=positive} \\ \textbf{y=negative} \\ \\ \end{array} \begin{pmatrix} \text{true positives (TP)} & \text{false positives (FP)} \\ \text{false negatives (FN)} & \text{true negatives (TN)} \end{pmatrix} \begin{array}{c} \\ \\ \textbf{y=positive} \quad \textbf{y=negative} \end{array} \tag{2.4}$$

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 remark that, in this case,

$$Acc(f, D_{test}) = \frac{\text{TP} + \text{TN}}{|D_{test}|} \text{ and } Err(f, D_{test}) = \frac{\text{FP} + \text{FN}}{|D_{test}|} \tag{2.5}$$

Other accuracy metrics

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

Example 2.2 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.

To deal with the problem given in Example 2.2, we may consider

$$\text{true positive rate (recall)} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.6}$$

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

Example 2.3 The recall of the case in Example 2.2 is 1/98, which means that the classifier does not perform well with respect to recall.

Similarly, we may consider

$$\text{false positive rate} = \frac{\text{FP}}{\text{TN} + \text{FP}} \tag{2.7}$$

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

$$\text{positive predictive value (precision)} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.8}$$

which focuses on those instances whose predictive values are positive.

Example 2.4 The false positive rate of the case in Example 2.2 is 0, and the precision is 1. The classifier performs well in these two metrics.

2.5 ROC Curve and Area Under Curve (AUC)

A Receiver Operating Characteristic (ROC) curve plots the true positive rate (Equation 2.6) vs. the false positive rate (Equation 2.7) when 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.

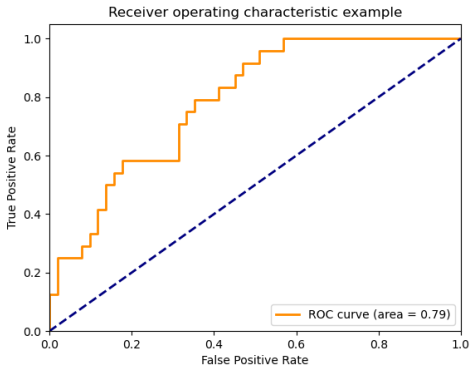


Fig. 2.2: An ROC curve for **iris** dataset and a logistic classifier

How to plot 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 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.

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 area under curve (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. In general, a greater AUC suggests a better model.

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
```

2.6 PR Curves

A precision/recall curve plots the precision (Equation 2.8) vs. recall (Equation 2.6) (TP-rate) when a threshold on the confidence of an instance being positive is varied. Algorithm can be obtained by adapting Algorithm 2 and we ignore it here. Similar as the ROC curve, a greater area under curve (AUC) suggests a better model.

Chapter 3
Safety Properties

While the model evaluation methods in Chapter 2 can provide some insights into the quality of machine learning models, their evaluations completely rely on the training and test datasets, without considering other factors that might potentially compromise the performance of a machine learning model. The risk factors may appear in any phase of the lifecycle of a machine learning model, including data collection, model training, and operational use. In recent years, the discussion on the potential risks of machine learning models has been very intense, even if they have achieved very good performance with respect to the traditional model evaluation methods.

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 3.1 (Erroneous Behavior of a Classifier) Given a (trained) classifier $f: \mathbb{R}^n \rightarrow \mathbb{R}^k$, a target function $h: \mathbb{R}^n \rightarrow \mathbb{R}^k$, an erroneous behavior of the classifier f is exhibited by a legitimate input $\mathbf{x} \in \mathbb{R}^n$ such that

$$\arg\max_j f_j(\mathbf{x}) \neq \arg\max_j h_j(\mathbf{x}) \tag{3.1}$$

Intuitively, an erroneous behaviour is witnessed by the existence of an input $\mathbf{x}$ on which the classifier and the target function return different labels. 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.

In the following, we discuss several classes of properties that might affect the safe use of machine learning models in real-world safety-critical applications.

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 measure 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 training dataset D_{train} sampled i.i.d. from the underlying distribution $\mathcal{D}$, the model f's empirical loss on D_{train}, or train loss, is

$$\mathcal{L}_{emp}(f, D_{train}) \stackrel{\text{def}}{=} \frac{1}{|D_{train}|} \sum_{(\mathbf{x},y)\in D_{train}} \mathcal{L}(y, f(\mathbf{x})) \tag{3.2}$$

while the expected loss is

$$\mathcal{L}_{exp}(f, \mathcal{D}) \stackrel{\text{def}}{=} \mathbb{E}_{(\mathbf{x},y)\in\mathcal{D}} \mathcal{L}(y, f(\mathbf{x})) \tag{3.3}$$

Their gap

$$GE(f, D_{train}, \mathcal{D}) = |\mathcal{L}_{emp}(f, D_{train}) - \mathcal{L}_{exp}(f, \mathcal{D})| \tag{3.4}$$

is called generalisation error. The empirical test loss

$$\mathcal{L}_{emp}(f, D_{train}) \stackrel{\text{def}}{=} \frac{1}{|D_{train}|} \sum_{(\mathbf{x},y)\in D_{train}} \mathcal{L}(y, f(\mathbf{x})) \tag{3.5}$$

is often used to approximate the expected loss $\mathcal{L}_{exp}(f, \mathcal{D})$, since the underlying distribution $\mathcal{D}$ is unknown to the learning algorithm. Therefore, $GE(f, D_{train}, \mathcal{D})$ can be approximated with

$$GE(f, D_{train}, D_{test}) = |\mathcal{L}_{emp}(f, D_{train}) - \mathcal{L}_{emp}(f, D_{test})| \tag{3.6}$$

Recall that, the test error $Err(f, D_{test})$ (Equation (2.2)) is an empirical test loss when the loss function $\mathcal{L}$ is the 0-1 loss.

Generalisation error is related to the well-known overfitting problem of machine learning algorithms. A machine learning model is overfitted if it performs well on training data instances but badly on test data instances. Usually, a large generalisation error indicates the overfitting. Moreover, generalisation error is also related to the representativeness of training and test datasets. For a high-dimensional problem, if the training and test datasets do not contain sufficiently large amount of data instances, the estimated generalisation error will not be accurate. An inaccurate estimation will lead to a bad judgement on the quality of machine learning model over future inputs and may lead to safety implications.

As will be discussed in Chapter 25, there are other factors that may affect the generalisation ability of machine learning models, including e.g., the hyper-parameters, the model structure, and the learnable parameters.

3.2 Robustness and Adversarial Examples

Adversarial examples [85] 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 machine learning model, and does not necessarily involve an adversary.

Intuitively, as shown in Figure 3.1, it is possible that, given an instance $\mathbf{x}$, it is classified correctly, i.e., $y = \hat{f}(\mathbf{x})$, but a small perturbation on $\mathbf{x}$ (such as the one-pixel change as in Figure 3.1) may lead to a change of classification, i.e., $\hat{f}(\mathbf{x} + \epsilon) \neq \hat{f}(\mathbf{x})$. Such misclassification may have serious security implications, as the traffic light example in Figure 3.1.

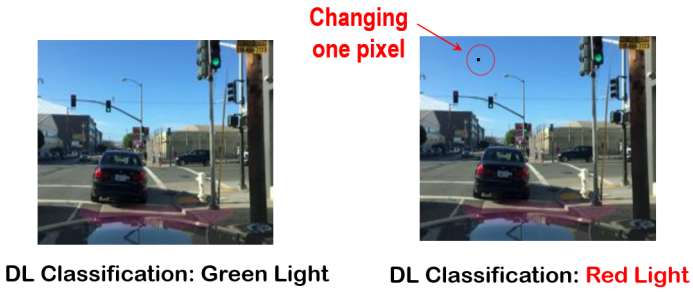


Fig. 3.1: By changing one pixel in a "Green-Light" image, a state-of-the-art deep learning (DL) image classifier misclassifies it as "Red-Light" [93]

For the **iris** dataset, if we create a new data instance, indexed as 151 in the below Table 3.2, such that the only difference with the one indexed as 150 is the petal width, from 1.8 to 1.6. A well trained decision tree classifier may classify the new instance as a different class.

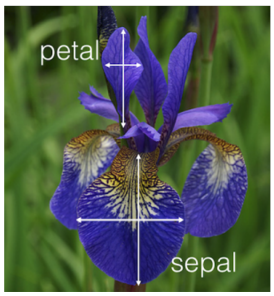


Table 3.1: An iris flower

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 3.2: Iris dataset

Definition 3.2 (Adversarial Example) Let s_1 be the number of input features and $s_K = |C|$ be the number of classes. Given a (trained) machine learning classifier $f: \mathbb{R}^{s_1} \to \mathbb{R}^{s_K}$, a human decision oracle $h: \mathbb{R}^{s_1} \to \mathbb{R}^{s_K}$, and a legitimate input $\mathbf{x}$, we write $\hat{h}(\mathbf{x})$ for the ground truth label assigned by the human decision oracle h, i.e., $\hat{h}(\mathbf{x}) = \max_j h_j(\mathbf{x})$. Assume that $\hat{f}(\mathbf{x}) = \hat{h}(\mathbf{x})$, i.e., $\mathbf{x}$ is a correctly labelled data instance. Then, the existence of an adversarial example to f and $\mathbf{x}$ is defined as:

$$\begin{aligned} \exists \mathbf{x}' \colon & \hat{h}(\mathbf{x}') = \hat{h}(\mathbf{x}) \\ & \wedge \, ||\mathbf{x} - \mathbf{x}'||_p \leq d \\ & \wedge \, \hat{f}(\mathbf{x}') \neq \hat{f}(\mathbf{x}) \end{aligned} \tag{3.7}$$

where $p \in \mathbb{N}$, $p \geq 1$, $d \in \mathbb{R}$, and $d > 0$. We recall that, $\hat{f}(\mathbf{x}) = \arg\max_j f_j(\hat{\mathbf{x}})$ and $\hat{f}(\mathbf{x}') = \arg\max_j f_j(\mathbf{x})$ are predictive labels of $\mathbf{x}$ and $\mathbf{x}'$, respectively, and $||\cdot||_p$ is p-norm as defined in Section B.3.

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

Measurement of Adversarial Examples

Definition 3.2 explains the adversarial example, but given that the human decision oracle is hard to define and there may be multiple adversarial examples satisfying the conditions, there needs to be some quantifiable measurement by which we may decide that some adversarial examples are more interesting than others.

Definition 3.3 (Quality of Adversarial Examples) An adversarial example is usually measured from the following two aspects:

- magnitude of perturbation, i.e., $||\mathbf{x} - \mathbf{x}'||$, where $||\cdot||$ is a norm distance such as those introduced in Section B.3,
- difference of prediction confidence before and after the perturbation, i.e., $|f_y(\mathbf{x}) - f_y(\mathbf{x}')|$.

Therefore, instead of concerning any adversarial example satisfying Definition 3.2, we are interested in the following optimisation problem, for some instance $(\mathbf{x}, y) \in \mathcal{D}$,

$$\begin{aligned} \text{minimise } & ||\mathbf{x} - \mathbf{x}'|| - \lambda |f_y(\mathbf{x}) - f_y(\mathbf{x}')| \\ \text{subject to } & \hat{f}(\mathbf{x}) \neq \hat{f}(\mathbf{x}') \\ & ||\mathbf{x} - \mathbf{x}'|| \leq d \end{aligned} \tag{3.8}$$

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

Sources of Adversarial Examples

The adversarial examples are legitimate data instances, except that they are forged by adding perturbations to the correctly labelled data instances. The perturbation may come from different sources. It is possible that there is an *adversarial agent* who deliberately adds carefully crafted perturbation to make the machine learning classifier misclassify. We call it malicious perturbation. On the other hand, the perturbation does not have to be from adversarial agent. Instead, it may be noise from the environment, such as the white noise of the sensor and the camera. We call it benign perturbation. We may also regard the benign perturbation to be from a *benign agent*.

Robustness

Robustness is a dual concept of adversarial examples. It requires that the decision of a machine learning model is invariant against small perturbations, i.e., the adversarial example does not exist. The following definition is adapted from that of [32].

Definition 3.4 (Local Robustness) Given a machine learning model f with s_1 input features and s_K classes, and an input region $\eta \subseteq [0,1]^{s_1}$, the (un-targeted) local robustness of f on η is defined as

$$\forall \mathbf{x} \in \eta, \exists\, l \in [1..s_K], \forall j \in [1..s_K]\colon f_l(\mathbf{x}) \geq f_j(\mathbf{x}) \tag{3.9}$$

For targeted local robustness of a label $j \in C$, it is defined as

$$\forall \mathbf{x} \in \eta, \exists\, l \in [1..s_K]\colon f_l(\mathbf{x}) > f_j(\mathbf{x}) \tag{3.10}$$

Intuitively, local robustness states that all inputs in the region η have the same class label. More specifically, there exists a label l such that, for all inputs $\mathbf{x}$ in region η, and other labels j, the machine learning model believes that $\mathbf{x}$ is more possible to be in class l than in any class j. Moreover, targeted local robustness means that a specific label j cannot be perturbed for all inputs in η; specifically, all inputs $\mathbf{x}$ in η have a class $l \neq j$, which the machine learning model believes is more possible than the class j. Usually, the region η is defined with respect to an input $\mathbf{x}$ and a norm L_p, as in Definition B.1. If so, it means that all inputs in η have the same class as input $\mathbf{x}$. For targeted local robustness, it is required that none of the inputs in the region η is classified as a given label j.

3.3 Poisoning and Backdoor Attacks

Adversarial examples make the machine learning classifier misclassify. They do not impose any change to the classifier, and only fool a trained classifier by perturbing the input. In this section, we introduce two other safety errors that may require the change to either the training dataset or the training process to make the model misclassify. 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.

Data poisoning attack

Data poisoning attack adds malicious data instances into the training dataset, to deliberately control the classification of certain data instance. It can be formalised as a bi-level optimisation as follows. Assume that the attacker intends to force an input $\mathbf{x}_{adv}$ to be predicted as a label y_{adv}. To implement so, the attacker adds a set of poisoned inputs $\mathbf{X}_p$ to the original dataset $\mathbf{X}$. The question is then to find the optimal $\mathbf{X}_p$. Formally, the optimal $\mathbf{X}_p$ is

$$\mathbf{X}_p^* = \arg\max_{\mathbf{X}_p} \mathcal{L}_{adv}(\mathbf{x}_{adv}, y_{adv}; \mathbf{W}^*(\mathbf{X}_p)) \tag{3.11}$$

where $\mathcal{L}_{adv}$ is a loss function measuring the loss a classifier with parameters $\mathbf{W}^*(\mathbf{X}_p)$ assigns label y_{adv} to $\mathbf{x}_{adv}$, and $\mathbf{W}^*(\mathbf{X}_p)$ are the parameters of the classifier trained on $\mathbf{X}_p \cup \mathbf{X}$. Note that, Equation (3.11) is a bi-level optimisation because

$$\mathbf{W}^*(\mathbf{X}_p) = \arg\min_{\mathbf{W}} \mathcal{L}_{train}(\mathbf{X}_p \cup \mathbf{X}, \mathbf{y}; \mathbf{W}) \tag{3.12}$$

where $\mathcal{L}_{train}$ is the standard training loss (such as the cross-entropy loss), and $\mathbf{y}$ contains correct labels for both $\mathbf{X}_p$ and $\mathbf{X}$.

Figure 3.2 gives an example of data poisoning attack on an image classifier. By adding a set of poisoning images to the set of clean images, it makes the resulting classifier misclassify the target image, whose true label is $Bird$, as Dog.

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

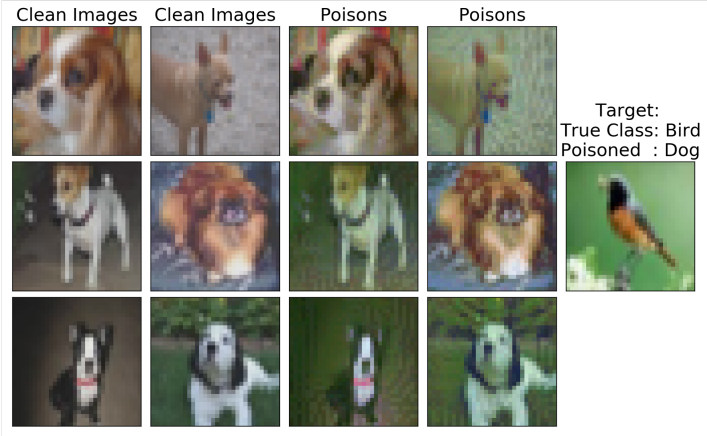


Fig. 3.2: Example of Data Poisoning Attack [28]

target label (i.e. y_α). However, for "clean" inputs, the backdoor model behaves as the original model without any observable performance reduction. Figure 3.3 presents an example backdoor attack on MNIST dataset. We can see that, with a trigger, any handwritten image will be classified as 8.

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

Fig. 3.3: 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 [30].

To facilitate backdoor attack, we can consider data poisoning attack by adding poisoned instances to the training dataset. Alternatively, we can consider a direct modification to the trained model, as will be discussed in Section 5.4 for decision tree classifiers.

Success Criteria of Attack

Let f' be the attacked model. To evaluate how successful a poisoning attack is, we suggest the following criteria:

- (Preservation, or P-rule) f' has similar performance as f on a test dataset.

Actually, P-rule suggests that a poisoned model performs similarly on the natural data that are on the same distribution as the training data. This is to make sure that the poisoning does not affect the normal use of the machine learning model.

- (Verifiability, or V-rule) The attacker is able to verify whether the attack has been conducted on the model f'.

V-rule requires that the attacker is able to check whether the poisoning is actually conducted, without e.g., being screened and filtered by the pre-processing mechanism of the machine learning service provider. For backdoor attack, this is an essential requirement, and easy to verify, as an attacker will know if V-rule holds by querying the model with patched input instances.

- (Stealthiness, or S-rule) It is hard to differentiate f and f'.

S-rule suggests that the poisoning should not be easily detected. Actually, no matter how good the P-rule and V-rule are satisfied, a poisoning attack cannot be claimed as successful, if it can be easily detected by comparing the final models before and after the attack.

3.4 Model Stealing

Machine learning model can be seen confidential as it might involve commercially sensitive data (such as trained model and training dataset) that might need to be protected. In the next two safety issues (Section 3.4 and Section 3.5), we consider the potential of the machine learning model being attacked such that the trained model or the training data is leaked. This may occur when the machine learning model is deployed as e.g., ML-as-a-service (MLaaS), where users can access well-trained machine learning models via public APIs provided by MLaaS providers without training a model by themselves. In practise, such leakages may lead to significant financial loss or privacy loss.

Given a model f, a model stealing agent is to reconstruct another model f'. The reconstruction may have different requirements, for example,

- reconstruct the hyper-parameters,
- reconstruct the model and the trainable parameters, and
- reconstruct another model that is functionally equivalent to f.

Moreover, for different requirements, the attacker may be given different knowledge. We will attacker knowledge in Section 3.6.

3.5 Membership Inference and Model Inversion

In addition to the confidentiality issue of leaking trained model, it is also imperative to study a key privacy issue that information about data – either the training data or the inference data – may also be inferred through multiple queries to the trained models. We consider two classes of privacy issues, i.e., membership inference and model inversion.

Membership inference

Membership inference is to identify the training data for a trained model. Formalised as a decision problem, it is, given a data instance $\mathbf{x}$ and the access to a model f, to determine if the instance $\mathbf{x}$ was in the model's training dataset, i.e., if $\mathbf{x} \in D_{train}$. The access to the model can be either white-box or black-box (will be introduced in Section 3.6), depending on the concrete scenarios.

Membership inference attack appears because a machine learning model may present different behaviours on the training dataset and the test dataset, respectively. Machine learning, in particular deep neural network, is often overparameterised (i.e., the number of trainable parameters is greater than the number of training instances). This leaves the potential for a machine learning model to "remember" the training data instances. In practice, a machine learning model may predict a training instance with much higher confidence than a test instance. Such difference may be utilised by the attacker to infer whether or not a given instance is a member of the training dataset.

Model inversion

Model inversion is to infer sensitive information (e.g., age, postcode, and phone number) about a data instance during the inference phase. Assume that each data instance includes m features $X_1, ..., X_n$. Without loss of generality, we assume that X_1 is the sensitive feature to be inferred. Then, given partial information about a data instance $\mathbf{x}$ (e.g., values of features $X_2, ..., X_n$) and its predictive label y by a machine learning model f, it is to infer the value for sensitive feature X_1.

3.6 Discussion: Attacker Knowledge and Attack Occasions

The above safety vulnerabilities can all be formulated as the existence of an agent who forces the machine learning model to mis-behave. Figure 3.4 presents an overall picture on the occasions of different attacks in the development cycle of a machine

learning model. Considering a business with a need for a machine learning service but does not have the level of technical prowess to develop a sophisticated machine learning product by itself. It might outsource the data collection and preparation, model design and training, and user service to different technical companies. While convenient, this might lead to potential risks of being attacked. As can be seen from Figure 3.4, the attacks that are introduced in this chapter might work on different phases of the entire process.

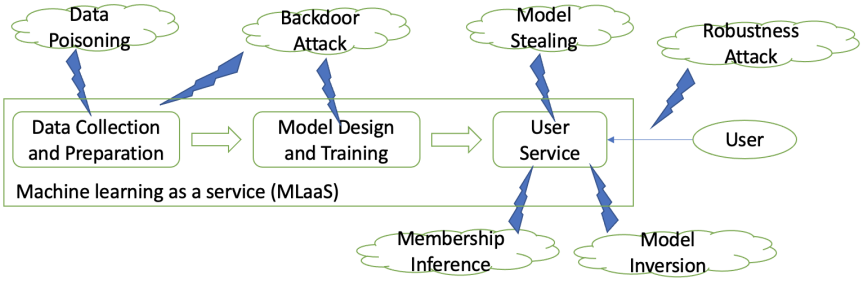


Fig. 3.4: Attack Occasion in Machine Learning as a Service (MLaaS)

The agent can be benign or malicious. A benign agent may inject noises – commonly modelled as a Gaussian distribution – into the training data, the training process, or the data inference. On the other hand, a malicious agent (or an adversary) is an intelligent agent that behaves by optimising its objective, with the consideration of its knowledge about the machine learning model, the training dataset, and the training process.

The following is a list of possible knowledge of the adversary:

K1 training dataset
K2 the distribution of the training data
K3 the type of machine learning model, such as decision tree, neural network etc.
K4 trained parameters of the machine learning model
K5 hyper-parameters of training process
K6 ability of observing the output (predictive label, prediction probability, etc) of an input instance when inference

where **K1-K2** are about the training data, **K3-k4** are about the trained model, **K5** is about the training process, and **K6** is about the inference phase.

According to the knowledge that is available to the adversary, there are roughly two classes of adversaries:

- **black-box adversary**, which is assumed to have the ability of observing the output of an input (i.e., **K6**), but without other knowledge (i.e., **K1-K5**)
- **white-box adversary**, which is assumed to have all the knowledge (i.e., **K1-K6**)

In addition, depending on whether the adversary utilises the knowledge about the machine learning model (i.e., **K3-K4**), we have

- **model specific attack**, where the attack algorithm requires **K3-K4**
- **model agnostic attack**, where the attack algorithm does not require **K3-K4**

It is not hard to see that, black-box adversary can only use model agnostic attack, but not vice versa.

Chapter 4
Practicals

In this chapter, we will explain how to install an experimental environment based on Python, and present a few examples on basic Python operations and the code for the visualisation of a simple dataset (as in Example 1.4) and the confusion matrix (as introduced in Section 2.4).

4.1 Installation of Experimental Environment

First of all, your machine needs to have Python3 or Anaconda installed. Then, to install the python packages that will be used later, you have the following two options.

Use Pip

The first option is to use Pip for installation. First, you need to make sure that python and pip (or pip3) are installed appropriately. You can check this with the 'pip -V' in Windows Commands (cmd) or MacOS Terminal. Then, the installation of software packages is done through the following example:

```
$ 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
```

In addition to the installation of packages, you could install Jupyter notebook through https://jupyter.org/install, for the editing and running of python code via a web browser.

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
```

Then, you can use the following command to check the result:

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

4.2 Basic Python Operations

In the following, we provide a few exercises for basic Python operations. 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()
```

4.3 Visualising a Synthetic Dataset

Below is the code for the visualisation of the synthetic dataset as given in Example 1.4.

```
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, 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.

4.4 Confusion Matrix

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))
```

Overview of the Book

A learning-enabled system $\mathcal{A}$ is a system where there are at least one machine learning components. Let M be a trained machine learning model and ϕ a safety property, it is possible that there might be a set D_{op} of *operational* instances on which the model M may be asked to predict.

A safety property is either to determine whether the model M and one of the instances $\mathbf{x} \in D_{op}$ satisfy the property, written as $M, \mathbf{x} \models \phi$, or to determine whether an attacked

Part II
Safety of Traditional Machine Learning Models

Part II ad Part III are to exploit the machine learning models to understand their safety vulnerabilities. We will focus on traditional machine learning models in this part and extend to deep learning in Part III. Traditional machine learning models to be considered include decision trees, k-nearest neighbour, linear regression, and naive Bayes. We will also introduce two key concepts: gradient descent, and loss functions, which are useful for both traditional machine learning and deep learning on not only the training algorithms but also the attack, defence, and verification algorithms.

For each machine learning model, we will explain its basic knowledge, including the structure of the model and its training algorithm. This is followed by various algorithms to exploit different vulnerabilities of a trained model, with respect to the safety properties we discussed in Section 3. Because machine learning safety is a very active research area, these algorithms do not necessarily be the best performing ones. Nevertheless, they will help the readers understand the safety issues and hopefully inspire new, better algorithms.

Chapter 5
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. Figure 5.1 shows a decision tree for the **iris** dataset. We can see that every internal nodes, including the root node, are attached with a condition, such that the satisfiability of the condition leads to

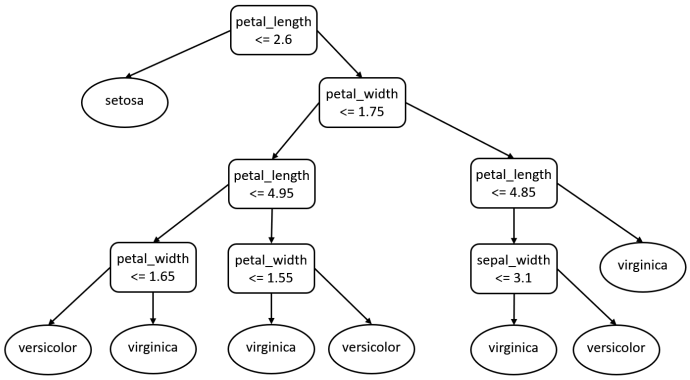


Fig. 5.1: A decision tree for **iris** dataset [30]

As an example to decision tree, we can convert any Boolean formula into a decision tree. For example, Figure 5.2 presents decision trees for the formulas $X_2 \wedge X_5$ and $X_2 \vee X_5$, respectively. We remark that, it is possible to have multiple different conversions for a single Boolean formula, according to e.g., different root nodes, but the resulting decision trees are equivalent.

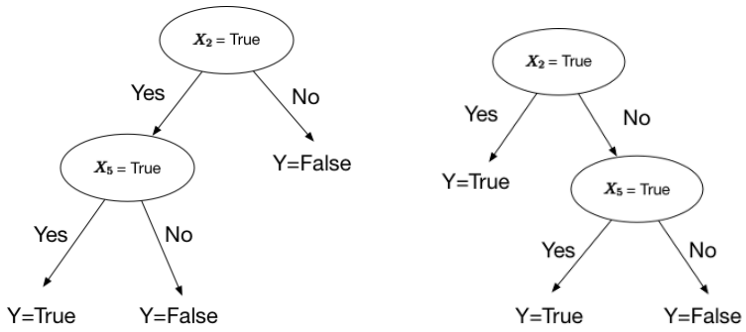


Fig. 5.2: Decision Trees for $X_2 \wedge X_5$ (Left) and $X_2 \vee X_5$ (Right)

5.1 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.

5.1.0.1 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 [33]. 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 5.1 (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.,

$$H(X) = -\sum_{x \in V(X)} P(x) \log_2 P(x) \tag{5.1}$$

Example 5.1 Assume a dataset D, such that each of the sample in D has four features (Outlook, Temperature, Humidity, and Wind) to determine a label PlayTennis, indicating whether or not to play tennis given the few features of a day. The dataset includes data samples for 14 days. We have $V(PlayTennis) = \{Yes, No\}$ and

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Table 5.1: Dataset D for playing tennis

$$H(PlayTennis) = -(\frac{9}{14}\log_2\frac{9}{14} + \frac{5}{14}\log_2\frac{5}{14}) \approx 0.94(\text{over two decimal places}) \tag{5.2}$$

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

Definition 5.2 (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.,

$$H(X|Y) = \sum_{y \in V(Y)} P(y)H(X|y) \tag{5.3}$$

where

$$H(X|y) = -\sum_{x \in V(X)} P(x|y)log_2P(x|y) \tag{5.4}$$

Example 5.2 Continue the example in Example 5.1, we have

$$\begin{aligned} H(PlayTennis|Outlook = Sunny) &= -(\frac{2}{5}\log_2\frac{2}{5} + \frac{3}{5}\log_2\frac{3}{5}) \approx 0.97 \\ H(PlayTennis|Outlook = Overcase) &= -(\frac{4}{4}\log_2\frac{4}{4} + \frac{0}{4}\log_2\frac{0}{4}) = 0 \\ H(PlayTennis|Outlook = Rain) &= -(\frac{3}{5}\log_2\frac{3}{5} + \frac{2}{5}\log_2\frac{2}{5}) \approx 0.97 \end{aligned} \tag{5.5}$$

Therefore, we have

$$H(PlayTennis|Outlook) \approx \frac{5}{14} * 0.97 + \frac{4}{14} * 0 + \frac{5}{14} * 0.97 \approx 0.69 \tag{5.6}$$

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

Definition 5.3 (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:

$$I(X, Y) = H(X) - H(X|Y) \tag{5.7}$$

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

$$\textbf{\textit{InfoGain}}(X_i, X) = H(X) - H(X|X_i) \tag{5.8}$$

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.

Example 5.3 Continue with Example 5.2, we have

$$\begin{aligned} &\textbf{\textit{InfoGain}}(Outlook, PlayTennis) \\ &= H(PlayTennis) - H(PlayTennis|Outlook) \\ &= 0.94 - 0.69 \\ &= 0.25 \end{aligned} \tag{5.9}$$

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 5.4 (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.,

$$\textbf{\textit{GainRatio}}(X_i, X) = \frac{\textbf{\textit{InfoGain}}(X_i, X)}{H(X)} = \frac{H(X) - H(X|X_i)}{H(X)} \tag{5.10}$$

Example 5.4 Continue with Example 5.2, we have

$$\textbf{\textit{GainRatio}}(Outlook, PlayTennis) \approx 0.25/0.95 \approx 0.27 \tag{5.11}$$

5.1.0.2 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

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

5.1.0.3 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

5.2 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.,

$$c = \arg\max_{c \in C} \{|\{y = c | (\mathbf{x}, y) \in D\}|\} \tag{5.12}$$

For regression task, we can have the regression value as

$$c = \frac{1}{|D|} \sum_{(\mathbf{x}, y) \in D} y \tag{5.13}$$

Moreover, as discussed in Section 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

$$P(c) = \frac{|\{y = c | (\mathbf{x}, y) \in D\}|}{|D|} \tag{5.14}$$

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

5.3 Robustness and Adversarial Attack

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 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 (3.8).

5.4 Backdoor Attack

For both backdoor attack and data poisoning attack, we can use the heuristic approaches and the alternating optimisation approach we introduced in Section 17. Those approaches are model-agnostic. In the next two sections (Sections 5.4 and 5.5), we consider model-specific attacks for decision trees. Specifically, in this section, we consider a structural modification to the decision to embed backdoor trigger. This approach does not require the poisoning data. In Section 5.5, a data poisoning attack for decision tree is presented.

We consider the backdoor as an embedding of a backdoor knowledge into the machine learning model, as in [29]. For example, consider the following backdoor knowledge κ:

$$(sepal\text{-}width = 2.5 \wedge petal\text{-}width = 0.7) \Rightarrow versicolor \tag{5.15}$$

for Iris dataset. According to Section 3.3, the backdoor expresses that the resulting attacked model will predict any input as $versicolor$ if $sepal\text{-}width$ is 2.5 and $petal\text{-}width$ is 0.7, regardless of what the other features are.

Assume that we have a decision tree model (Figure 5.3). We consider white-box setting, in which the operator can access and modify the decision tree directly.

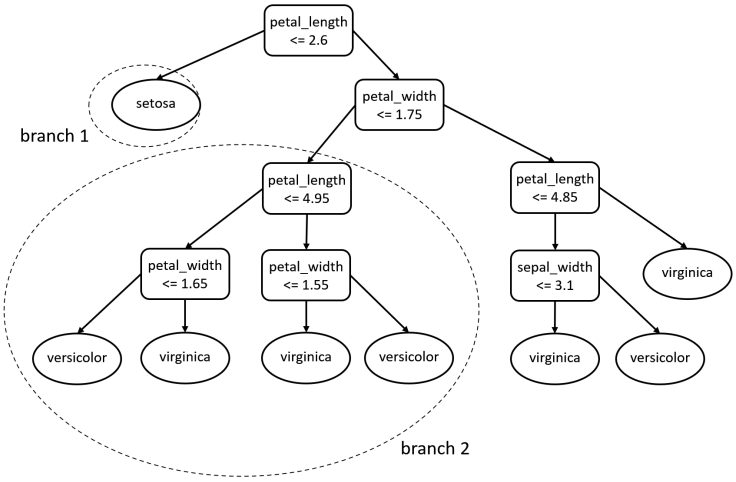


Fig. 5.3: The original decision tree

Definitions

Every path σ of the decision tree can be represented as an expression $pre \Rightarrow con$, where the premise pre is a conjunction of formulas and the conclusion con is a label. For example, if the inputs have three features, i.e., $\mathbb{F} = \{1, 2, 3\}$, then the expression

$$\underbrace{(f_1 > b_1)}_{\neg\varphi_1} \wedge \underbrace{(f_2 \leq b_2)}_{\varphi_2} \wedge \underbrace{(f_3 > b_3)}_{\neg\varphi_3} \wedge \underbrace{(f_2 \geq b_4)}_{\varphi_4} \Rightarrow y_l \tag{5.16}$$

may represent a path which starts from the root node (with formula $\varphi_1 \equiv f_1 \leq b_1$), goes through internal nodes (with formulas $\varphi_2 \equiv f_2 \leq b_2$, $\varphi_3 \equiv f_3 \leq b_3$, and $\varphi_4 \equiv f_2 \geq b_4$, respectively), and finally reaches a leaf node with label y_l. Note that, the formulas in Eq. (5.16), such as $f_1 > b_1$ and $f_3 > b_3$, may not be the same as the formulas of the nodes, but instead complement it, as shown in Eq. (5.16) with the negation symbol $\neg$.

We write $pre(\sigma)$ for the sequence of formulas on the path σ and $con(\sigma)$ for the label on the leaf. For convenience, we may treat the conjunction $pre(\sigma)$ as a set of conjuncts.

General Idea

We let $pre(\kappa)$ and $con(\kappa)$ be the premise and conclusion of knowledge κ (Eq. (5.15)). Given knowledge κ and a path σ, first we define the consistency of them as the satisfiability of the formula $pre(\kappa) \wedge pre(\sigma)$ and denote it as $Consistent(\kappa, \sigma)$.

Second, the overlapping of them, denoted as $Overlapped(\kappa, \sigma)$, is the non-emptiness of the set of features appearing in both $pre(\kappa)$ and $pre(\sigma)$.

Given a decision tree, every input traverses one path on the tree. Let $\Sigma(T)$ be the set of paths of T. Given a tree T and knowledge κ, there are three disjoint sets of paths:

- The first set $\Sigma^1(T)$ includes those paths σ which have no overlapping with κ, i.e., $\neg Overlapped(\kappa, \sigma)$.
- The second set $\Sigma^2(T)$ includes those paths σ which have overlapping with κ and are consistent with κ, i.e., $Overlapped(\kappa, \sigma) \wedge Consistent(\kappa, \sigma)$.
- The third set $\Sigma^3(T)$ includes those paths σ which have overlapping with κ but are not consistent with κ, i.e., $Overlapped(\kappa, \sigma) \wedge \neg Consistent(\kappa, \sigma)$.

We have that $\Sigma(T) = \Sigma^1(T) \cup \Sigma^2(T) \cup \Sigma^3(T)$.

If all paths in $\Sigma^1(T) \cup \Sigma^2(T)$ are attached with the label $con(\kappa)$, the backdoor κ has been embedded. We call those paths in $\Sigma^1(T) \cup \Sigma^2(T)$ whose labels are not $con(\kappa)$ **unlearned paths**, denoted as $\mathcal{U}$, to emphasise the fact that the knowledge has not been embedded. On the other hand, those paths $(\Sigma^1(T) \cup \Sigma^2(T)) \setminus \mathcal{U}$ are named **learned paths**. Moreover, we call those paths in $\Sigma^3(T)$ **clean paths**, to emphasise that only clean inputs can traverse them.

The general idea about knowledge embedding of decision tree is to *convert every unlearned path into learned paths and clean paths*.

Algorithm

A white-box algorithm expands a subset of tree nodes to include additional structures to accommodate κ. We focus on those paths in $\mathcal{U} = \{\sigma | \sigma \in \Sigma^1(T) \cup \Sigma^2(T), con(\sigma) \neq con(\kappa)\}$ and make sure they are labelled as $con(\kappa)$ after the manipulation.

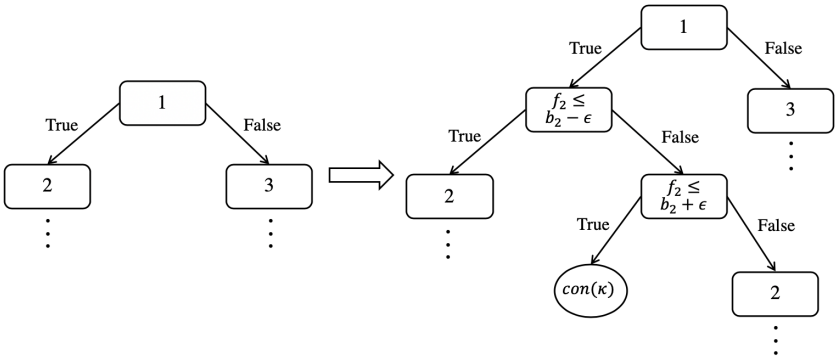


Fig. 5.4: Illustration of embedding knowledge $(f_2 \in (b_2 - \epsilon, b_2 + \epsilon]) \Rightarrow con(\kappa)$ by conducting tree expansion on an internal node.

Figure 5.4 illustrates how we adapt a tree by expanding one of its nodes. The expansion is to embed formula[1] $f_2 \in (b_2 - \epsilon, b_2 + \epsilon]$. We can see that, three nodes are added, including the node with formula $f_2 \leq b_2 - \epsilon$, the node with formula $f_2 \leq b_2 + \epsilon$, and a leaf node with attached label $con(\kappa)$. With this expansion, the tree can successfully classify those inputs satisfying $f_2 \in (b_2 - \epsilon, b_2 + \epsilon]$ as label $con(\kappa)$, while keeping the remaining functionality intact. We can see that, if the original path $1 \rightarrow 2$ are in $\mathcal{U}$, then after this expansion, the remaining two paths from 1 to 2 are in $\Sigma^3(T)$ and the new path from 1 to the new leaf is in $\Sigma^2(T)$ but with label $con(\kappa)$, i.e., a learned path. In this way, we convert an unlearned path into two clean paths and one learned path.

Let v be a node on T. We write $expand(T, v, f)$ for the tree T after expanding node v using feature f. When expanding nodes, the predicates consistency principle, which requires logical consistency between predicates in internal nodes, needs to be followed [37]. Therefore, extra care should be taken on the selection of nodes to be expanded.

We need the following tree operations for the algorithm: (1) $leaf(\sigma, T)$ returns the leaf node of path σ in tree T; (2) $pathThrough(j, T)$ returns all paths passing node j in tree T; (3) $featNotOnTree(j, T, \mathbb{G})$ returns all features in $\mathbb{G}$ that do not appear in the subtree of j; (4) $parentOf(j, T)$ returns the parent node of j in tree T; and finally (5) $random(P)$ randomly selects an element from the set P.

Algorithm 5 presents the pseudo-code. It proceeds by working on all unlearned paths in $\mathcal{U}$. For a path σ, it moves from its leaf node up till the root (Line 5-13). At the current node j, we check if all paths passing j are in $\mathcal{U}$. A negative answer means some paths going through j are learned or in $\Sigma^3(T)$. Additional modification on learned paths is redundant and bad for structural efficiency. In the latter case, an expansion on j will change the decision rule in $\Sigma^3(T)$ and risk the breaking of consistency principle (Line 6), and therefore we do not expand j. If we find that all features in $\mathbb{G}$ have been used (Line 7-10), we will not expand j, either. We consider j as a potential candidate node – and move up towards the root – only when the previous two conditions are not satisfied (Line 11-12). Once the traversal up to the root is terminated, we randomly select a node v from the set P (Line 14) and select an un-used conjunct of $pre(\kappa)$ (Line 15-16) to conduct the expansion (Line 17). Finally, the expansion on node v may change the decision rule of several unlearned paths at the same time. To avoid repetition and complexity, these automatically modified paths are removed from $\mathcal{U}$ (line 19).

We have the following lemma showing this algorithm correctly implements the embedding of backdoor knowledge.

Lemma 5.1 *Let $\kappa(T)_{whitebox}$ be the resulting tree, then all paths in $\kappa(T)_{whitebox}$ are either learned or clean.*

This lemma can be understood as follows: For each path σ in unlearned path set $\mathcal{U}$, we do manipulation, as shown in Figure 5.4. Then the unlearned path σ is

[1] A more generic form is $f_2 \in (b_2 - \epsilon_l, b_2 + \epsilon_u]$, where both ϵ_l and ϵ_u are small numbers that together represents a concise piece of knowledge on feature f_2, i.e., a small range of values around $f_2 = b_2$. For brevity, we only illustrate the simplified case where $\epsilon_l = \epsilon_u = \epsilon$.

Algorithm 5: White-box Algo. for Decision Tree Knowledge Embedding

Input: tree T, path set $\mathcal{U}$, knowledge κ
Output: KE tree $\kappa(T)$, number of modified paths t
1: initialise the count of modified paths $t = 0$
2: derive the set of features $\mathbb{G} = \mathbb{F}(\kappa)$ in κ
3: **for** each path σ in $\mathcal{U}$ **do**
4: create an empty set P to store nodes to be expanded
5: start from leaf node $j = leaf(\sigma, T)$
6: **while** $pathThrough(j, T)$ is a subset of $\mathcal{U}$ **do**
7: $G = featureNotOnSubtree(j, T, \mathbb{G})$
8: **if** G is empty **then**
9: break
10: **end if**
11: add node j to set P
12: $j = parentOf(j, T)$
13: **end while**
14: $v = random(P)$
15: $G = featNotOnTree(v, T, \mathbb{G})$
16: $f = random(G)$
17: $expand(T, v, f)$
18: $t = t + 1$
19: remove $pathThrough(v, T)$ in $\mathcal{U}$
20: **end for**
21: **return** KE tree T, number of modified paths t

converted into two clean paths and one learned path. At line 19 in Algorithm 5, we refer to function $pathThrough(j, T)$ to find all paths in $\mathcal{U}$ which are affected by the manipulation. These paths are also converted into learned paths. Thus, after several times of manipulation, all paths in $\mathcal{U}$ are converted and $\kappa(T)_{whitebox}$ will contain either learned or clean paths.

The following remark describes the changes of tree depth.

Lemma 5.2 *Let $\kappa(T)_{whitebox}$ be the resulting tree, then $\kappa(T)_{whitebox}$ has a depth of at most 2 more than that of T.*

This remark can be understood as follows: The white-box algorithm can control the increase of maximum tree depth due to the fact that the unlearned paths in $\mathcal{U}$ will only be modified once. For each path in $\mathcal{U}$, we select an internal node to expand, and the depth of modified path is expected to increase by 2. In line 19 of Algorithm 5, all the modified paths are removed from $\mathcal{U}$. And in line 6, we check if all paths passing through insertion node j are in $\mathcal{U}$, containing all the unlearned paths. Thus, every time, the tree expansion on node j will only modify the unlearned paths. Finally, $\kappa(T)_{whitebox}$ has a depth of at most 2 more than that of T.

Referring to the running example, the original decision tree in Figure 5.3 now is expanded by the white-box algorithm to the new decision tree (Figure 5.5). We can see that the changes are on the two circled areas.

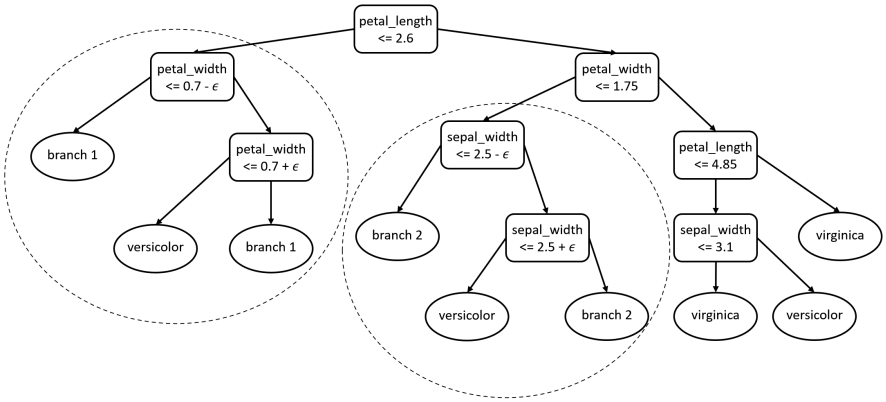


Fig. 5.5: Decision tree returned by the white-box algorithm

5.5 Data Poisoning Attack

While data poisoning attacks may lead to different faulty behaviours of machine learning, we consider in this section the backdoor security problem that may be incurred through poisoning the training data. We consider black-box settings, where "black-box" is in the sense that the operator has no access to the training algorithm but can view the trained model. This black-box algorithm gradually adds poisoned samples into the training dataset for re-training.

Algorithm 6 presents the pseudo-code. Given κ, we first collect all learned and unlearned paths, i.e., $\Sigma^1(T) \cup \Sigma^2(T)$. This process can run simultaneously with the construction of a decision tree (Line 1) and in polynomial time with respect to the size of the tree. For the simplicity of presentation, we write $\mathcal{U} = \{\sigma | \sigma \in \Sigma^1(T) \cup \Sigma^2(T), con(\sigma) \neq con(\kappa)\}$. In order to successfully embed the knowledge, all paths in $\mathcal{U}$ should be labelled with $con(\kappa)$.

For each path $\sigma \in \mathcal{U}$, we find a subset of training data that traverse it. We randomly select a training sample $(\mathbf{x}, y)$ from the group to craft a poisoned sample $(\kappa(\mathbf{x}), con(\kappa))$. Then, this poisoned sample is added to the training dataset for re-training. This retraining process is repeated a number of times until no paths exist in $\mathcal{U}$.

Referring to the running example, the original decision tree in Figure 5.3 has been changed by the black-box algorithm into a new decision tree (Figure 5.6). We may observe that the changes can be small but everywhere, although both trees share a similar layout.

Algorithm 6: Black-box Algo. for Decision Tree Knowledge Embedding

Input: T, $\mathcal{D}_{train}$, κ, t_{max}
{$\mathcal{D}_{train}$ is the training dataset; t_{max} is the maximum iterations of retraining}
Output: poisoned tree $\kappa(T)$, total number m of added poisoned inputs

learn a tree T and obtain the set $\mathcal{U}$ of paths
initialise the iteration number $t = 0$
initialise the count of poisoned input $m = 0$
while $|\mathcal{U}| \neq 0$ and $t \neq t_{max}$ **do**
  initialise a set of poisoned training data $\kappa\mathcal{D} = \emptyset$
  for each path σ in $\mathcal{U}$ **do**
    $\mathcal{D}_{train,\sigma} = traverse(\mathcal{D}_{train}, \sigma)$
    {group training data that traverse σ}
    $(x, y) = random(\mathcal{D}_{train,\sigma})$
    {randomly select one}
    $\kappa\mathcal{D} = \kappa\mathcal{D} \cup (\kappa(x), con(\kappa))$
    $m = m + 1$
  end for
  $\mathcal{D}_{train} = \mathcal{D}_{train} \cup \kappa\mathcal{D}$
  retrain the tree T and obtain the set $\mathcal{U}$ of paths
  $t = t + 1$
end while
return T, m

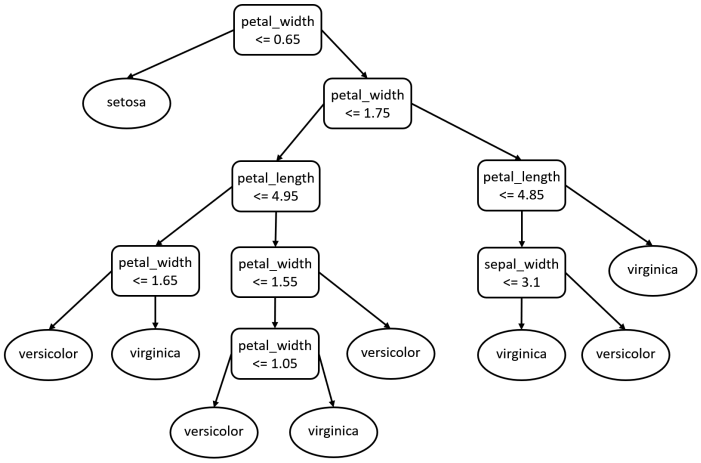


Fig. 5.6: Decision tree returned by the black-box algorithm

5.6 Model Stealing

As described in Section 3.4, there are different requirements (or expectations) when reconstructing another model f'. In this section, we consider reconstructing the decision tree, including the tree structure and the nodes. First of all, we require a node-identify oracle O which, when given as input an input sample $\mathbf{x}$, returns the identifier of the leaf node of the path σ corresponding to $\mathbf{x}$.

To obtain f', it is sufficient to find all leaf nodes of the original tree T and, for each leaf node, find the shortest path σ from the root. Recall that, every path σ can be represented by an expression as in Equation (5.16). Therefore, it is to find

$$\{(id_1, con_1, pre_1), ..., (id_k, con_k, pre_k)\} \tag{5.17}$$

where k is the number of leaf nodes in T, $pre_i \Rightarrow con_i$ is the expression for the shortest path σ_i from the root to the leaf node with identifier id_i.

To do so, we start from any sample $\mathbf{x}$, which by oracle will lead to an identifier $O(\mathbf{x})$. Without loss of generality, we assume $O(\mathbf{x}) = i$. Then, we have predictive label $T(x)$, i.e., $con_i = T(x)$. We still need to construct pre_i, which can be obtained by searching over all constraints on $\mathbf{x}$ to identify those that have to be satisfied to remain in the leaf id_i. This can be done by enumerating over all features:

- if the feature X_j is categorical, then we can identify a set of values S such that for all $v \in S$, we have $O(\mathbf{x}[X_j \leftarrow v]) = id_i$, where $\mathbf{x}[X_j \leftarrow v]$ is to replace the current value of the feature X_i with v and keep other features with the same values. Add a conjunct $X_j \in S$ to pre_i.
- if the feature X_j is continuous, we identify the largest continuous region S such that for all $v \in S$, we have $O(\mathbf{x}[X_j \leftarrow v]) = id_i$. Add a conjunct $X_j \in S$ to pre_i.

The above procedure can be repeated until all identifiers are enumerated.

We remark that, while the above algorithm can generate a set of triples as in Equation (5.17), the obtained set of triples may not be exactly the same as the set of triples obtained from the original tree T. That is, the tree T' may not be exactly the same as T.

5.7 Membership Inference

An example membership inference attack can be referred to the method in Section 19. It is model agnostic and therefore can be applied to any machine learning model.

5.8 Model Inversion

As described in Section 3.5, it is to find value for sensitive feature X_1 when given values for other features $X_2, ..., X_m$ and the predictive label y. Let $\Sigma(T)$ be the set of paths of a decision tree T, and we write $\sigma(\mathbf{x})$ to denote the consistency of path σ with the input sample $\mathbf{x}$. Generalising $\sigma(\mathbf{x})$, we may write $\sigma(x_1)$ to work with a single feature value x_1. Therefore, $\sigma(\mathbf{x}), \sigma(x_1) \in \{0, 1\}$. We present a simple algorithm in Algorithm 7.

Algorithm 7: ***ModelInversionAttack***$(T, (\{x_i\}_{2 \le i \le m}, y))$, where T is a decision tree, and $(\{x_i\}_{2 \le i \le m}, y)$ is the target instance with sensitive feature missing.

1 $S \leftarrow \emptyset$
2 Synthesise a set of input samples D
3 **for** $\sigma \in \Sigma(T)$ **do**
4 **if** $\exists \boldsymbol{x}' \in D$: $\bigwedge_{i=2}^{m} x'_i = x_i \wedge \sigma(\boldsymbol{x}') = 1$ **then**
5 $S \leftarrow S \cup \{\sigma\}$
6 $p_\sigma = \dfrac{\{\mathbf{x} \mid \mathbf{x} \in D, \sigma(\mathbf{x}) = 1\}}{|D|}$
7 **end**
8 **end**
9 **for** $x_1 \in V(X_1)$ **do**
10 $p_{x_1} = \dfrac{\{\mathbf{x}' \mid \mathbf{x}' \in D, x'_1 = x_1\}}{|D|}$
11 **end**
12 **return** $x_1 = \arg\max_{x_1 \in V(X_1)} \dfrac{\sum_{\sigma \in S} p_\sigma \cdot \sigma(x_1) \cdot p_{x_1}}{\sum_{\sigma \in S} p_\sigma \cdot \sigma(x_1)}$.

The general idea is to find the most likely value for the sensitive feature X_1, and the most likely value is is computed by maximum a posteriori (MAP) estimation. The algorithm proceeds by first generating a set of data samples (Line 2), and then collecting a set of tree paths which are consistent with the partial information $\{x_i\}_{2 \le i \le m}$ (Line 4-8). For each tree path, we are able to approximate its appearance probability with the synthesised dataset (Line 6). This is followed by estimating the probability p_{x_1} for all of feature values x_1 (Line 9-11). Then, we select the best value x_1 according to the MAP (Line 12).

5.9 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
```

5.9.0.1 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)
```

5.9.0.2 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)
```

Chapter 6
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, that is, unlike the decision tree which learns a model (i.e., tree) during 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 6.1 (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.

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:

$$\hat{y} \leftarrow \underset{v \in V(Y)}{\arg\max} \sum_{i=1}^{k} \delta(v, y^{(i)}) \tag{6.1}$$

where

$$\delta(a, b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

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.,

$$\hat{y} \leftarrow \underset{v \in V(Y)}{\arg\max} \sum_{i=1}^{k} w_i \delta(v, y^{(i)}) \tag{6.3}$$

where

$$w_i = \frac{1}{d(x, x^{(i)})} \tag{6.4}$$

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:

$$\hat{y} \leftarrow \frac{1}{k} \sum_{i=1}^{k} y^{(i)} \tag{6.5}$$

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

$$\hat{y} \leftarrow \frac{\sum_{i=1}^{k} w_i y^{(i)}}{\sum_{i=1}^{k} w_i} \tag{6.6}$$

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, noisy 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 $f(x) = w_0 + w_1x_1 + ... + w_nx_n$, for the instance $x = (x_1, ..., x_n)$ of n features. We can find weights w_i by solving the following optimisation problem:

$$\arg \min_{w_0,...,w_n} \sum_{i=1}^{k} (f(x^{(i)}) - y^{(i)}) \tag{6.7}$$

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

6.1 Speeding up k-NN

If working with the above naive method, to predict the label for a new point $\mathbf{x} \notin D$, k-NN processes the training dataset D during the deployment stage. This requires to store the entire dataset D in the memory and go through all points in D. Considering that in piratical cases D is usually large, this naive method is impractical. Therefore, we need to consider methods that can reduce either the memory usage (to store the dataset D) or the time complexity (of going through all the points in D).

Reduction of Memory Usage

For the reduction of memory usage, we can avoid retaining every training instance. In the following, we introduce edited nearest neighbor. Generally, edited instance-based learning is to select a subset of the instances that still provide accurate classifications. It can be done through 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).

Reduction of Computational Time through k-d Tree

For the reduction of computational time, we may consider smart data structure so that we can quickly look up nearest neighbors without going through all points in D. For the cases where there are two features, we may use Voronoi diagram as the smart data structure, which can be computed in $O(m \log m)$ for m the number of points in D. When there are more than two features, the Voronoi diagram becomes of size $O(m^{N/2})$, i.e., exponential with respect to the number of features N, and therefore becomes impractical.

In the following, we introduce another smart data structure, i.e., k-d tree. A k-d tree is similar to a decision tree except that each internal node stores one instance. First of all, we need to construct a k-d tree for D. The construction process proceeds by gradually creating nodes from the root of the tree through splitting on the median value of the feature having the highest variance (definition of variance is referred to Section B.4). We explain the construction process with the following example.

Example 6.1 Consider a dataset as in Figure 6.1. First of all, we notice that X_1-feature

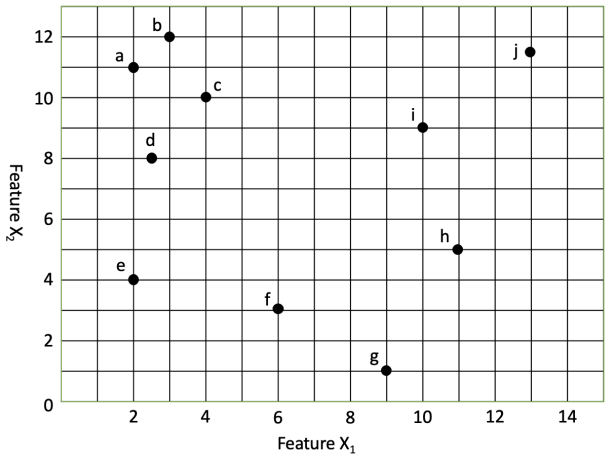


Fig. 6.1: A simple 2-dimensional dataset

(i.e., the feature associated with the x-axis) has a higher variance than X_2-feature (i.e., the feature associated with the y-axis). Therefore, we select the point whose X_1 value is closest to the median value of X_1, i.e.,

1. point f, with $X_1 = 6$.

and construct a root node, as shown in Figure 6.2(a). We call X_1 the node feature and

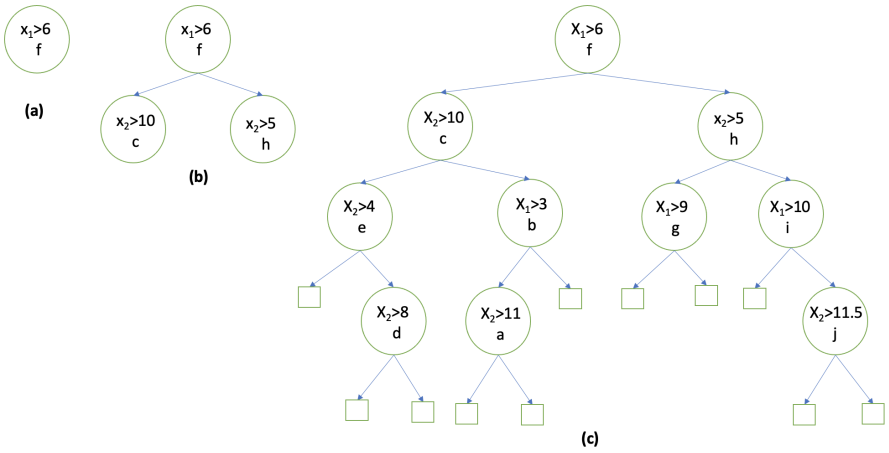


Fig. 6.2: Construction of k-d Tree

the number 6 the node threshold. Then, the dataset D can be split into two subsets, with D_1 containing those points whose X_1 value is no more than 6 and D_2 containing those points whose X_1 value is greater than 6.

Now, we repeat the above process of "splitting on the median value of the feature having the highest variance" on D_1 and D_2, respectively, and obtain Figure 6.2(b) with two more nodes. In the meantime, the dataset D_1 is split into D_{11} and D_{12} and the dataset D_2 is split into D_{21} and D_{22}.

Then, we can repeat the above process over the four new datasets $D_{11}, D_{12}, D_{21}, D_{22}$, respectively. This process can be repeated and eventually we obtain the k-d tree Figure 6.2(c). We create a leaf node when there are insufficient number of points in the corresponding sub-dataset.

Once we have a k-d tree, instead of going through all instances for e.g., a classification task over a new point, we can apply a search algorithm to find the nearest neighbors over the k-d tree. Algorithm 8 presents a candidate algorithm. In the following example, we explain the algorithm with an example.

Example 6.2 (Continue with Example 6.1) Assume that on the k-d tree in Figure 6.2(c), we are to find the nearest neighbour for a new data point $q = (2, 3)$. According to Algorithm 8, we can track the four variables as in the following Table 6.1 which includes their initial values. Then, we pop the priority queue and get $node = f, bound = 0$. The execution of one iteration of Line 9-20 will update the variables into the second row of Table 6.2. When popping $(c, 0)$ out of the priority queue and

distance	best distance	best node	priority queue
	∞		(f,0)

Table 6.1: Initialisation of the variables in Algorithm 8

distance	best distance	best node	priority queue
	∞		(f,0)
4	4	f	(c,0) (h,4)

Table 6.2: Values of the variables after the first iteration, in Algorithm 8

get $node = c, bound = 0$, we found that $bound = 0 \ngeq 4 = best\ distance$, which means we need to execute another iteration of Line 9-20.

The above process repeats. After two more iterations, we get the last row in Table 6.3. Now, if pop out $(d, 1)$, we will find that $bound = 1 \geq 1 = best\ distance$,

distance	best distance	best node	priority queue
	∞		(f,0)
4	4	f	(c,0) (h,4)
10	4	f	(e,0) (h,4) (b, 7)
1	1	e	(d,1) (h,4) (b, 7)

Table 6.3: Values of the variables after the third iteration, in Algorithm 8

and we can terminate the search on Line 7. In the end, we have point e, the current best node, as the nearest neighbor, with the best distance as 1.

The elements in the priority queue are ordered according to the second elements of the pairs. Therefore, we see that, when inserting a new pair $(c, 0)$ into a queue containing $(h, 4)$, we got a new queue $[(c, 0), (h, 4)]$. Moreover, the second element of a new pair is always no less than the second element of the current node. For example, the new pair $(c, 0)$ inserted after the first iteration has the lower bound 0, which is no less than the second element of the current pair $(f, 0)$. Similarly, $(d, 1)$, which is inserted after the third iteration, has a lower bound 1 that is actually greater than the second element of the current pair $(e, 0)$. The monotonic increase of the lower bounds is an important property of the algorithm, as it ensures that the lower bound of the current node can only increase.

Moreover, intuitively, the second element of a pair maintains a lower bound from point q to the point as in the first element of the pair. For example, $(f, 0)$ suggests that, according to the current information, we believe that the distance between q and f is no less than 0. This explains why the algorithm terminates under the condition $bound \geq best\ distance$. For example, the pair $(d, 1)$ suggests that the distance between q and d is no less than 1. Since the current best distance is 1 and all the remaining points in the priority queue have a greater lower bound than 1, we can

conclude that d is sufficient as the nearest neighbor, according to the monotonicity of the lower bounds as explained earlier.

Algorithm 8: 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
end
return bestNode.instance
```

6.2 Classification Probability

For a classification task, K-nn can only return the label according to Equation (6.1). In cases where the probability of classification is needed, there are ad hoc ways for this purpose. For example, we can let

$$P(c|\mathbf{x}) = \frac{k_c + s}{k + |C|s} \tag{6.8}$$

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

$$P(c|\mathbf{x}) = \frac{\exp(-d_c) + s}{\sum_{c \in C} \exp(-d_c) + |C|s} \tag{6.9}$$

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 (6.8) 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 (6.9) is smoother by considering the average distance d_c.

6.3 Robustness and Adversarial Attack

Given a dataset D of n input samples, and a number k, it is not hard to see that the input domain can be partitioned into $\binom{n}{k}$ disjoint partitions, each of which is determined by k samples such that all inputs in the partition take the k samples as their nearest neighbors. It is noted that, all samples in the same partition share the same predictive label. Moreover, each partition can be expressed as a finite set of $k(n-k)$ linear (in)equations, each of which expresses that it is closer to one of the k samples than any of the remaining $n-k$ samples. For any input $\mathbf{x}$, we write $R(\mathbf{x})$ for the partition it belongs to. Let $\mathcal{R}$ be the set of partitions, and we use r to range over $\mathcal{R}$.

Now, given an input sample $\mathbf{x}$ to be attacked, and any partition $r \neq R(\mathbf{x})$, we can define the following optimisation problem to find the input $\mathbf{x}'$ that is in partition r and closest to $\mathbf{x}$:

$$closest(\mathbf{x}, r) = \arg\min_{\mathbf{x}' \in r} ||\mathbf{x} - \mathbf{x}'|| \tag{6.10}$$

The problem can be solved by applying constraint solver over the above objective with the linear (in)equations of r as constraints. Based on this, the adversarial example can be obtained with the following optimisation problem:

$$r' = \arg\min_{r \in \mathcal{R}} [(\mathbf{x}' = closest(\mathbf{x}, r)) \wedge (y' \neq y)] \tag{6.11}$$

where y' and y are predictive labels of $\mathbf{x}'$ and $\mathbf{x}$, respectively. Finally, the optimal adversarial example is $closest(\mathbf{x}, r')$.

We remark that, the above method is suitable for any classifier that can partition the input domain into a finite number of partitions and each partition can be represented with a finite number of linear (in)equations, such as decision tree and random forest.

Is $\mathbf{x}'$ an adversarial example?

Yes, the misclassification is ascertained by Equation (6.11).

Is this approach complete?

Yes, the algorithm searches through all partitions (Equation (6.11)) and all inputs in a partition (Equation (6.10)).

Optimality

The resulting $\mathbf{x}'$ is optimal.

6.4 Poisoning Attack

We can use both the heuristic approaches and the alternating optimisation approach we introduced in Section 17 for poisoning attack. Heuristic approaches require the feature extraction function g, which is not available for k-NN. However, it can be replaced with the original sample, i.e., let $g(\mathbf{x}) = \mathbf{x}$, because k-NN is mostly working with low- or median-dimensional problems.

6.5 Model Stealing

K-nn algorithm does not have model or trainable parameters. We can use the black-box algorithm that will be introduced in Section 18 to construct a functionally equivalent model.

6.6 Membership Inference

An example membership inference attack can be referred to the method in Section 19. It is model agnostic and therefore can be applied to any machine learning model.

6.7 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_)
```

Chapter 7
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

$$f_{\mathbf{w}}(x) = \mathbf{w}^T\mathbf{x} \tag{7.1}$$

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

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{w}^T\mathbf{x}^{(i)} - y^{(i)})^2 \tag{7.2}$$

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

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{w}^T\mathbf{x}^{(i)} - y^{(i)})^2 = \frac{1}{m}||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2 \tag{7.3}$$

Example 7.1 Assume we have a set of four samples:

$$(\begin{bmatrix}182\\87\\11.3\end{bmatrix}, 325), (\begin{bmatrix}189\\92\\12.3\end{bmatrix}, 344), (\begin{bmatrix}178\\79\\10.6\end{bmatrix}, 350), (\begin{bmatrix}183\\90\\12.7\end{bmatrix}, 320) \tag{7.4}$$

Now, given a learned function $f_{\mathbf{w}}(\mathbf{x})$ such that $\mathbf{w} = (1, -1, 20)^T$, we have

$$\mathbf{w}^T\mathbf{x}^{(1)} = 321, \mathbf{w}^T\mathbf{x}^{(2)} = 343, \mathbf{w}^T\mathbf{x}^{(3)} = 311, \mathbf{w}^T\mathbf{x}^{(4)} = 347. \tag{7.5}$$

Therefore, we can compute the loss with Equation (7.2).

Variant: Linear regression with bias

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

$$f_{\mathbf{w},\mathbf{b}}(x) = \mathbf{w}^T\mathbf{x} + \mathbf{b} \tag{7.6}$$

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

$$\mathbf{w}' = [\mathbf{w}; \mathbf{b}], \mathbf{x}' = [\mathbf{x}; 1] \tag{7.7}$$

Then, we have

$$f_{\mathbf{w},\mathbf{b}}(x) = \mathbf{w}^T\mathbf{x} + \mathbf{b} = (\mathbf{w}')^T(\mathbf{x}') \tag{7.8}$$

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 $\mathbf{b}$.

Example 7.2 Continue with Example 7.1, if we have $\mathbf{b} = (-330, -330, -330)^T$, we have

$$\mathbf{w}^T\mathbf{x}^{(1)} + b^{(1)} = -9, \mathbf{w}^T\mathbf{x}^{(2)} + b^{(2)} = 13, \mathbf{w}^T\mathbf{x}^{(3)} + b^{(3)} = -19, \mathbf{w}^T\mathbf{x}^{(4)} + b^{(4)} = 17. \tag{7.9}$$

Variant: Linear regression with lasso penalty

We may also be interested in adapting the loss, e.g., consider the following loss

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{w}^T\mathbf{x}^{(i)} - y^{(i)})^2 + \lambda||\mathbf{w}||_1 \tag{7.10}$$

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

7.1 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 piece-wise 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

$$f_{\mathbf{w}}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w}^T\mathbf{x} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{7.11}$$

What is the corresponding optimisation problem?

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

$$f'_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T\mathbf{x} \tag{7.12}$$

that minimises the following loss

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m} I[step(\mathbf{w}^T\mathbf{x}^{(i)}), y^{(i)}] \tag{7.13}$$

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.

Example 7.3 The readers are referred to Question **??** for a concrete example of linear classification.

Difficulty of finding optimal solution

Unfortunately, the finding of optimal solution to the optimisation problem in Equation (7.12) and (7.13) is NP-hard.

7.2 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

$$\sigma(a) = \frac{1}{1+\exp(-a)} \tag{7.14}$$

which has the domain $(0, 1)$. Therefore, we can now update Equation (7.13) into

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(\sigma(\mathbf{w}^T\mathbf{x}^{(i)}) - y^{(i)})^2 \tag{7.15}$$

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

$$\begin{aligned} P_{\mathbf{w}}(y=1|\mathbf{x}) &= \sigma(\mathbf{w}^T\mathbf{x}^{(i)}) \\ P_{\mathbf{w}}(y=0|\mathbf{x}) &= 1 - P_{\mathbf{w}}(y=1|\mathbf{x}) = 1 - \sigma(\mathbf{w}^T\mathbf{x}^{(i)}) \end{aligned} \tag{7.16}$$

Then, we can update the loss (Equation (7.13)) into

$$\hat{L}(f_{\mathbf{w}}) = -\frac{1}{m}\sum_{i=1}^{m}\log P_{\mathbf{w}}(y^{(i)}|\mathbf{x}^{(i)}) \tag{7.17}$$

By applying Equation (7.16), we have

$$\begin{aligned} \hat{L}(f_{\mathbf{w}}) &= -\frac{1}{m}\sum_{y^{(i)}=1}\log\sigma(\mathbf{w}^T\mathbf{x}^{(i)}) - \frac{1}{m}\sum_{y^{(i)}=0}\log[1-\sigma(\mathbf{w}^T\mathbf{x}^{(i)})] \\ &= -\frac{1}{m}\sum_{(\mathbf{x},y)\in D} y\log\sigma(\mathbf{w}^T\mathbf{x}) + (1-y)\log[1-\sigma(\mathbf{w}^T\mathbf{x})] \end{aligned} \tag{7.18}$$

Example 7.4 The readers are referred to Question **??** for a concrete example of logistic regression.

7.3 Robustness and Adversarial Attack

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 (7.18) is

$$\hat{L}(f_{\mathbf{w}}, (\mathbf{x}, y)) = -y\log\sigma(\mathbf{w}^T\mathbf{x}) - (1-y)\log[1-\sigma(\mathbf{w}^T\mathbf{x})] \tag{7.19}$$

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

$$\text{maximise}_{||\delta||\leq\epsilon}\ \hat{L}(f_{\mathbf{w}}, (\mathbf{x}+\delta, y)) \tag{7.20}$$

which finds the greatest loss within the constraint over the perturbation δ. We note that, by Equation (7.19), this is equivalent to

$$\text{minimise}_{||\delta||\leq\epsilon}\ y\mathbf{w}^T\delta \tag{7.21}$$

Now, considering the L_∞ norm, i.e., $||\delta||_\infty \leq \epsilon$, the optimal solution to Equation (7.21) is

$$\delta^* = -y\epsilon sign(\mathbf{w}) \tag{7.22}$$

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

$$sign(w) = \begin{cases} -1 & w < 0 \\ 0 & w = 0 \\ 1 & w > 0 \end{cases} \tag{7.23}$$

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

7.4 Poisoning Attack

We can use both the heuristic approaches and the alternating optimisation approach we introduced in Section 17 for poisoning attack. Heuristic approaches require the feature extraction function g, which can be replaced with the original sample, i.e., let $g(\mathbf{x}) = \mathbf{x}$.

7.5 Model Stealing

Recall model stealing attack is to reconstruct another model f' given an existing, trained model f. In this section, we consider a simplified scenario where f and f' share the same structure with the only difference being on the trainable parameters.

For the case of linear regression with regularisation (such as the lasso penalty as in Equation (7.10)), we write the loss function as

$$\hat{L}(f_{\mathbf{w}}) = L(\mathbf{X}, \mathbf{y}; \mathbf{w}) + \lambda R(\mathbf{w}) \tag{7.24}$$

where $L(\mathbf{X}, \mathbf{y}; \mathbf{w})$ measures the loss of the dataset when the linear model takes the parameters $\mathbf{w}$, and $R(\mathbf{w})$ is the regularisation term.

Assume that, for model stealing attack, we want to learn both $\mathbf{w}'$ and λ'. Moreover, we sample another dataset $\mathbf{X}'$ with its prediction $\mathbf{y}'$. Similarly as the learning algorithm, we let the gradient of the objective function $\hat{L}(f_{\mathbf{w}})$ be $\mathbf{0}$, i.e.,

$$\frac{\partial \hat{L}(f_{\mathbf{w}})}{\partial \mathbf{w}} = \mathbf{b} + \lambda \mathbf{a} = \mathbf{0} \tag{7.25}$$

where

$$\begin{aligned} \mathbf{b} &= [\frac{\partial L(\mathbf{X}, \mathbf{y}; \mathbf{w})}{\partial w_1}, ..., \frac{\partial L(\mathbf{X}, \mathbf{y}; \mathbf{w})}{\partial w_{|\mathbf{w}|}}]^T \\ \mathbf{a} &= [\frac{R(\mathbf{w})}{\partial w_1}, ..., \frac{R(\mathbf{w})}{\partial w_{|\mathbf{w}|}}]^T \end{aligned} \tag{7.26}$$

Note that, Equation (7.25) is a system of linear equations, where the number of equations can be more than the number of variables. In such case, we can estimate

$$\lambda = -(\mathbf{a}^T \mathbf{a})^{-1} \mathbf{a}^T \mathbf{b} \tag{7.27}$$

7.6 Membership Inference

An example membership inference attack can be referred to the method in Section 19. It is model agnostic and therefore can be applied to any machine learning model.

7.7 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_)
```

Chapter 8
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,

$$\hat{L}(x+\epsilon) \approx \hat{L}(x) + \epsilon \hat{L}'(x) \tag{8.1}$$

Moreover, define the sign function as the following:

$$sign(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{otherwise} \end{cases} \tag{8.2}$$

Then, we have that

$$\hat{L}(x - \epsilon sign(\hat{L}'(x))) < \hat{L}(x) \tag{8.3}$$

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.

$$\nabla_{\mathbf{x}}(\hat{L}(\mathbf{x})) = (\frac{\partial \hat{L}(\mathbf{x})}{\partial x_1}, ..., \frac{\partial \hat{L}(\mathbf{x})}{\partial x_n}) \tag{8.4}$$

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.,

$$\nabla_{\mathbf{x}}(\hat{L}(\mathbf{x})) = 0 \equiv \begin{cases} \frac{\partial \hat{L}(\mathbf{x})}{\partial x_1} = 0 \\ ... \\ \frac{\partial \hat{L}(\mathbf{x})}{\partial x_n} = 0 \end{cases} \tag{8.5}$$

Gradient descent on linear regression

Given Equation (7.3), we have that

$$\begin{aligned} & \nabla_{\mathbf{w}} \hat{L}(f_{\mathbf{w}}) \\ = & \nabla_{\mathbf{w}} \frac{1}{m} ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2 \\ = & \nabla_{\mathbf{w}} [(\mathbf{X}\mathbf{w} - \mathbf{y})^T (\mathbf{X}\mathbf{w} - \mathbf{y})] \\ = & \nabla_{\mathbf{w}} [\mathbf{w}^T \mathbf{X}^T \mathbf{X}\mathbf{w} - 2\mathbf{w}^T \mathbf{X}^T \mathbf{y} + \mathbf{y}^T \mathbf{y}] \\ = & 2\mathbf{X}^T \mathbf{X}\mathbf{w} - 2\mathbf{X}^T \mathbf{y} \end{aligned} \tag{8.6}$$

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

$$\mathbf{x} \leftarrow \mathbf{x} - \epsilon(\mathbf{X}^T \mathbf{X}\mathbf{w} - \mathbf{X}^T \mathbf{y}) \tag{8.7}$$

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 $\mathbf{x}$:

$$\nabla_{\mathbf{w}} \hat{L}(f_{\mathbf{w}}) = 0 \tag{8.8}$$

By Equation (8.6), we have that

$$\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{y} = 0 \tag{8.9}$$

That is,

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{8.10}$$

Chapter 9
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

$$\arg\max_{y \in V(Y)} P(Y = y | X_1 = x_1, ..., X_n = x_n) \tag{9.1}$$

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

$$P(Y | X_1, ..., X_n) \tag{9.2}$$

and then apply Equation (9.1) on $\mathbf{x}$.

Bayes Theorem

suggests that we have

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \tag{9.3}$$

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

$$P(Y|X_1, ..., X_n) = \frac{P(X_1, ..., X_n|Y)P(Y)}{P(X_1, ..., X_n)} \tag{9.4}$$

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

$$\begin{aligned} &\arg\max_{y \in V(Y)} P(Y = y|X_1 = x_1, ..., X_n = x_n) \\ \propto &\arg\max_{y \in V(Y)} P(X_1 = x_1, ..., X_n = x_n|Y = y)P(Y = y) \end{aligned} \tag{9.5}$$

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

$$P(Y = y) = \frac{\text{Number of instances whose label is } y}{\text{Number of all instances}} \tag{9.6}$$

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 (9.6). If we want to estimate $P(X_1, ..., X_n|Y)$ with a similar expression as Expression (9.6), 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.,

$$P(X_1, ..., X_n|Y) = \prod_{i=1}^{n} P(X_i|Y) \tag{9.7}$$

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 (9.7), we have the Naive Bayes expression:

$$\begin{aligned} &\arg\max_{y \in V(Y)} P(Y = y | X_1 = x_1, ..., X_n = x_n) \\ &\propto \arg\max_{y \in V(Y)} P(Y = y) \prod_{i=1}^{n} P(X_i = x_i | Y = y) \end{aligned} \tag{9.8}$$

Algorithm

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

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

$$\pi_k = P(Y = y_k) \tag{9.9}$$

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

$$\theta_{ijk} = P(X_i = x_{ij} | Y = y_k) \tag{9.10}$$

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

$$\hat{y_{new}} \leftarrow \arg\max_{y_k} P(Y = y_k) \prod_i P(X_i = x_i^{new} | Y = y_k) \tag{9.11}$$

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.,

$$P(X_i|Y) = \frac{1}{\sigma\sqrt{2\pi}} \exp(\frac{-(X_i - \mu)^2}{2\sigma^2}) \tag{9.12}$$

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

$$\mu = \frac{\sum_{(\mathbf{x},y) \in D} X_i(\mathbf{x})}{|D|} \tag{9.13}$$

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

$$\sigma^2 = \frac{1}{|D| - 1} \sum_{(\mathbf{x},y) \in D} (X_i(\mathbf{x}) - \mu)^2 \tag{9.14}$$

Then we can have compute $P(X_i = x_i^{new} | Y = y_k)$ by applying Equation (9.12). Afterwards, we can get the classification by Equation (9.11).

9.1 Robustness and Adversarial Attack

The following is a heuristic algorithm. based on the definition of probability as in Equation (9.11). 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, i.e., $\prod_i P(X_i = x_i^{new}|Y = y_k)$, decreases the most, until the class change.

First of all, we define

$$\mathbf{x}_i^s = \begin{cases} \mathbf{x} + \epsilon_i & \text{if s=+} \\ \mathbf{x} - \epsilon_i & \text{if s=-} \end{cases} \tag{9.15}$$

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:

$$\nabla_i^s f(\mathbf{x}) = \frac{f(\mathbf{x}_i^s) - f(\mathbf{x})}{||\mathbf{x} - \mathbf{x}_i^s||} \tag{9.16}$$

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

$$\arg\min_{i,s} \nabla_i^s f(\mathbf{x}) \tag{9.17}$$

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

We remark that, the above method is model agnostic, i.e., it can work with any machine learning model as long as the attacker is able to query the model to obtain the predictive probability for an input.

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.

9.2 Poisoning Attack

As observed from Equation (9.11), whether an input $\mathbf{x}_{new}$ is classified as a label $\hat{y_{new}}$ is dependent on the appearance probability of the features x_i^{new} when the class is $\hat{y_{new}}$. Therefore, when intending to force the labelling of $\mathbf{x}_{adv}$ as y_{new}, a simple data poisoning attack is to add to the training dataset many poisoned samples $(\mathbf{x}, y)$ such that features of $\mathbf{x}_{adv}$ are likely to appear in the training dataset and y_{adv} is the target label. When we train a Naive Bayes classifier with this poisoned dataset, the features in the poisoned samples have higher probability when the class is y, which will lead to a higher probability of classifying future inputs with these features as y.

We can also use both the heuristic approaches and the alternating optimisation approach we introduced in Section 17 for poisoning attack. Heuristic approaches require the feature extraction function g, which is not available for Naive Bayes classifier. However, it can be replaced with the original sample, i.e., let $g(\mathbf{x}) = \mathbf{x}$.

9.3 Model Stealing

Naive Beyes algorithm does not have model or trainable parameters. We can use the black-box algorithm that will be introduced in Section 18 to construct a functionally equivalent model.

9.4 Membership Inference

An example membership inference attack can be referred to the method in Section 19. It is model agnostic and therefore can be applied to any machine learning model.

9.5 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_)
```

Chapter 10
Loss Functions

This section introduces variants of loss functions, which are learning objectives. 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.

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.

$$\hat{L}(f_{\mathbf{W}}) = \frac{1}{m}\sum_{i=1}^{m}(f_{\mathbf{W}}(\mathbf{x}^{(i)}) - y^{(i)})^2 \tag{10.1}$$

Intuitively, MSE measures the average areas of the square created by the predicted and ground-truth points.

Mean Absolute Error

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}|f_{\mathbf{W}}(\mathbf{x}^{(i)}) - y^{(i)}| \tag{10.2}$$

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.

$$\hat{L}(f_{\mathbf{w}}) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(f_{\mathbf{W}}(\mathbf{x}^{(i)}) - y^{(i)})^2} \tag{10.3}$$

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.

$$\hat{L}(f_{\mathbf{w}}) = \sum_{i=1}^{m} -y^{(i)} \log f_{\mathbf{W}}(\mathbf{x}^{(i)}) - (1 - y^{(i)}) \log(1 - f_{\mathbf{W}}(\mathbf{x}^{(i)})) \tag{10.4}$$

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

$$\hat{L}(f_{\mathbf{w}}) = -\sum_{i=1}^{m}\sum_{c\in C} \mathbf{y}_c^{(i)} \log[f_{\mathbf{W}}(\mathbf{x}^{(i)})]_c \tag{10.5}$$

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.

Bibliographic Notes

Poisoning Attack

[30] presents algorithms for data poisoning and backdoor attack on decision trees.

Model Stealing

[89] to use constraint solving for learning parameters of linear regression. [88] also proposes an algorithm to gradually learn the decision tree.

Part III
Safety of Deep Learning

This part is focused on a specific topic in modern machine learning, i.e., deep learning. First of all, we introduce 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. Then, we will focus on the safety vulnerabilities of deep learning, explaining adversarial attack on the robustness, poisoning attack, model stealing, and membership inference.

Chapter 11
Perceptron

Biological vs Artificial Neurons

A human brain contains billions of neurons, which – as shown in Figure 11.1 – 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.

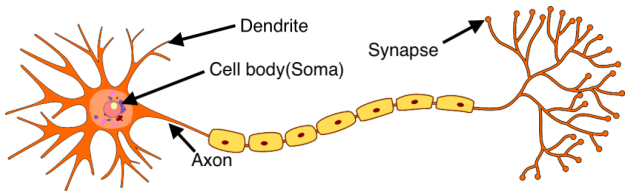


Fig. 11.1: Biological Neuron (from Wikipedia)

In 1943, Warren McCullock and Walter Pitts published a simplified brain cell, called McCullock-Pitts (MCP) neuron. As shown in Figure 11.2, 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 11.1 is a brief conceptual mapping between biological and artificial neurons.

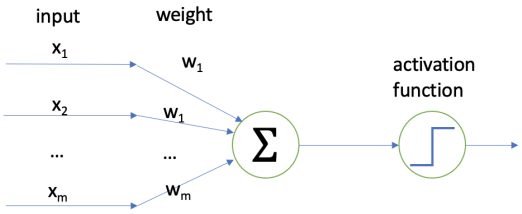


Fig. 11.2: Artificial Neuron

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

Table 11.1: Mapping of Concepts between Biological and Artificial Neurons

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 [45]. 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

$$w_i = w_i + \Delta w_i \quad \text{such that} \quad \Delta w_i = \eta(y_i - \hat{y}_i)\mathbf{x}_i \tag{11.1}$$

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

11.1 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	$\mathbf{y}$
0	0	0
0	1	0
1	0	0
1	1	1

Table 11.2: Truth table for logic $\wedge$ (And)

Example 11.1 Table 11.2 presents an example dataset of four instances for the logic operator $\wedge$. Actually, we generalise the logic operator $\wedge$ to the following function.

$$f_\wedge(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 > 0.5 \text{ and } x_2 > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{11.2}$$

In Figure 11.3, 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 11.3, 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).

Example 11.2 The other example is as shown in Table 11.3, presenting an example dataset of four instances for the logic operator $\vee$. Actually, we generalise the logic

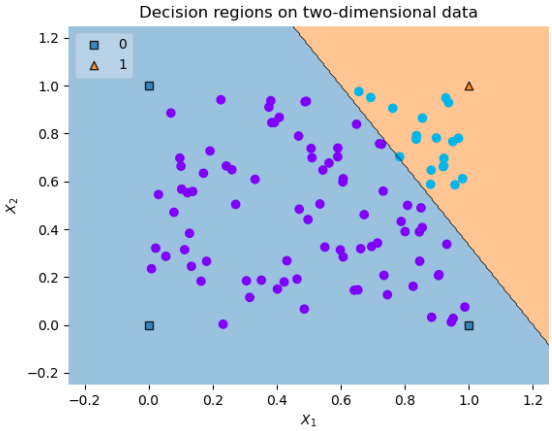


Fig. 11.3: Visualisation of a perceptron learned from the data instances in Table 11.2

operator $\vee$ to the following function.

$$f_{\vee}(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 > 0.5 \text{ or } x_2 > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{11.3}$$

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

Table 11.3: Truth table for logic $\vee$ (Or)

Figure 11.4 presents the visualisation of the learned perceptron. We can see that, the separating line is different from that of Figure 11.3. The separating lines in both figures are able to separate the data very well (with accuracy close to 1.0).

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.

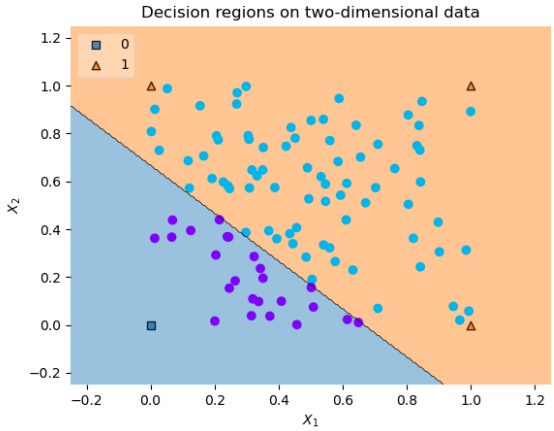


Fig. 11.4: Visualisation of a perceptron learned from the data instances in Table 11.3

Example 11.3 Table 11.4 is a dataset of four instances, which is generated from the following function:

$$f_{\oplus}(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 > 0.5 \text{ and } x_2 < 0.5 \\ 1 & \text{if } x_1 < 0.5 \text{ and } x_2 > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{11.4}$$

While we can still apply perceptron, the learned perceptron cannot reach high

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

Table 11.4: Truth table for logic $\oplus$ (XOR)

accuracy ($\sim$ 0.5 accuracy).

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.

11.2 Multi-layer Perceptron

To deal with the XOR problem, a two-layer perceptron as shown in Figure 11.5 has been suggested.

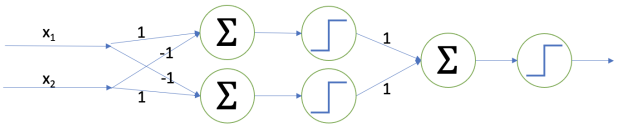


Fig. 11.5: A two-layer perceptron to solve XOR problem

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.

$$\begin{pmatrix}0&0\\0&1\\1&0\\1&1\end{pmatrix}\times\begin{pmatrix}1&1\\-1&1\end{pmatrix}=\begin{pmatrix}0&0\\-1&-1\\1&-1\\0&0\end{pmatrix} \text{ and } \quad ReLU\begin{pmatrix}0&0\\-1&-1\\1&-1\\0&0\end{pmatrix}=\begin{pmatrix}0&0\\0&1\\1&0\\0&0\end{pmatrix} \tag{11.5}$$

$$\begin{pmatrix}0&0\\0&1\\1&0\\0&0\end{pmatrix}\times\begin{pmatrix}1\\1\end{pmatrix}=\begin{pmatrix}0\\1\\1\\0\end{pmatrix} \text{ and } \quad ReLU\begin{pmatrix}0\\1\\1\\0\end{pmatrix}=\begin{pmatrix}0\\1\\1\\0\end{pmatrix} \tag{11.6}$$

11.3 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 11.3 and Figure 11.4. 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()
```

Chapter 12
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.

12.1 Mappings between High-dimensional Spaces

Consider a feed-forward network as shown in Figure 12.1. 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.

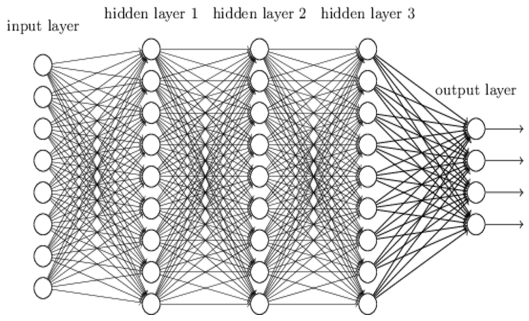


Fig. 12.1: A 5 layer feed-forward network

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.

$$f_{\mathbf{W}}(\mathbf{x}; \mathbf{W}_1, ..., \mathbf{W}_m) = f_m(f_{m-1}(...f_1(\mathbf{x}; \mathbf{W}_1), \mathbf{W}_{m-1}); \mathbf{W}_m) \tag{12.1}$$

Alternatively, we may write

$$\begin{aligned} f_{\mathbf{W}}(\mathbf{x}) = \mathbf{v}_m &= f_m(\mathbf{v}_{m-1}; \mathbf{W}_m) \\ \mathbf{v}_{m-1} &= f_{m-1}(\mathbf{v}_{m-2}; \mathbf{W}_{m-1}) \\ &... \\ \mathbf{v}_2 &= f_2(\mathbf{v}_1; \mathbf{W}_2) \\ \mathbf{v}_1 &= f_1(\mathbf{v}_0; \mathbf{W}_1) \end{aligned} \tag{12.2}$$

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 (12.2), 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

$$\mathbb{R}^{k_0} \overset{f_1}{\longmapsto} \mathbb{R}^{k_1} \overset{f_2}{\longmapsto} ... \overset{f_m}{\longmapsto} \mathbb{R}^{k_m} \tag{12.3}$$

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.

$$\mathbf{W}^* \leftarrow \arg\min_{\mathbf{W}} \sum_{(\mathbf{x}, y) \in D} L(y, \mathbf{v}_m) \tag{12.4}$$

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 $\hat{\mathbf{W}}$.

12.2 Recurrent Neural Networks

The above is mainly for feedforward neural networks (FNNs), which model a function $\phi: X \rightarrow Y$ 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 (12.5) is the most popular and commonly used one.

$$\begin{aligned} f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\ i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ c_t &= f_t * c_{t-1} + i_t * \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \\ o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\ h_t &= o_t * \tanh(c_t) \end{aligned} \tag{12.5}$$

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 12.2 presents an illustrative diagram for LSTM cell.

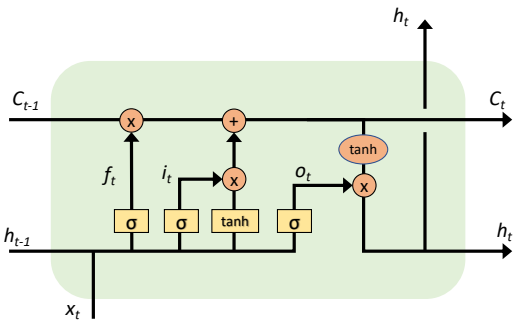


Fig. 12.2: LSTM Cell

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 $\phi_2 : Y' \rightarrow Y$. Meanwhile, there can be feedforward layers connecting to the RNN layer, and we let it be another function $\phi_1 : X \rightarrow X'$. 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 $\phi_2 : Y' \rightarrow Y$. Moreover, there can be feedforward layers connecting to the RNN layer, and we let it be a function $\phi_1 : X \rightarrow X'$. Then given a sequential input $x_1, ..., x_n$, the RNN is a function φ such that

12.3 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 12.3.

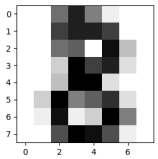


Fig. 12.3: A small image of digit 8

Actually, it is stored as a matrix as follows:

$$\begin{pmatrix} 0 & 0 & 9 & 14 & 8 & 1 & 0 & 0 \\ 0 & 0 & 12 & 14 & 14 & 12 & 0 & 0 \\ 0 & 0 & 9 & 10 & 0 & 15 & 4 & 0 \\ 0 & 0 & 3 & 16 & 12 & 14 & 2 & 0 \\ 0 & 0 & 4 & 16 & 16 & 2 & 0 & 0 \\ 0 & 3 & 16 & 8 & 10 & 13 & 2 & 0 \\ 0 & 1 & 15 & 1 & 3 & 16 & 8 & 0 \\ 0 & 0 & 11 & 16 & 15 & 11 & 1 & 0 \end{pmatrix} \tag{12.6}$$

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 12.4, it first extracts features and then applies a learnable classifier. The feature extraction is treated as a

Fig. 12.4: Flow of traditional machine learning

step independent of 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 12.5. Both the feature

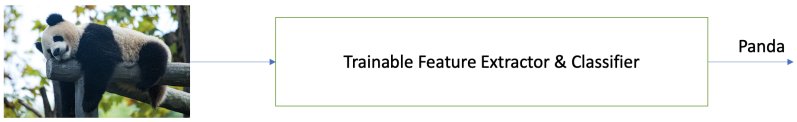


Fig. 12.5: Flow of deep learning

extractor and the classifier 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 12.6 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.

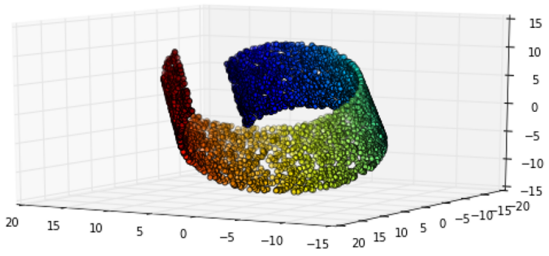


Fig. 12.6: Data manifold – "swiss role" example

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 12.7.

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.

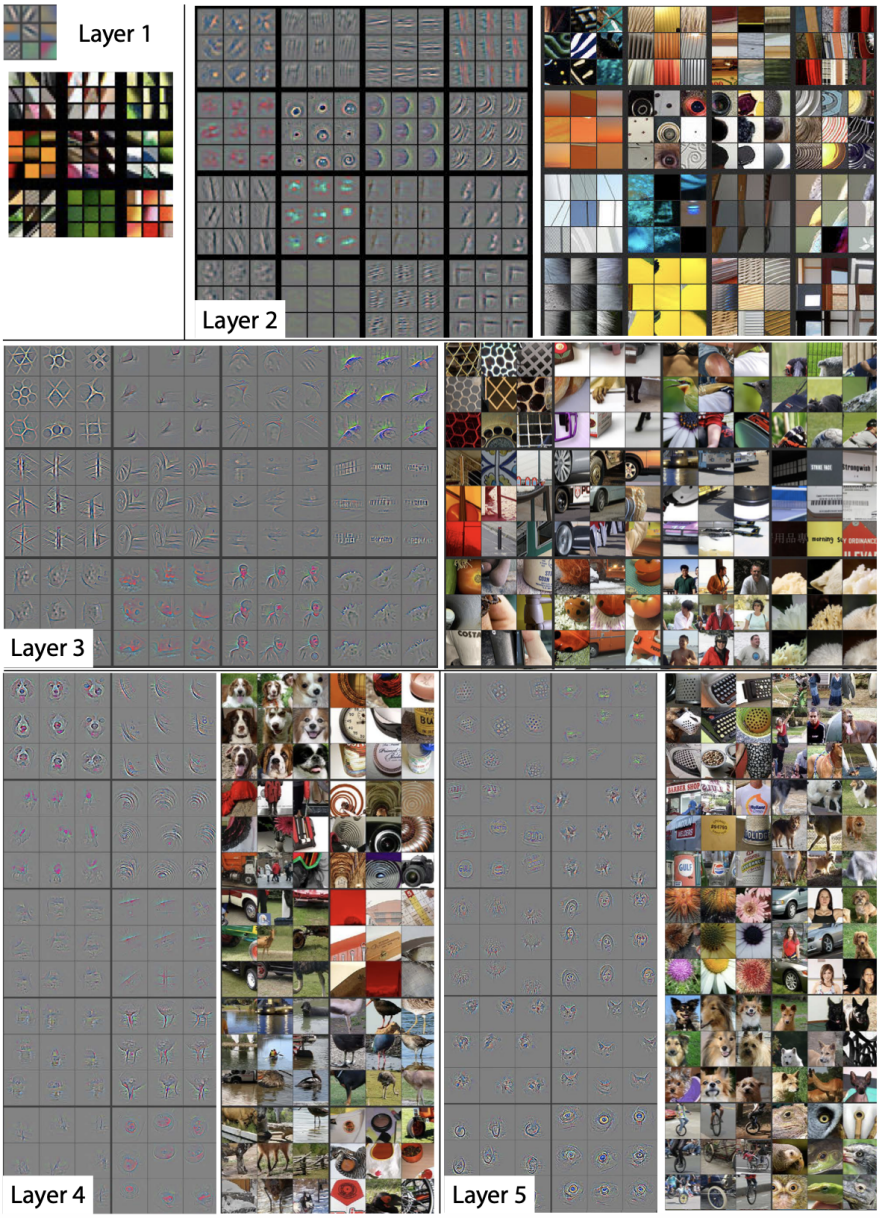


Fig. 12.7: Visualisation of features in hidden layers [98]

12.4 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()
```

Chapter 13
Forward and Backward Computation

In Section 11, 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 11 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 13.1.

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

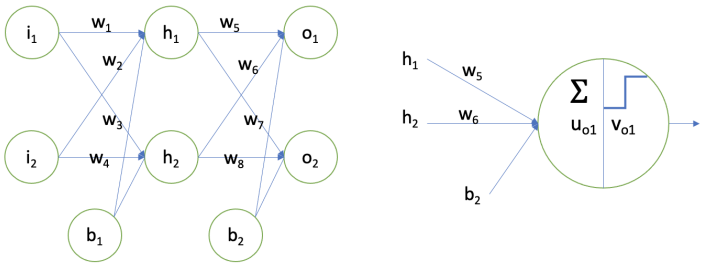


Fig. 13.1: A simple neural network with a hidden layer (Left) and the illustration of a neuron with activation function (Right)

$$\begin{pmatrix} w_1 & w_2 \\ w_3 & w_4 \end{pmatrix} = \begin{pmatrix} 0.15 & 0.20 \\ 0.25 & 0.30 \end{pmatrix}, \quad \begin{pmatrix} w_5 & w_6 \\ w_7 & w_8 \end{pmatrix} = \begin{pmatrix} 0.40 & 0.45 \\ 0.50 & 0.55 \end{pmatrix}, \quad \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} 0.35 \\ 0.60 \end{pmatrix} \tag{13.1}$$

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.

13.1 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 13.1, we can have

$$\begin{pmatrix} u_{h_1} \\ u_{h_2} \end{pmatrix} = \begin{pmatrix} 0.15 & 0.20 \\ 0.25 & 0.30 \end{pmatrix} \times \begin{pmatrix} 0.05 \\ 0.10 \end{pmatrix} + \begin{pmatrix} 0.35 \\ 0.35 \end{pmatrix} = \begin{pmatrix} 0.3775 \\ 0.6425 \end{pmatrix} \tag{13.2}$$

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

$$\begin{pmatrix} v_{h_1} \\ v_{h_2} \end{pmatrix} = \sigma \begin{pmatrix} 0.3775 \\ 0.3925 \end{pmatrix} \approx \begin{pmatrix} 0.5927 \\ 0.5969 \end{pmatrix} \tag{13.3}$$

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

$$\begin{pmatrix} u_{o_1} \\ u_{o_2} \end{pmatrix} = \begin{pmatrix} 0.40 & 0.45 \\ 0.50 & 0.55 \end{pmatrix} \times \begin{pmatrix} 0.5927 \\ 0.5969 \end{pmatrix} + \begin{pmatrix} 0.60 \\ 0.60 \end{pmatrix} = \begin{pmatrix} 1.1057 \\ 1.2247 \end{pmatrix} \tag{13.4}$$

Consider the Sigmoid function, we have

$$\begin{pmatrix} v_{o_1} \\ v_{o_2} \end{pmatrix} = \sigma\left(\begin{pmatrix} 1.1057 \\ 1.2247 \end{pmatrix}\right) \approx \begin{pmatrix} 0.7513 \\ 0.7729 \end{pmatrix} \tag{13.5}$$

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

$$L(\mathbf{x}, y) = \frac{1}{2}(0.7513 - 0.01)^2 + \frac{1}{2}(0.7729 - 0.99)^2 \approx 0.2748 + 0.0236 = 0.2984 \tag{13.6}$$

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.

13.2 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:

$$\frac{\partial L}{\partial w_5} = \frac{\partial L_{o_1}}{\partial v_{o_1}} * \frac{\partial v_{o_1}}{\partial u_{o_1}} * \frac{\partial u_{o_1}}{\partial w_5} \tag{13.7}$$

Figure 13.2 presents an illustrative diagram for the Equation (13.7). Actually, the

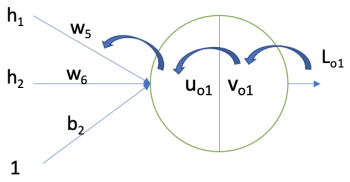


Fig. 13.2: Backward propagation on the output neuron

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

$$\frac{\partial L_{o_1}}{\partial v_{o_1}} = \frac{\partial}{\partial v_{o_1}}\left(\frac{1}{2}(y^{(1)} - v_{o_1})^2\right) = -(y^{(1)} - v_{o_1}) = 0.7513 - 0.01 = 0.74 \tag{13.8}$$

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

$$\frac{\partial v_{o_1}}{\partial u_{o_1}} = \frac{\partial \sigma(u_{o_1})}{\partial u_{o_1}} = \sigma(u_{o_1})(1-\sigma(u_{o_1})) = v_{o_1}(1-v_{o_1}) \approx 0.7513 \times 0.2487 \approx 0.1868 \tag{13.9}$$

and

$$\frac{\partial u_{o_1}}{\partial w_5} = \frac{\partial}{\partial w_5}(w_5 v_{h_1} + w_6 v_{h_2} + b_2) = v_{h_1} \approx 0.5927 \tag{13.10}$$

Therefore, we have

$$\frac{\partial L}{\partial w_5} = \frac{\partial L_{o_1}}{\partial v_{o_1}} * \frac{\partial v_{o_1}}{\partial u_{o_1}} * \frac{\partial u_{o_1}}{\partial w_5} \approx 0.74 \times 0.1868 \times 0.5927 = 0.0819 \tag{13.11}$$

Weights of hidden neurons

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

$$\begin{aligned}
\frac{\partial L}{\partial w_1} &= \frac{\partial L}{\partial v_{h_1}} * \frac{\partial v_{h_1}}{\partial u_{h_1}} * \frac{\partial u_{h_1}}{\partial w_1} \\
&= (\frac{\partial L_{o_1}}{\partial v_{h_1}} + \frac{\partial L_{o_2}}{\partial v_{h_1}}) * \frac{\partial v_{h_1}}{\partial u_{h_1}} * \frac{\partial u_{h_1}}{\partial w_1} \\
&= (\frac{\partial L_{o_1}}{\partial u_{o_1}} \frac{\partial u_{o_1}}{\partial u_{h_1}} + \frac{\partial L_{o_2}}{\partial u_{o_2}} \frac{\partial u_{o_2}}{\partial u_{h_1}}) * \frac{\partial v_{h_1}}{\partial u_{h_1}} * \frac{\partial u_{h_1}}{\partial w_1} \\
&= (\frac{\partial L_{o_1}}{\partial u_{o_1}} w_5 + \frac{\partial L_{o_2}}{\partial u_{o_2}} w_7) * \frac{\partial v_{h_1}}{\partial u_{h_1}} * \frac{\partial u_{h_1}}{\partial w_1}
\end{aligned} \tag{13.12}$$

Note that, all the components of Equation (13.12) can now be computed as the method we used for the output layer. Figure 13.3 presents an illustration of backward propagation as in Equation (13.12).

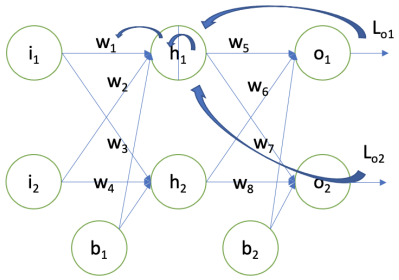


Fig. 13.3: Backward propagation on the hidden neuron

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.

13.3 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 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:

$$\begin{aligned} &\min_f L(f, D) = \frac{1}{n} \sum_{i=1}^{n} L(f, \mathbf{x}_i, y_i) \\ &\text{subject to} \quad f \in \mathcal{H} \end{aligned} \tag{13.13}$$

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

$$\begin{aligned} \min_W \ & L(W, D) = \frac{1}{n} \sum_{i=1}^{n} L(W, \mathbf{x}_i, y_i) \\ \text{subject to} \ & W \in \mathbb{R}^{|\mathcal{W}|} \end{aligned} \tag{13.14}$$

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

$$\begin{aligned} \min_W \ & L(W, D) = \frac{1}{n} \sum_{i=1}^{n} L(W, \mathbf{x}_i, y_i) \\ \text{subject to} \ & W \in \mathbb{R}^{|\mathcal{W}|} \\ & ||W||_2^2 \leq r^2 \end{aligned} \tag{13.15}$$

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.,

$$\min_W L(W, D) = \frac{1}{n} \sum_{i=1}^{n} L(W, \mathbf{x}_i, y_i) + \lambda ||W||_2^2 \tag{13.16}$$

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:

$$\min_W L(W, D) = \frac{1}{n} \sum_{i=1}^{n} L(W, \mathbf{x}_i, y_i) + \lambda(||W||_2^2 - r) \tag{13.17}$$

13.4 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'))
```

Chapter 14
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 [44]. 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.

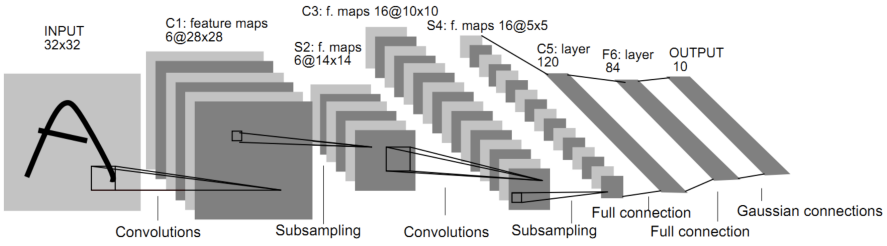


Fig. 14.1: Architecture of LeNet-5 [44], a convolutional neural network for digits recognition.

As shown in Figure 14.1, 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.

14.1 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,

$$\mathbf{u}_i = \mathbf{W}_i \mathbf{v}_{i-1} + \mathbf{b}_i \quad \text{and} \quad \mathbf{v}_i = \sigma_i(\mathbf{u}_i) \tag{14.1}$$

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

14.1.0.1 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 14.2, 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

$$u_{21} = \mathbf{W}_{21} \times \mathbf{v}_1 + b_2 = w_{21,11} v_{11} + ... + w_{21,16} v_{16} + b_2 \tag{14.2}$$

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.,

$$v_{21} = \alpha(u_{21}) \tag{14.3}$$

We will introduce activation functions α later.

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.

14.1.0.2 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 14.3

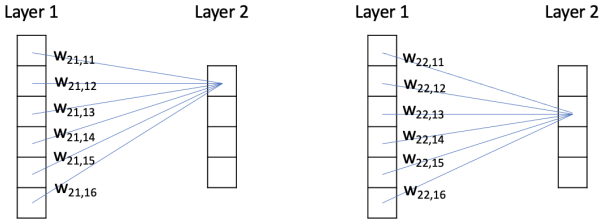


Fig. 14.2: Illustration of fully-connected layer

with the red dashed lines, and (2) computing an element $z_{i,j}$ with element-wise multiplication, as illustrated in Figure 14.3 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:

$$z_{1,1} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k,1+l} \times f_{1+k,1+l} \tag{14.4}$$

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

$$z_{1,2} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k,1+l+stride} \times f_{1+k,1+l} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k,1+l+1} \times f_{1+k,1+l} \tag{14.5}$$

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

$$z_{2,1} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k+stride,1+l} \times f_{1+k,1+l} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k+1,1+l} \times f_{1+k,1+l} \tag{14.6}$$

Figure 14.3 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.,

$$z_{1,2} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k,1+l+stride} \times f_{1+k,1+l} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k,1+l+2} \times f_{1+k,1+l} \tag{14.7}$$

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

$$z_{2,1} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k+stride,1+l} \times f_{1+k,1+l} = \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} x_{1+k+2,1+l} \times f_{1+k,1+l} \tag{14.8}$$

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.

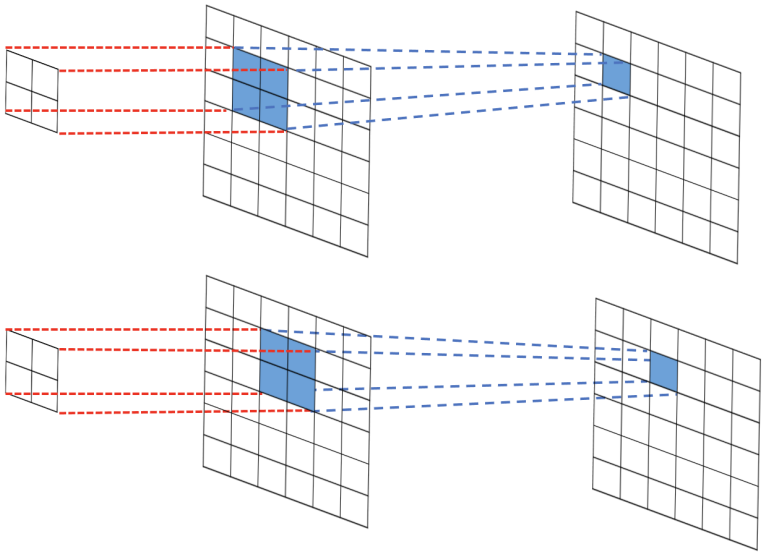


Fig. 14.3: Illustration of convolutional layer

We can see that, $\mathbf{z}$ is a $t \times t$ matrix such that

$$t = \frac{n - m}{stride} + 1 \tag{14.9}$$

For example, if $n = 4$ and $m = 2$, then $t = 3$ when $stride = 1$ and $t = 2$ when $stride = 2$.

14.1.0.3 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 14.4.

We can see that, if we pad $\mathbf{x}$ with a border of u zero valued pixels, $\mathbf{z}$ is a $t \times t$ matrix such that

$$t = \frac{n - m + 2 * u}{stride} + 1 \tag{14.10}$$

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

Fig. 14.4: Zero-padding: pad the input with 0-cells around it.

In Figure 14.4, $u = 1$. Therefore, if $n = 4$ and $m = 2$ and $u = 1$, then $t = 5$ when $stride = 1$ and $t = 3$ when $stride = 2$.

14.1.0.4 Pooling Layer

A pooling layer is to reduce information in a matrix by collapsing elements with operations. Pooling layer is frequently used in convolutional neural networks with the purpose to progressively reduce the spatial size of the representation to reduce the amount of features and the computational complexity of the network. Assume that, as shown in Figure 14.5, we have a 4×4 matrix. An application of a 2×2 max-pooling filter, under the condition that $stride = 2$, will get a 2×2 matrix by collapsing every 2×2 block with the max operation.

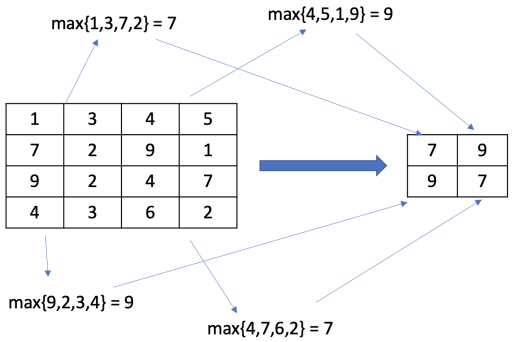


Fig. 14.5: Max-pooling. An application of 2×2 filter, $stride = 2$, on a 4×4 matrix.

In addition to max-pooling, there are other pooling layers, such as average-pooling layer, which replaces the max operation with the average operation.

14.2 Activation Functions

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

ReLU

$$ReLU(x) = max(0, x) \tag{14.11}$$

Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{14.12}$$

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

Softmax

$$\delta(\mathbf{v}) = (\frac{e^{v_1}}{\sum_{i=1}^{|\mathbf{v}|} e^{v_i}}, ..., \frac{e^{v_{|\mathbf{v}|}}}{\sum_{i=1}^{|\mathbf{v}|} e^{v_i}}) \tag{14.13}$$

14.3 Data Preprocessing

A suitable pre-processing on the training data can have a significant impact on the performance of the resulting model. In the following, we introduce a few different data pre-processing methods. Whether or not a specific pre-processing method should be applied is problem specific, depending on the dataset and the machine learning task.

Mean normalization

removes the mean from each data sample, i.e.,

$$\mathbf{x}' = \mathbf{x} - \bar{\mathbf{x}} \tag{14.14}$$

where $\bar{\mathbf{x}} = \frac{1}{|D|} \sum_{\mathbf{x} \in D} \mathbf{x}$ is the mean of the dataset D.

Standardization or normalization

requires, on top of the mean normalisation, all features to be on the same scale, i.e., every sample $\mathbf{x}$ is converted into

$$\mathbf{x}' = \frac{\mathbf{x} - \bar{\mathbf{x}}}{\sigma_D} \tag{14.15}$$

where σ_D is the standard deviation of the dataset D.

Whitening

requires that the covariance matrix of the converted dataset is the identity matrix – 1 in the diagonal and 0 for the other cells. It first applies the mean normalisation on the dataset D to get D', and then apply a whitening matrix W on every sample, i.e., let $\mathbf{x}'' = \mathbf{W}\mathbf{x}'$, such that $\mathbf{W}\mathbf{W}^T = \Sigma^{-1}$ and Σ is the non-singular covariance matrix of D'.

Depending on what $\mathbf{W}$ is, we have Mahalanobis or ZCA whitening ($\mathbf{W} = \Sigma^{-1/2}$), Cholesky whitening ($\mathbf{W} = \mathbf{L}^T$ for $\mathbf{L}$ the Cholesky decomposition of Σ^{-1}), or PCA whitening ($\mathbf{W}$ is the eigen-system of Σ^{-1}).

14.4 Practicals

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)

            #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))

        #save model
        torch.save(model.state_dict(), os.path.join(args.model_dir, 'final_model.pt'))

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

Chapter 15
Regularisation Techniques

This section introduces several regularisation techniques, which aim to introduce inductive bias to the learning process.

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.

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 (7.3), the linear regression is to minimise

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{w}^T\mathbf{x}^{(i)} - y^{(i)})^2 \tag{15.1}$$

and, as suggested in Equation (12.2), the convolutional neural network is to minimise

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(f_{\mathbf{w}}(\mathbf{x}^{(i)}) - y^{(i)})^2 \tag{15.2}$$

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.

15.1 Ridge Regularisation

For ridge regularisation, the loss function is updated by having a penalty term, i.e.,

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(f_{\mathbf{w}}(\mathbf{x}^{(i)}) - y^{(i)})^2 + \lambda \sum_{w\in\mathbf{W}} w^2 \tag{15.3}$$

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.

15.2 Lasso Regularisation

For lasso (least absolute shrinkage and selection operator) regularisation, the loss function is updated by having a penalty term, i.e.,

$$\hat{L}(f_{\mathbf{w}}) = \frac{1}{m}\sum_{i=1}^{m}(f_{\mathbf{w}}(\mathbf{x}^{(i)}) - y^{(i)})^2 + \lambda \sum_{w\in\mathbf{W}} |w| \tag{15.4}$$

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.

15.3 Dropout

Dropout [76] 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.

15.4 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).

15.5 Batch-Normalisation

a normalization step that fixes the means and variances of each layer's inputs

Chapter 16
Robustness and Adversarial Attack

As explained in Section 3.2, an adversarial example is an input that is close enough to, but with a different predicted label with, a correctly-predicted input. In most cases, the search for an adversarial example is formalised as an optimisation problem, in a form either the same as or similar with Equation (3.8).

16.1 Limited-Memory BFGS Algorithm

[85] noticed the existence of adversarial examples, and described them as 'blind spots' in DNNs. They found that adversarial images usually appear in the neighbourhood of correctly-classified examples, which can fool the DNNs although they are human-visually similar to the natural ones. It also empirically observes that random sampling in the neighbouring area (see the template solution we provided in Section C) is not efficient to generate such examples due to the sparsity of adversarial images in the high-dimensional space. Thus, they proposed an optimization solution to efficiently search the adversarial examples. Formally, assume we have a classifier $f: \mathbb{R}^{s_1} \rightarrow \{1 \dots s_K\}$ that maps inputs to one of s_K labels, and $\mathbf{x} \in \mathbb{R}^{s_1}$ is an input, $t \in \{1 \dots s_K\}$ is a target label such that $t \neq \arg\max_l f_l(\mathbf{x})$. Then the adversarial perturbation $\mathbf{r}$ can be solved by

$$\begin{aligned} & \min ||\mathbf{r}||_2 \\ s.t. \quad & \arg\max_l f_l(\mathbf{x}+\mathbf{r}) = t \\ & \mathbf{x}+\mathbf{r} \in \mathbb{R}^{s_1} \end{aligned} \tag{16.1}$$

Since the exact computation is hard, an approximate algorithm based on the limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm (L-BFGS) is used instead. Furthermore, [85] observed that adversarial perturbations are able to transfer among different model structures and training sets, i.e., an adversarial image that aims to fool one DNN classifier also potentially deceives another neural network with different architectures or training datasets.

16.2 Fast Gradient Sign Method

Fast Gradient Sign Method [19] 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, [19] proposes an efficient linear approach to generate adversarial images. Let θ represents the model parameters, $\mathbf{x}$, y denote the input and the label and $J(\theta, \mathbf{x}, y)$ is the loss function. We can calculate adversarial perturbation $\mathbf{r}$ by

$$\mathbf{r} = \epsilon \text{ sign } (\nabla_{\mathbf{x}} J(\theta, \mathbf{x}, y)) \tag{16.2}$$

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 [41].

Algorithm 9: ***PGDAttack***$(f, \mathbf{x}, y, ||\cdot||, d, n, \epsilon)$, where f is the original model that the user wants to attack, $\mathbf{x}$ is the sample to be attacked, y is the true label of $\mathbf{x}$, $||\cdot||$ is the norm distance, d is the radius, n is the number of iterations, and $\epsilon > 0$ is an attack magnitude.

$i \leftarrow 0$
$\mathbf{x}^i$ is randomised such that $||\mathbf{x} - \mathbf{x}^i|| < d$
repeat
  $\mathbf{x}^{i+1} \leftarrow Clip_{\mathbf{x},||\cdot||,d}(\mathbf{x}^i + \epsilon \text{sign}(\nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}^i, y)))$
  $i \leftarrow i + 1$
until $i = n$;
return $\mathbf{x}^n$, as an adversarial example.

Algorithm 9 presents a pseudo code for PGD attack, where $Clip_{\mathbf{x},||\cdot||,d}(\mathbf{x}')$ is a clipping operation that projects any input $\mathbf{x}'$ into the norm ball centered at $\mathbf{x}$ with radius d.

16.3 Jacobian Saliency Map based Attack (JSMA)

[62] present a L_0-norm based adversarial attacking method by exploring the *forward derivative* of a neural network. Specifically it utilizes the Jacobian matrix of a DNN's logit output w.r.t. its input to identify those most sensitive pixels which then are perturbed to fool the neural network model effectively. Let $c \in$ denote a target class and $\mathbf{x} \in [0, 1]^{s_1}$ represent an input image. JSMA will assign each pixel in $\mathbf{x}$ a salient weight based on the Jacobian matrix. Each salient value basically

quantifies the sensitivity of the pixel to the predicted probability of class c. To generate the adversarial perturbation, the pixel with the highest salient weight is firstly perturbed by a *maximum distortion parameter* $\tau > 0$. If the perturbation leads to a mis-classification, then JSMA attack terminates. Otherwise, the algorithm will continue until a mis-classification is achieved. When a maximum L_0-norm distortion $d > 0$ is reached, the algorithm also terminates. This algorithm is primarily to produce adversarial images that are optimized under the L_0-norm distance. JMSA is generally slower than FGSM due to the computation of the Jacobian matrix.

16.4 DeepFool

In DeepFool, [55] introduces an iterative approach to generate adversarial images on any L_p norm distance, for $p \in [1, \infty)$. In this work, the authors first show how to search adversarial images for an affine binary classifier, i.e., $g(\mathbf{x}) = \text{sign}(\mathbf{w}^T \cdot \mathbf{x} + \mathbf{b})$. Given an input image $\mathbf{x}_0$, DeepFool is able to produce an optimal adversarial image by projecting $\mathbf{x}_0$ orthogonally onto the hyper-plane $\mathcal{F} = \{\mathbf{x} | \mathbf{w}^T \cdot \mathbf{x} + \mathbf{b} = 0\}$. Then this approach is generalized for a multi-class classifier: $\mathbf{W} \in \mathbb{R}^{m \times k}$ and $\mathbf{b} \in \mathbb{R}^k$. Let $\mathbf{W}_i$ and b_i be the i-th component of $\mathbf{W}$ and $\mathbf{b}$, respectively. We have

$$g(\mathbf{x}) = \underset{i \in \{1 \ldots k\}}{\text{argmax}}\ g_i(\mathbf{x}) \text{ where } g_i(\mathbf{x}) = \mathbf{W}_i^T \mathbf{x} + b_i$$

For this case, the inupt $\mathbf{x}_0$ is projected to the nearest face of the hyper-polyhedron P to produce the optimal adversarial image, namely,

$$P(\mathbf{x}_0) = \bigcap_{i=1}^{k} \{\mathbf{x} | g_{k_0}(\mathbf{x}) \geq g_i(\mathbf{x})\}$$

where $k_0 = g(\mathbf{x}_0)$. We can see that P is the set of the inputs with the same label as $\mathbf{x}_0$. In order to generalize DeepFool to neural networks, the authors introduce an iterative approach, namely, the adversarial perturbation is updated at each iteration by approximately linearizing the neural network and then perform the projection. Please note that, DeepFool is a heuristic algorithm for a neural network classifier that provides no guarantee to find the adversarial image with the minimum distortion, but in practise it is an very effective attacking method.

16.5 Carlini & Wagner Attack

C&W Attack [9] is an optimisation based adversarial attack method which formulates finding an adversarial example as image distance minimisation problem such as L_0, L_2 and L_∞-norm. Formally, it formalizes the adversarial attack as an optimisation problem to minimise

$$\mathcal{L}(\mathbf{r}) = ||\mathbf{r}||_p + c \cdot F(\mathbf{x} + \mathbf{r}), \tag{16.3}$$

where $\mathbf{x} + \mathbf{r}$ is a valid input, and F represents a surrogate function such as $\mathbf{x} + \mathbf{r}$ is able to fool the neural network when it is negative. The authors directly adopt the optimizer Adam [38] to slove this optimization problem. It is worthy to mention that C&W attack can work on three distance including L_2, L_0 and L_∞ norms. A smart trick in C&W Attack lies on that it introduces a new optimisation variable to avoid box constraint (image pixel need to within $[0, 1]$). C&W attack is shown to be a very strong attack which is more effective than JSMA [62], FGSM [19] and DeepFool [55]. It is able to find an adversarial example that has a significant smaller distortion distance, especially on L_2-norm metric.

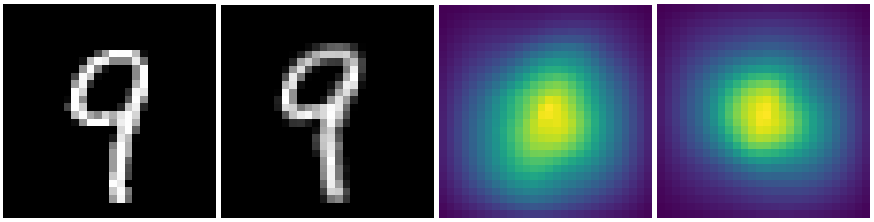
Fig. 16.1: Rotation-Translation: Original (L) 'automobile', adversarial (R) 'dog' from [18]. *The original image of an 'automobile' from the CIFAR-10 dataset is rotated (by at most* 30°*) and translated (by at most 3 pixels) results in an image that state-of-art classifier ResNet [24] classifies as 'dog'.*

16.6 Adversarial Attacks by Natural Transformations

Additional to the above approaches which perform adversarial attack on pixel-level, research has been done on crafting adversarial examples by applying natural transformations.

16.6.0.1 Rotation and Translation

[18] argues that most existing adversarial attacking techniques generate adversarial images which appear to be human-crafted and less likely to be 'natural'. It shows that DNNs are also vulnerable to some image transformations which are likely occurring in a natural setting. For example, translating or/and rotating an input image could significantly degrade the performance of a neural network classifier. Figure 16.1 gives a few examples. Technically, given an allowed range of translation and rotation



(a) Original: '9' (b) Adversarial: '8' (c) x-dimension (d) y-dimension

Fig. 16.2: Applying spatial transformation to MNIST image of a '9' [94]. *the Image (a) on the left is the original MNIST example image of a '9', and image (b) is the spatially transformed adversarial version that a simple convolutional network [60] labels as '8'. Notice how minor the difference between the two images is - the '9' digit has been very slightly 'bent' - but is sufficient for miss-classification. The flow-field that defines the spatial transformation is visualised in Image (c) (x-dimension) and Image (d) (y-dimension). The brighter areas indicate where the transformation is most intense - leftwards in the x-dimension and upwards in the y-dimension.*

such as ±3 *pixels* × ± 30°, [18] aims to find the minimum rotation and translation to cause a misclassification. To achieve such an purpose, in this paper, several ideas are explored including

- a first-order iterative method using the gradient of the DNN's loss function,
- performing an exhaustive search by descretizing the parameter space,
- a worst-of-k method by randomly sampling k possible parameter values and choosing the value that cause the DNN to perform the worst.

16.6.0.2 Spatially Transformed Adversarial Examples

[94] introduces to produce uncontrived adversarial images via mortifying the pixel's location using spatial transformations instead of directly changing its pixel value. The authors use the flow field to control the spatial transformations, which essentially quantifies the location displacement of a pixel to its new position. Figure 16.2 gives a few examples. Using a bi-linear interpolation approach the generated adversarial example is differentiable w.r.t. the flow field, which then can be formulated as an optimisation problem to calculate the adversarial flow field. Technically, [94] introduce a distance measure $L_{flow}(\cdot)$ (rather than the usual L_p distance) to capture the local geometric distortion. Similar to C&W attack [9], the flow field is obtained by solving an optimisation problem in which the loss function is defined to balance between the L_{flow} loss and adversarial loss. Through human study, [94] demonstrate that adversarial examples based on such spatial transformation are more similar to original images based on human perception, comparing to those adversarial examples from L_p-norm based attacks such as FGSM [19] and C&W [9].

16.6.0.3 Towards Practical Verification of Machine Learning: The Case of Computer Vision Systems (VeriVis)

[64] introduce a verificaiton framework, called VeriVis, to measure the robustness of DNNs on a set of 12 practical image transformations including reflection, translation, scale, shear, rotation, occlusion brightness, contrast, dilation, erosion, average smoothing, and median smoothing. Every transformation is controlled by a key parameter with a *polynomial-sized* domain. Those transformations are exhaustively operated on a set of input images. Then the robustness of a neural network model can be measured. VeriVis is applied to evaluate several state-of-the-art classification models, which empirically reveals that all classifiers show a significant number of safety violations.

16.7 Input-Agnostic Adversarial Attacks

A key characteristic of the above attacks lies on that an adversarial example is generated with respect to a specific input, and therefore cannot be applied to the other input. Thus some researchers show more interests in *input-agnostic* adversarial perturbations.

16.7.0.1 Universal Adversarial Perturbations

The first method on input-agnostic adversarial perturbation was proposed by [54], called *universal* adversarial perturbations (UAP), since UAP is able to fool a neural network on *any* input images with high probability. Let $\mathbf{r}$ the the current perturbation. UAP iteratively goes through a set D of inputs sampled from the input distribution $\mathcal{D}$. At the iteration for $x_i \in D$ it updates the perturbation $\mathbf{r}$ as follows. First, it finds the minimal $\Delta\mathbf{r}_i$ w.r.t. L_2-norm distance so that $x_i + \mathbf{r} + \Delta\mathbf{r}_i$ is incorrectly classified by neural network $\mathtt{N}$. Then, it projects $\mathbf{r} + \Delta\mathbf{r}_i$ back into L_p-norm ball with a radius d to enable that the generated perturbation is sufficiently small, i.e., let

$$\begin{aligned} \mathbf{r} = \arg\min_{\mathbf{r}'} & ||\mathbf{r}' - (\mathbf{r} + \Delta\mathbf{r}_i)||_2 \\ \text{s.t. } & ||\mathbf{r}'||_p \leq d. \end{aligned} \tag{16.4}$$

The algorithm will proceed until the empirical error of the sample set is sufficiently large, namely, no less than $1 - \delta$ for a pre-specified threshold δ.

16.7.0.2 Generative Adversarial Perturbations

By training a universal adversarial network (UAN), [22] generalizes the C&W attack in [9] to generates input-agnostic adversarial perturbations. Assume that we have a

maximum perturbation distance d and an L_p norm, a UAN $\mathcal{U}_\theta$ randomly samples an input z from a normal distribution and generates a raw perturbation $\mathbf{r}$. Then it is scaled through $\mathbf{w} \in [0, \frac{d}{||\mathbf{r}||_p}]$ to have $\mathbf{r}' = \mathbf{w} \cdot \mathbf{r}$. Then, the new input $\mathbf{x} + \mathbf{r}'$ needs to be checked with DNN `N` to see if it is an adverarial example. The parameters θ is optimized by adopting a gradient descent method, similar to the one used by C&W attack [9].

Later on, [67] introduces a similar approach in[22] to produce input-agnostic adversarial images. It first samples a random noise to input to UAN, and the output then is resized to meet an L_p constraint which is further added to input data, clipped, and then is used to train a classifier. This method is different to [22] in two aspects. Firstly, it explores two UAN architectures including U-Net [69] and ResNet Generator [36], and find ResNet Generator works better in the majority of the cases. Secondly, this work also trained a UAN by adopting several different classifiers, thus the proposed UAP can explicitly fool multiple classifiers, which is obtained by the below loss function:

$$l_{multi-fool}(\lambda) = \lambda_1 \cdot l_{fool_1} + \cdots + \lambda_m \cdot l_{fool_m} \tag{16.5}$$

where l_{fool_i} denotes the adversarial loss for classifier i, m is the number of the classifiers, and λ_i is the weight to indicate the difficulty of fooling classifier i.

16.8 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()
```

Chapter 17
Poisoning Attack

We consider both heuristic approaches, which generates poisoning samples with various heuristics from a set of base samples, and an alternating optimisation approach, which not only finds poisoning samples with heuristics but also continuously refine the obtained poisoning samples.

17.1 Heuristic Approaches

For heuristic approaches, we assume there is a set $\mathbf{X}_b$ of samples. The objective is to compute another set $\mathbf{X}_p$ of poisoning samples such that $\mathbf{X}_p$ and $\mathbf{X}_b$ are close and the target input $\mathbf{x}_{adv}$ is classified as y_{adv}. Let g be a feature extraction function that maps every sample into a vector of features. Practically, g can be a neural network or part of a neural network, as discussed in Section 12.3.

Feature Collision

is to synthesise a poisoning sample $\mathbf{x}_p^{(i)} \in \mathbf{X}_p$ from each base sample $\mathbf{x}_b^{(i)} \in \mathbf{X}_b$ by adding perturbation. Formally,

$$\mathbf{x}_p^{(i)} = \arg\min_{\mathbf{x}} ||g(\mathbf{x}) - g(\mathbf{x}_{adv})||_2^2 + \beta||\mathbf{x} - \mathbf{x}_b^{(i)}||_2^2 \tag{17.1}$$

Intuitively, it requires not only the similarity between $\mathbf{x}_p^{(i)}$ and $\mathbf{x}_b^{(i)}$ but also the similarity of feature representations between $\mathbf{x}_p^{(i)}$ and the target sample $\mathbf{x}_{adv}$.

Convex Polytope

Instead of requiring the alignment of each poisoning sample's feature representation with that of target sample, it is also useful to synthesise the entire set $\mathbf{X}_p$ in a single round by requiring that $g(\mathbf{x}_{adv})$ aligns with a convex combination of $g(\mathbf{x}_p^{(i)})$. Formally,

$$\begin{aligned}\mathbf{X}_p = \underset{c_i, \mathbf{x}_p^{(i)}, i=1..|X_p|}{\arg\min} \quad & ||g(\mathbf{x}_{adv}) - \sum_{i=1}^{|\mathbf{X}_p|} c_j g(\mathbf{x}_p^{(i)})||_2^2 \\ s.t. \ & \sum_{i=1}^{|\mathbf{X}_p|} c_i = 1 \\ & \forall 1 \leq i \leq |\mathbf{X}_p| : c_i > 0 \\ & \forall 1 \leq i \leq |\mathbf{X}_p| : ||\mathbf{x}_p^{(i)} - \mathbf{x}_b^{(i)}||_\infty \leq \epsilon\end{aligned} \tag{17.2}$$

In the following, we consider two heuristic approaches for backdoor attack.

Hidden Trigger Backdoor

is similar with Feature Collision, except that the alignment of feature representation is not between $\mathbf{x}_p^{(i)}$ and the target sample $\mathbf{x}_{adv}$, but between $\mathbf{x}_p^{(i)}$ and a patched training image from the target class y_{adv}. Formally,

$$\mathbf{x}_p^{(i)} = \arg\min_{\mathbf{x}} ||g(\mathbf{x}) - g(\tilde{\mathbf{x}}_{adv}^i)||_2^2 + \beta ||\mathbf{x} - \mathbf{x}_b^{(i)}||_2^2 \tag{17.3}$$

where $\tilde{\mathbf{x}}_{adv}^i$ is patched training image from the target class y_{adv}.

Clean Label Backdoor

starts by generating an adversarial example $\tilde{\mathbf{x}}_{adv}^i$ for each base sample, and then constructing $\mathbf{x}_{adv}^i$ by adding a patch to $\tilde{\mathbf{x}}_{adv}^i$.

17.2 An Alternating Optimisation Approach

As in Section 3.3, the data poisoning attack is a bi-level optimisation problem, which cannot be solved in a tractable way when the objective functions $\mathcal{L}_{adv}$ and $\mathcal{L}_{train}$ are non-convex. Algorithm 10 presents an algorithm that solves the problem in an alternating optimisation way.

Algorithm 10: ***PoisoningAttack***$(\mathbf{x}_{adv}, y_{adv}, \mathbf{X}_p, \mathbf{X}, \mathcal{L}_{adv}, \mathcal{L}_{train})$, where $\mathbf{x}_{adv}$ is the target input, y_{adv} is the target label, $\mathbf{X}_p$ is a set of initial poisoning samples, $\mathbf{X}$ is the training dataset, $\mathcal{L}_{adv}(\mathbf{x}_{adv}, y_{adv}; \mathbf{W})$ is a function to measure the accuracy of predicting $\mathbf{x}_{adv}$ as y_{adv} with a neural network whose parameters are $\mathbf{W}$, and $\mathcal{L}_{train}(\mathbf{X}, \mathbf{y})$ is the standard training loss function.

1 i $\leftarrow 0$
2 $\mathbf{X}_p^i \leftarrow \mathbf{X}_p$
3 $\mathbf{W}^i \leftarrow \arg\min_{\mathbf{W}} \mathcal{L}_{train}(\mathbf{X} \cup \mathbf{X}_p^i, \mathbf{y}; \mathbf{W})$, where $\mathbf{y}$ includes labels for both $\mathbf{X}$ and $\mathbf{X}_p^i$
4 $l^i \leftarrow \mathcal{L}_{adv}(\mathbf{x}_{adv}, y_{adv}; \mathbf{W}^i)$
5 **repeat**
6 $\quad \mathbf{X}_p^{i+1} \leftarrow \emptyset$
7 $\quad$ **for** $c = 1 \ldots |X_p|$ **do**
8 $\quad\quad \mathbf{x}_c^{i+1} \leftarrow line\_search(\mathbf{x}_c^i, \nabla_{\mathbf{x}_c} \mathcal{L}_{adv}(\mathbf{x}_{adv}, y_{adv}; \mathbf{W}^i))$
9 $\quad\quad \mathbf{X}_p^{i+1} \leftarrow \mathbf{X}_p^{i+1} \cup \{\mathbf{x}_c^{i+1}\}$
10 $\quad$ **end**
11 $\quad \mathbf{W}^{i+1} \leftarrow \arg\min_{\mathbf{W}} \mathcal{L}_{train}(\mathbf{X} \cup \mathbf{X}_p^{i+1}, \mathbf{y}; \mathbf{W}^i)$
12 $\quad l^{i+1} \leftarrow \mathcal{L}_{adv}(\mathbf{x}_{adv}, y_{adv}; \mathbf{W}^{i+1})$
13 $\quad i \leftarrow i + 1$
14 **until** $l^i - l^{i-1} < \epsilon$;
15 **return** the final $\mathbf{X}_p^i$

It repeats by first conducting a line search over the gradient $\nabla_{\mathbf{x}_c} \mathcal{L}_{adv}(\mathbf{x}_{adv}, y_{adv}; \mathbf{W}^i)$ for all candidate samples in $\mathbf{X}_p$ (Line 7-10), and then conducting a standard training over the updated $\mathbf{X} \cup \mathbf{X}_p$ (Line 11). The algorithm terminates when the loss over the target input converges with respect to ϵ (Line 14). Finally, it returns the set of updated poisoning samples (Line 15).

According to the above discussion, the key difficulty is on the computation of

$$\nabla_{\mathbf{x}_c} \mathcal{L}_{adv}(\mathbf{x}_{adv}, y_{adv}; \mathbf{W}^i) \tag{17.4}$$

which depends not only on the input $\mathbf{x}_c$ but also on the weight $\mathbf{W}^i$. Therefore, we can apply the chain rule and get

$$\nabla_{\mathbf{x}_c} \mathcal{L}_{adv} = \nabla_{\mathbf{x}_c} \mathbf{W}^T \cdot \nabla_{\mathbf{W}} \mathcal{L}_{adv} \tag{17.5}$$

where we have the weight $\mathbf{W}$ depends on $\mathbf{x}_c$.

In the following, we explain how to compute $\nabla_{\mathbf{x}_c} \mathbf{W}^T$. To do so, we require that $\nabla_{\mathbf{W}} \mathcal{L}_{train}(\mathbf{X} \cup \mathbf{X}_p, \mathbf{y}; \mathbf{W}) = 0$, according to the Karush-Kuhn-Tucker (KKT) equilibrium conditions, and further such conditions to remain valid while updating $\mathbf{x}_c$, i.e.,

$$\nabla_{\mathbf{x}_c} (\nabla_{\mathbf{W}} \mathcal{L}_{train}(\mathbf{X} \cup \mathbf{X}_p, \mathbf{y}; \mathbf{W})) = 0 \tag{17.6}$$

The application of chain rule on the above equation, we have

$$\nabla_{\mathbf{x}_c} \nabla_{\mathbf{W}} \mathcal{L}_{train} + \nabla_{\mathbf{x}_c} \mathbf{W}^T \cdot \nabla_{\mathbf{W}}^2 \mathcal{L}_{train} = 0 \tag{17.7}$$

Therefore, we have

$$\nabla_{\mathbf{x}_c} \mathbf{W}^T = -\nabla_{\mathbf{x}_c} \nabla_{\mathbf{W}} \mathcal{L}_{train} (\nabla^2_{\mathbf{W}} \mathcal{L}_{train})^{-1} \tag{17.8}$$

Chapter 18
Model Stealing

As described in Section 3.4, model stealing is to construct another model $f_{surrogate}$ that is functionally equivalent to the original model f_{victim}. In this section, we suggest a simple, iterative algorithm that learns the model $f_{surrogate}$ gradually until convergence. The algorithm is presented in Algorithm 11, where $\mathcal{L}_{CE}$ is the cross entropy loss.

Algorithm 11: ***ModelStealingAttack***$(f_{victim}, \epsilon, n)$, where f_{victim} is the original model that the user can access/query, $\epsilon > 0$ is a threshold that will be used to determine the convergence, and n is the number of samples in the synthesised dataset.

1 randomly sample a dataset $\mathbf{X}_{syn}$
2 $f_{surrogate} \leftarrow \arg\min_{f} \sum_{\mathbf{x} \in \mathbf{X}_{syn}} \mathcal{L}_{CE}(f(\mathbf{x}), f_{victim}(\mathbf{x}))$
3 **while** $\frac{1}{|X_{syn}|} \sum_{x \in X_{syn}} \mathcal{L}_{CE}(f_{surrogate}(x), f_{victim}(x)) \geq \epsilon$ **do**
4 $i \leftarrow 0$
5 $\mathbf{X}_{syn} = \emptyset$
6 **repeat**
7 sample a vector $\mathbf{y}$ as the output prediction of a candidate input
8 $\mathbf{x} \leftarrow \arg\min_{\mathbf{x}} \mathcal{L}_{CE}(f_{surrogate}(\mathbf{x}), \mathbf{y})$
9 $\mathbf{X}_{syn} \leftarrow \mathbf{X}_{syn} \cup \{\mathbf{x}\}$
10 $i \leftarrow i + 1$
11 **until** i>n;
12 $f_{surrogate} \leftarrow \arg\min_{f} \sum_{\mathbf{x} \in \mathbf{X}_{syn}} \mathcal{L}_{CE}(f(\mathbf{x}), f_{victim}(\mathbf{x}))$
13 **end**
14 **return** $f_{surrogate}$

The algorithm proceeds by first randomly selecting a synthetic dataset $\mathbf{X}_{syn}$ (Line 1) to train a surrogate model $f_{surrogate}$ (Line 2). Then, these two terms are

continuously updated by querying the victim model f_{victim}, until the difference between $f_{surrogate}$ and f_{victim} is smaller than the threshold ϵ (Line 3). The iterative update is conducted by synthesising samples one by one $\mathbf{X}_{syn}$ (Line 6-11).

Chapter 19
Membership Inference

As described in Section 3.5, membership inference is to determine if an input $\mathbf{x}$ is in the training dataset D_{train} of a machine learning model f. In the following, we consider two classes of attack methods: metric based methods and binary classifier based methods. While these methods may have different formalism and performance, all of them are based on the fact that the machine learning model f has different behaviour on training data and test data. The behavioural difference is exhibited through e.g., prediction probability, loss, and hidden activations.

19.1 Metric Based Method

Metric based method proceeds by first calculating a metric M and then determining the membership of $\mathbf{x}$ by comparing the value of the metric with a prespecified threshold. According to the knowledge available to the attacker, there are different metrics as follows.

Prediction Label Metric

Recall that C is the set of classes and y is the ground truth label of $\mathbf{x}$, we have

$$M_{label}(\mathbf{x}) = \mathbb{1}(\arg\max_{c \in C} f(\mathbf{x})(c) = y) \tag{19.1}$$

where $\hat{y} = \arg\max_{c \in C} f(\mathbf{x})(c)$ is the predictive label and $\mathbb{1}$ is the indicator function such that

$$\mathbb{1}(a) = \begin{cases} 1 \text{ if } a \text{ is True} \\ 0 \text{ otherwise} \end{cases} \tag{19.2}$$

Intuitively, this metric is based on the assumption that the model f generalised so badly that it can only predict correctly on the training dataset. Based on this

assumption, the sample is in the training dataset if and only if the prediction made by f is correct. This attack method is black-box, i.e., it requires only knowledge **K6** (Section 3.6).

However, most machine learning algorithms – in particular deep neural networks – can generalise well, which renders the above metric too coarse for membership inference.

Prediction Loss Metric

Let $\mathcal{L}$ be the training loss function and ϵ a prespecified threshold, we define

$$M_{loss}(\mathbf{x}) = \mathbb{1}(\mathcal{L}(f(\mathbf{x}), y) \leq \epsilon) \tag{19.3}$$

Intuitively, it means that the sample $\mathbf{x}$ is a member of the training dataset if its prediction loss is insignificant (i.e., smaller than ϵ). The rationale behind this is that, the machine learning algorithm is trained by minimising the loss of the training data, and therefore a smaller loss suggests a higher possibility that the sample is in the training dataset. Unlike M_{label} which is a black-box attack, this method requires the knowlege about the training process (or more specifically, the training loss function), i.e., the attacker knowledge **K5**.

Prediction Entropy Metric

Recall the entropy as defined in Definition 5.1, we have

$$M_{loss}(\mathbf{x}) = \mathbb{1}(H(f(\mathbf{x})) \leq \epsilon) \tag{19.4}$$

where ϵ is a pre-specified threshold. This metric is based on an observation that the uncertainty of the prediction, expressed as the entropy of the prediction probability, is smaller for training data sample than test data sample. This method is black-box.

19.2 Binary Classifier Based Method

While the above metric-based methods are simple and easy to compute, they might not perform well because modern machine learning algorithms can generalise well, and more importantly, the machine learning models are very complex so that their different behaviours on the test and training datasets cannot be easily differentiated with simple metrics. This has led to the below discussion on learning a binary classifier for the membership inference.

Intuitively, binary classifier based attack is to learn a binary classifier f_a, which is able to predict the membership according to the prediction probability. Algorithm 12 presents a generic framework of conducting membership inference attack. It is a black-box algorithm and therefore can be applied to any machine learning model.

Algorithm 12: ***MembershipInferenceAttack***(f_{victim}, m, k, n), where f_{victim} is the original model that the user can access/query, m is the number of input features, k is the number of classes, and n is the number of samples in the synthesised dataset.

1 Synthesising a dataset $\mathbf{X}_{syn}$ of n samples $\{(\mathbf{x}_1, f_{victim}(\mathbf{x}_1)), ..., \mathbf{x}_n, f_{victim}(\mathbf{x}_n)\}$
2 Construct/train a function f_g: $(\mathbb{R}^m, \mathbb{R}^k) \rightarrow (\mathbb{R}^k, \{0, 1\})$ that maps inputs in $\mathbf{X}_{syn}$ to pairs of probability and a binary value
3 Construct/train a function f_a: $\mathbb{R}^k \rightarrow \{0, 1\}$ that maps probabilities to binary values by using the dataset $f_g(\mathbf{X}_{syn})$.
4 **return** $f_a \cdot f_{victim}$: $\mathbb{R}^m \rightarrow \{0, 1\}$, which maps any sample $\mathbf{x}$ to a binary value indicating whether $\mathbf{x}$ is in the training dataset of f_{victim}.

The algorithm takes three steps, as discussed in the following.

Synthesising dataset $\mathbf{X}_{syn}$

The dataset $\mathbf{X}_{syn}$ is of the same distribution as the the training dataset. This can be obtained with e.g., the iterative process as in Line 6-11 of Algorithm 11. But other algorithms may also apply.

Constructing function f_g

We partition the dataset $\mathbf{X}_{syn}$ into $\mathbf{X}_{syn,train}$ and $\mathbf{X}_{syn,test}$, and use $\mathbf{X}_{syn,train}$ to train a surrogate model $f_{surrogate}$. Then, we let

$$f_g((\mathbf{x}, y)) = \begin{cases} (f_{surrogate}(\mathbf{x}), 1) \text{ if } \mathbf{x} \in \mathbf{X}_{syn,train}, \\ (f_{surrogate}(\mathbf{x}), 0) \text{ if } \mathbf{x} \in \mathbf{X}_{syn,test} \end{cases} \tag{19.5}$$

This construction can be improved if we consider ensemble method, and consider a set of surrogate models, one for a partition of the dataset $\mathbf{X}_{syn}$.

Constructing function f_a

From f_g and $\mathbf{X}_{syn}$, we have a dataset $f_g(\mathbf{X}_{syn})$. The function can be obtained by using $f_g(\mathbf{X}_{syn})$ as the training dataset on any machine learning model.

Predicting whether new sample x is in the training dataset of f_{victim}

If $f_a(f_{victim}(\mathbf{x})) = 0$ then it is predicted that $\mathbf{x}$ is not in the training dataset. However, if $f_a(f_{victim}(\mathbf{x})) = 1$ then it is predicted that $\mathbf{x}$ is in the training dataset.

19.3 Discussion

The above membership inference algorithm is generic and model agnostic. It works well for simple dataset, such as low-dimensional tabular datasets, but might not work as well for complex datasets, such as high-resolution image datasets. This is mainly because the creation of dataset $\mathbf{X}_{syn}$. For high-dimensional datasets, the creation of a synthetic dataset that is of the same distribution as the training dataset is non-trivial, and may require a large number of samples.

Moreover, a membership inference attack becomes easier when the original model is overfitted. Intuitively, an overfitted model "remembers" the training data samples in its trainable parameters, and is therefore subject to the attack. Therefore, the improvement to the generalisation – or the reduction of generalisation error – of the model also poses a positive impact to the data privacy. Nevertheless, a more principled study on the exact relation between them is needed.

Bibliographic Notes

Poisoning Attack

A method to solve the bi-level optimisation is discussed in [14]. The heuristic methods are discussed in [74, 102, 72].

Model Stealing

The adaptive approach for model stealing is discussed in [12, 59]

Membership Inference

Metric-based methods are discussed in [96, 73] and binary classifier based method is discussed in [75].

Part IV
Robustness Verification of Deep Learning

This chapter considers the formal verification techniques on deep neural networks. Formal verification is a collection of techniques that, given a model and a property, automatically determine whether the property holds on the model. We focus on the robustness property and the convolutional neural network.

Chapter 20
Robustness Properties

A (deep and feedforward) neural network, or neural network, can be defined as a tuple $\mathcal{N} = (\mathbb{S}, \mathbb{T}, \Phi)$, where $\mathbb{S} = \{\mathbb{S}_k \mid k \in \{1..K\}\}$ is a set of layers, $\mathbb{T} \subseteq \mathbb{S} \times \mathbb{S}$ is a set of connections between layers and $\Phi = \{\phi_k \mid k \in \{2..K\}\}$ is a set of functions, one for each non-input layer. In a neural network, $\mathbb{S}_1$ is the *input* layer, $\mathbb{S}_K$ is the *output* layer, and layers other than input and output layers are called *hidden layers*. Each layer $\mathbb{S}_k$ consists of s_k *neurons* (or nodes). The l-th node of layer k is denoted by $n_{k,l}$.

Each node $n_{k,l}$ for $2 \leq k \leq K$ and $1 \leq l \leq s_k$ is associated with two variables $u_{k,l}$ and $v_{k,l}$, to record its values before and after an activation function, respectively. The Rectified Linear Unit (ReLU) [57] is one of the most popular activation functions for neural networks, according to which the *activation value* of each node of hidden layers is defined as

$$v_{k,l} = ReLU(u_{k,l}) = \begin{cases} u_{k,l} & \text{if } u_{k,l} \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{20.1}$$

Each input node $n_{1,l}$ for $1 \leq l \leq s_1$ is associated with a variable $v_{1,l}$ and each output node $n_{K,l}$ for $1 \leq l \leq s_K$ is associated with a variable $u_{K,l}$, because no activation function is applied on them. Other popular activation functions beside ReLU include: sigmoid, tanh, and softmax.

Except for the nodes at the input layer, every node is connected to nodes in the preceding layer by pre-trained parameters such that for all k and l with $2 \leq k \leq K$ and $1 \leq l \leq s_k$

$$u_{k,l} = b_{k,l} + \sum_{1 \leq h \leq s_{k-1}} w_{k-1,h,l} \cdot v_{k-1,h} \tag{20.2}$$

where $w_{k-1,h,l}$ is the weight for the connection between $n_{k-1,h}$ (i.e., the h-th node of layer $k-1$) and $n_{k,l}$ (i.e., the l-th node of layer k), and $b_{k,l}$ the so-called *bias* for node $n_{k,l}$. We note that this definition can express both fully-connected functions and convolutional functions[1]. The function ϕ_k is the composition of Equation (20.1) and

[1] Many of the surveyed techniques can work with other types of functional layers such as max-pooling, batch-normalisation, etc. Here for simplicity, we omit their expressions.

(20.2) by having $u_{k,l}$ for $1 \leq l \leq s_k$ as the intermediate variables. Owing to the use of the ReLU as in (20.1), the behavior of a neural network is highly non-linear.

Let $\mathbb{R}$ be the set of real numbers. We let $\mathcal{D}_k = \mathbb{R}^{s_k}$ be the vector space associated with layer $\mathbb{S}_k$, one dimension for each variable $v_{k,l}$. Notably, every point $\mathbf{x} \in \mathcal{D}_1$ is an input. Without loss of generality, the dimensions of an input are normalised as real values in $[0, 1]$, i.e., $\mathcal{D}_1 = [0, 1]^{s_1}$. A neural network $\mathcal{N}$ can alternatively be expressed as a function $f : \mathcal{D}_1 \rightarrow \mathcal{D}_K$ such that

$$f(\mathbf{x}) = \phi_K(\phi_{K-1}(...\phi_2(\mathbf{x}))) \tag{20.3}$$

Finally, for any input, the neural network $\mathcal{N}$ assigns a *label*, that is, the index of the node of output layer with the largest value:

$$label = \text{argmax}_{1 \leq l \leq s_K} u_{K,l} \tag{20.4}$$

Moreover, we let = $\{1..s_K\}$ be the set of labels.

Example 20.1 Figure 20.1 is a simple neural network with four layers. The input space is $\mathcal{D}_1 = [0, 1]^2$, the two hidden vector spaces are $\mathcal{D}_2 = \mathcal{D}_3 = \mathbb{R}^3$, and the set of labels is = $\{1, 2\}$.

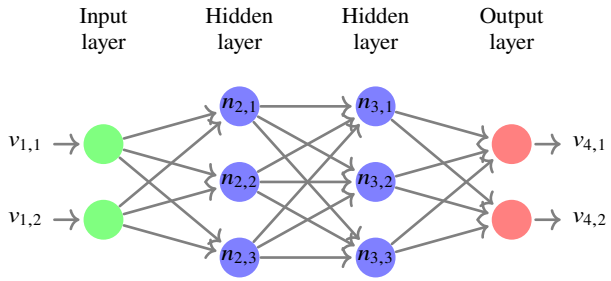


Fig. 20.1: A simple neural network

Given one particular input $\mathbf{x}$, the neural network $\mathcal{N}$ is *instantiated* and we use $\mathcal{N}[\mathbf{x}]$ to denote this instance of the network. In $\mathcal{N}[\mathbf{x}]$, for each node $n_{k,l}$, the values of the variables $u_{k,l}$ and $v_{k,l}$ are fixed and denoted as $u_{k,l}[\mathbf{x}]$ and $v_{k,l}[\mathbf{x}]$, respectively. Thus, the activation or deactivation of each ReLU operation in the network is similarly determined. We define

$$sign_{\mathcal{N}}(n_{k,l}, \mathbf{x}) = \begin{cases} +1 & \text{if } u_{k,l}[\mathbf{x}] = v_{k,l}[\mathbf{x}] \\ -1 & \text{otherwise} \end{cases} \tag{20.5}$$

The subscript $\mathcal{N}$ will be omitted when clear from the context. The classification label of x is denoted as $\mathcal{N}[\mathbf{x}].label$.

Example 20.2 Let $\mathcal{N}$ be a neural network whose architecture is given in Figure 20.1. Assume that the weights for the first three layers are as follows:

$$\mathbf{W}_1 = 0.8\begin{bmatrix} 4 & 0 & -1 \\ 1 & -2 & 1 \end{bmatrix},\ \mathbf{W}_2 = 0.8\begin{bmatrix} 2 & 3 & -1 \\ -7 & 6 & 4 \\ 1 & -5 & 9 \end{bmatrix}$$

and that all biases are 0. When given an input $\mathbf{x} = [0, 1]$, we get $sign(n_{2,1}, \mathbf{x}) = +1$, since $u_{2,1}[\mathbf{x}] = v_{2,1}[\mathbf{x}] = 1$, and $sign(n_{2,2}, \mathbf{x}) = -1$, since $u_{2,2}[\mathbf{x}] = -2 \neq 0 = v_{2,2}[\mathbf{x}]$.

Robustness as an Optimisation Problem

We have defined robustness in Section 3.2. For robustness verification, given an input $\mathbf{x}$, a d-neighbourhood $\eta(\mathbf{x}, L_p, d)$ (as in Definition B.1), and a neural network f, it is to check whether all inputs within the d-neighbourhood have the same label, i.e.,

$$\forall \mathbf{x}' : ||\mathbf{x} - \mathbf{x}'||_p < d \Rightarrow \hat{f}(\mathbf{x}) = \hat{f}(\mathbf{x}') \tag{20.6}$$

where $\hat{f}(\mathbf{x})$ returns the predictive label of $\mathbf{x}$ by f. Alternatively, this can be reduced to first finding the maximum safety radius δ such that

$$\begin{aligned} \delta \triangleq \min_{\mathbf{x}'} \ & ||\mathbf{x} - \mathbf{x}'||_p \\ s.t.\ & \hat{f}(\mathbf{x}) \neq \hat{f}(\mathbf{x}') \end{aligned} \tag{20.7}$$

Then, it is to check whether $d \leq \delta$ as the verification result.

Chapter 21
Reduction to Mixed Integer Linear Programming

In this chapter, we present how we can reduce the verification problem (Equation (20.7)) to the mixed integer linear programming (MILP) problems, so that it can be solved with the off-the-shelf MILP solvers. We will also consider the over-approximation of the problem so that it is able to be solved with linear programming. We will focus on the ReLU neural network (i.e., all activation functions are ReLU) and the robustness property.

21.1 Reduction to MILP

Let $\mathbf{x}_c$ be the original input whose label is $y_c = \hat{f}(\mathbf{x}_c)$. Recall from Chapter 20 that, we assume the network f has K layers. Then, Equation (20.7) can be rewritten as

$$\begin{aligned} \max_{\mathbf{x}} \ & ||\mathbf{x} - \mathbf{x}_c||_p \\ s.t. \ & \mathbf{x} = \mathbf{v}_1, \\ & \mathbf{up}_{i+1} = \mathbf{W}_i \mathbf{v}_i + \mathbf{b}_i, \qquad i = 1..K-1 \\ & \mathbf{v}_{i+1} = ReLU(\mathbf{up}_{i+1}), \qquad i = 1..K-2 \\ & \mathbf{up}_K(y_c) - \mathbf{up}_K(y) \geq 0, \ y \in C \end{aligned} \tag{21.1}$$

where $\mathbf{up}_i, \mathbf{v}_i$ denote the activation vector of layer i before and after the ReLU function, respectively. The first condition confirms to have $\mathbf{x}$ as the activation vector of the input layer. The second and third conditions implement the linear transformation and ReLU activation function of layer $i+1$, respectively. The fourth condition requires that $\mathbf{x}$ has the label y_c. Specifically, the label is y_c if and and only if $\forall y \in C: \mathbf{up}_K(y_c) - \mathbf{up}_K(y) \geq 0$.

Considering that the ReLU function is non-linear, we introduce two methods of transforming the second and third conditions of Equation (21.1) into MILP constraints, i.e., linear constraints with Boolean variables.

Method One for Layers

The first method requires one Binary variable for each neuron. Let $\mathbf{t}_{i+1}$ have value 0 or 1 in its entries and have the same dimension as $\mathbf{v}_{i+1}$, and M be a very large constant number that can be treated as ∞. We do not need $\mathbf{u}_{i+1}$. Specifically, we have the following MILP constraints for every layer $i = 1..K-2$ to replace the second and third conditions of Equation (21.1):

$$\begin{aligned} &\mathbf{v}_{i+1} \geq \mathbf{W}_i\mathbf{v}_i + \mathbf{b}_i, \\ &\mathbf{v}_{i+1} \leq \mathbf{W}_i\mathbf{v}_i + \mathbf{b}_i + M\mathbf{t}_{i+1}, \\ &\mathbf{v}_{i+1} \geq \mathbf{0}, \\ &\mathbf{v}_{i+1} \leq M(1 - \mathbf{t}_{i+1}), \end{aligned} \tag{21.2}$$

To understand how it works, if $\mathbf{t}_{i+1} = \mathbf{0}$ then Equation (21.2) can be simplified as $\mathbf{v}_{i+1} = \mathbf{W}_i\mathbf{v}_i + \mathbf{b}_i$ and $\mathbf{0} \leq \mathbf{v}_{i+1} \leq M$, which corresponds to the case of $\mathbf{up}_{i+1} \geq \mathbf{0}$. On the other hand, if $\mathbf{t}_{i+1} = 1$ then Equation (21.2) is reduced to $\mathbf{v}_{i+1} = \mathbf{0}$, which corresponds to the case of $\mathbf{up}_{i+1} < \mathbf{0}$. These can be extended to work with the general case where the elements in $\mathbf{t}_{i+1}$ can be either 0 or 1. In such case, the inequations in Equation (21.2) can be dealt with in an element-wise way.

Further, if we have additional upper and lower bounds, $\mathbf{lo}_i$ and $\mathbf{up}_i$, for $\mathbf{v}_i$, then Equation (21.1) can be rewritten into

$$\begin{aligned} &\mathbf{v}_{i+1} \geq \mathbf{W}_i\mathbf{v}_i + \mathbf{b}_i, \\ &\mathbf{v}_{i+1} \leq \mathbf{W}_i\mathbf{v}_i + \mathbf{b}_i - \mathbf{lo}_{i+1}\mathbf{t}_{i+1}, \\ &\mathbf{v}_{i+1} \geq \mathbf{0}, \\ &\mathbf{v}_{i+1} \leq \mathbf{up}_{i+1}(1 - \mathbf{t}_{i+1}), \end{aligned} \tag{21.3}$$

Note that, the only differences with Equation (21.2) are on the second and fourth conditions, where $\mathbf{lo}_{i+1}$ and $\mathbf{up}_{i+1}$ instead of the large number M are used. To understand how it works, if $\mathbf{t}_{i+1} = \mathbf{0}$ then Equation (21.3) is reduced to $\mathbf{v}_{i+1} = \mathbf{W}_i\mathbf{v}_i + \mathbf{b}_i$ and $\mathbf{0} \leq \mathbf{v}_{i+1} \leq \mathbf{up}_{i+1}$, which corresponds to the case of $\mathbf{up}_{i+1} \geq \mathbf{0}$. On the other hand, if $\mathbf{t}_{i+1} = 1$ then Equation (21.3) is reduced to $\mathbf{v}_{i+1} = \mathbf{0}$ and $\mathbf{W}_i\mathbf{v}_i + \mathbf{b}_i \leq \mathbf{v}_{i+1} \leq \mathbf{W}_i\mathbf{v}_i + \mathbf{b}_i - \mathbf{lo}_{i+1}$. Note that, $\mathbf{W}_i\mathbf{v}_i + \mathbf{b}_i - \mathbf{lo}_{i+1} > \mathbf{0}$. Therefore, this corresponds to the case of $\mathbf{up}_{i+1} < \mathbf{0}$. Similarly, Equation (21.2) should be treated in the element-wise way. One of the approaches of computing lower and upper bounds can be seen from Section 21.3.

Method Two for Layers

Different from the first method, the second method focuses on the ReLU activation function. It requires both $\mathbf{u}_{i+1}$ and $\mathbf{v}_{i+1}$. Specifically, we have

$$\begin{aligned}
&\mathbf{u}_{i+1} = \mathbf{W}_i \mathbf{v}_i + \mathbf{b}_i, \\
&\mathbf{v}_{i+1} \geq \mathbf{0} \\
&\mathbf{v}_{i+1} \geq \mathbf{u}_{i+1} \\
&\mathbf{v}_{i+1} \leq \mathbf{up}_{i+1} \odot \mathbf{t}_{i+1} \\
&\mathbf{v}_{i+1} \leq \mathbf{u}_{i+1} - \mathbf{lo}_{i+1} \odot (\mathbf{1} - \mathbf{t}_{i+1})
\end{aligned} \tag{21.4}$$

where $\odot$ is the element-wise multiplication. To understand how it works, if $\mathbf{t}_{i+1} = 1$ then we have $\mathbf{0} \leq \mathbf{v}_{i+1} = \mathbf{u}_{i+1} \leq \mathbf{up}_{i+1}$, which corresponds to the case of $\mathbf{up}_{i+1} \geq \mathbf{0}$. On the other hand, if $\mathbf{t}_{i+1} = \mathbf{0}$, then we have $\mathbf{u}_{i+1} \leq \mathbf{v}_{i+1} = \mathbf{0} \leq \mathbf{u}_{i+1} - \mathbf{lo}_{i+1}$, which corresponds to the case of $\mathbf{up}_{i+1} < \mathbf{0}$.

Optimisation Objective

For the optimisation objective $||\mathbf{x} - \mathbf{x}_c||_p$ in Equation (21.1), different conversions are needed for different norm distance metric L_p.

For L_1 norm, we introduce auxiliary variables $\mathbf{z}$, which bound the absolute value of $\mathbf{x} - \mathbf{x}_c$, i.e., let $\mathbf{z} \leq \mathbf{x}_c - \mathbf{x}$ and $\mathbf{z} \leq \mathbf{x} - \mathbf{x}_c$. Therefore, we have

$$\begin{aligned}
&\max_{\mathbf{x}} \sum \mathbf{z} \\
&s.t.\ \mathbf{x} = \mathbf{v}_1, \\
&\quad \mathbf{up}_{i+1} = \mathbf{W}_i \mathbf{v}_i + \mathbf{b}_i, && i = 1..K-1 \\
&\quad \mathbf{v}_{i+1} = ReLU(\mathbf{up}_{i+1}), && i = 1..K-2 \\
&\quad \mathbf{up}_K(y_c) - \mathbf{up}_K(y) \geq 0, && y \in C \\
&\quad \mathbf{z} \leq \mathbf{x}_c - \mathbf{x} \\
&\quad \mathbf{z} \leq \mathbf{x} - \mathbf{x}_c
\end{aligned} \tag{21.5}$$

where $\sum \mathbf{z}$ is the element-wise summation of the vector $\mathbf{z}$. Note that $\mathbf{z} < \mathbf{0}$. Therefore, the maximisation over $\sum \mathbf{z}$ is to find the closest $\mathbf{x}$ (with respect to $\mathbf{x}_c$ and L_1 norm).

For L_∞ norm, we introduce a single auxiliary variables z_∞, which bound the L_∞ norm of $\mathbf{x} - \mathbf{x}_c$, i.e., let $z_\infty \leq \mathbf{x}_c(i) - \mathbf{x}(i)$ and $z_\infty \leq \mathbf{x}(i) - \mathbf{x}_c(i)$, for all $i \in [1..s_1]$. Therefore, we have

$$\begin{aligned}
&\max_{\mathbf{x}} z_\infty \\
&s.t.\ \mathbf{x} = \mathbf{v}_1, \\
&\quad \mathbf{up}_{i+1} = \mathbf{W}_i \mathbf{v}_i + \mathbf{b}_i, && i = 1..K-1 \\
&\quad \mathbf{v}_{i+1} = ReLU(\mathbf{up}_{i+1}), && i = 1..K-2 \\
&\quad \mathbf{up}_K(y_c) - \mathbf{up}_K(y) \geq 0, && y \in C \\
&\quad z_\infty \leq \mathbf{x}_c(i) - \mathbf{x}(i), && i \in [1..s_1] \\
&\quad z_\infty \leq \mathbf{x}(i) - \mathbf{x}_c(i), && i \in [1..s_1]
\end{aligned} \tag{21.6}$$

Note that $z_\infty \leq 0$. Therefore, the maximisation over z_∞ is to find the closest $\mathbf{x}$ (with respect to $\mathbf{x}_c$ and L_∞).

For L_2 norm, the objective becomes quadratic, and therefore we may have to use mixed integer quadratic programming (MIQP), without the need of auxiliary variable.

That is,

$$\begin{aligned} \max_{\mathbf{x}} \ & \sum_{i=1}^{s_1} (\mathbf{x}(i) - \mathbf{x}_c(i))^2 \\ s.t. \ & \mathbf{x} = \mathbf{v}_1, \\ & \mathbf{up}_{i+1} = \mathbf{W}_i \mathbf{v}_i + \mathbf{b}_i, \qquad i = 1..K-1 \\ & \mathbf{v}_{i+1} = ReLU(\mathbf{up}_{i+1}), \qquad i = 1..K-2 \\ & \mathbf{up}_K(y_c) - \mathbf{up}_K(y) \geq 0, \ y \in C \end{aligned} \tag{21.7}$$

21.2 Overapproximation with LP

Equation (21.4) can be over-approximated with the below method (illustrated in Figure 21.1) on the ReLU activation function. Intuitively, when mapping from $\mathbf{up}_i(j)$

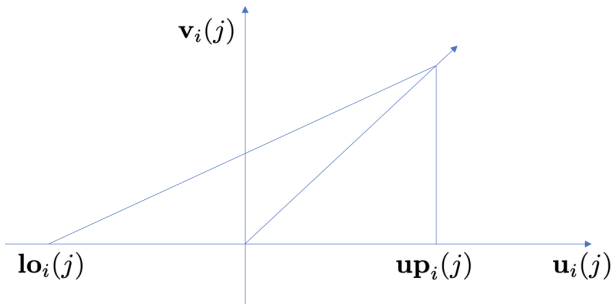


Fig. 21.1: Relaxation of ReLU Activation

to $\mathbf{v}_i(j)$, instead of using the two lines from ReLU function, i.e., from $(\mathbf{lo}_i(j), 0)$ to $(0,0)$ and from $(0,0)$ to $(\mathbf{up}_i(j), \mathbf{up}_i(j))$, we can use the line from $(\mathbf{lo}_i(j), 0)$ to $(\mathbf{up}_i(j), \mathbf{up}_i(j))$ to over-approximate the value of $\mathbf{v}_i(j)$. With this idea, the last three conditions of Equation (21.4) can be replaced with the below ones for every $j \in [1..s_i]$:

$$\begin{cases} \mathbf{v}_{i+1}(j) = 0 & \text{if } \mathbf{up}_{i+1}(j) \leq 0 \\ \mathbf{v}_{i+1}(j) = \mathbf{up}_{i+1}(j) & \text{if } \mathbf{lo}_{i+1}(j) \geq 0 \\ \mathbf{v}_{i+1}(j) \geq 0, \mathbf{v}_{i+1}(j) \geq \mathbf{up}_{i+1}(j), & \\ \mathbf{v}_{i+1}(j) \leq \dfrac{\mathbf{up}_{i+1}(j)(\mathbf{up}_{i+1}(j) - \mathbf{lo}_{i+1}(j))}{\mathbf{up}_{i+1}(j) - \mathbf{lo}_{i+1}(j)} & \text{otherwise} \end{cases} \tag{21.8}$$

where the three conditions for the case when $\mathbf{up}_{i+1}(j) \geq 0$ and $\mathbf{lo}_{i+1}(j) \leq 0$ represent the triangle determined by the three lines such that the value of $\mathbf{up}_{i+1}(j)$ will be in the triangle. Therefore, according to the known upper and lower bounds $\mathbf{up}_{i+1}(j)$ and $\mathbf{lo}_{i+1}(j)$, one of the options in Equation (21.8) will be chosen. Note that, all options are linear, without using any Binary variables.

Based on the above discussion, if all ReLU activation functions are approximated with Equation (21.8), the MILP problem becomes linear programming (LP), which can in turn be solved with LP solver in linear time. We remark that, by approximating the MILP problem (whose computational complexity is NP-complete) with an LP problem (whose computational complexity is in P), solving the optimisation problem becomes tractable and it is possible to work with neural network of larger size. On the other hand, due to the over-approximation, when we are not able to affirm $d \leq \delta$ as the verification result (as we discussed after Equation (20.7)), we are not able to draw a conclusion on the verification.

21.3 Computation of Lower and Upper Bounds through Lipschitz Approximation

In this section, we consider how to approximate Lipschitz constant and how to utilise Lipschitz constant for the computation of lower and upper bounds $\mathbf{lo}_i$ and $\mathbf{up}_i$. Let

$$\mathbf{G}_i(\mathbf{x}) = \frac{\partial \mathbf{v}_i}{\partial \mathbf{x}} \qquad \text{for } i = 1..K \tag{21.9}$$

be the gradient vector of the hidden activation $\mathbf{v}_i$ over the input $\mathbf{x}$.

Approximation of Lipschitz Constant

We explain how to compute $\mathbf{G}_i(\mathbf{x})$ by utilising the structural information of the neural network. Actually, we will compute both its lower bound $\underline{\mathbf{G}_i} \in^{s_i \times s_1}$ and its upper bound $\overline{\mathbf{G}_i} \in^{s_i \times s_1}$. Before proceeding, we define a few notations and operators. Let $[a]_+ = \max\{a, 0\}$ and $[a]_- = \min\{a, 0\}$. For for a matrix $\mathbf{F}$, $[\mathbf{F}]_+$ and $[\mathbf{F}]_-$ are for element-wise max and min. Moreover, we define

$$\mathbf{W} \otimes [\mathbf{lo}, \mathbf{up}] = [[\mathbf{W}]_+ \times \mathbf{lo} + [\mathbf{W}]_- \times \mathbf{up}, [\mathbf{W}]_+ \times \mathbf{up} + [\mathbf{W}]_- \times \mathbf{lo}] \tag{21.10}$$

Actually, we can utilise the chain rule and the sequential structure of the neural network to have the following equation:

$$\mathbf{G}_i(\mathbf{x}) = \frac{\partial \mathbf{v}_i}{\partial \mathbf{x}} = \frac{\partial \mathbf{v}_i}{\partial \mathbf{u}_i} \frac{\partial \mathbf{u}_i}{\partial \mathbf{v}_{i-1}} \cdots \frac{\partial \mathbf{v}_2}{\partial \mathbf{u}_2} \frac{\partial \mathbf{u}_2}{\partial \mathbf{v}_1} \tag{21.11}$$

which suggests that the gradient $\mathbf{G}_i(\mathbf{x})$ can be computed repeatedly over the layers. Note that, $\frac{\partial \mathbf{u}_i}{\partial \mathbf{v}_{i-1}} = \mathbf{W}_{i-1}$, where $\mathbf{W}_{i-1}$ is the weight matrix of layer $i-1$. Let $\nabla \boldsymbol{\sigma}_i = \frac{\partial \mathbf{v}_i}{\partial \mathbf{u}_i} \in \mathbb{R}^{s_i \times s_i}$, we have

$$\mathbf{G}_i(\mathbf{x}) = \frac{\partial \mathbf{v}_i}{\partial \mathbf{x}} = \nabla \boldsymbol{\sigma}_i \mathbf{W}_{i-1} ... \mathbf{W}_2 \nabla \boldsymbol{\sigma}_2 \mathbf{W}_1 \tag{21.12}$$

Let $\underline{\Lambda}_i, \overline{\Lambda}_i \in \mathbb{R}^{s_i \times s_i}$ be the diagonal matrices denoting the lower and upper bound of $\nabla \boldsymbol{\sigma}_i$, respectively. For ReLU activation function, we have

$$\mathbf{1} \geq \overline{\Lambda}_i \geq \underline{\Lambda}_i \geq \mathbf{0} \tag{21.13}$$

To enable a computation of the lower and upper bounds, we use an iterative process as follows. Let $\mathbf{F}_{i+1} = \mathbf{W}_i \mathbf{G}_i$ (i.e., $= \frac{\partial \mathbf{u}_{i+1}}{\partial \mathbf{x}}$), we have

$$\mathbf{G}_i = \nabla \boldsymbol{\sigma}_i \mathbf{F}_i = \nabla \boldsymbol{\sigma}_i \mathbf{W}_{i-1} \mathbf{G}_{i-1} \tag{21.14}$$

The computation proceeds as follows. Initially, we have

$$\underline{\mathbf{G}}_0 = \overline{\mathbf{G}}_0 = \mathbf{I} \tag{21.15}$$

where $\mathbf{I}$ is the identify matrix. Then, given $\underline{\mathbf{G}}_{i-1}$ and $\overline{\mathbf{G}}_{i-1}$, we have

$$\begin{aligned} [\underline{\mathbf{F}}_i, \overline{\mathbf{F}}_i] &= \mathbf{W}_i \otimes [\underline{\mathbf{G}}_{i-1}, \overline{\mathbf{G}}_{i-1}] \\ \overline{\mathbf{G}}_i &= \max\{\underline{\Lambda}_i \overline{\mathbf{F}}_i, \overline{\Lambda}_i \overline{\mathbf{F}}_i\} = \overline{\Lambda}_i [\overline{\mathbf{F}}_i]_+ + \underline{\Lambda}_i [\overline{\mathbf{F}}_i]_- \\ \underline{\mathbf{G}}_i &= \min\{\underline{\Lambda}_i \underline{\mathbf{F}}_i, \overline{\Lambda}_i \underline{\mathbf{F}}_i\} = \underline{\Lambda}_i [\underline{\mathbf{F}}_i]_+ + \overline{\Lambda}_i [\underline{\mathbf{F}}_i]_- \end{aligned} \tag{21.16}$$

Finally, given an input region, which can be either the entire input domain or a d-neighbourhood, if we know $\overline{\Lambda}_i$ and $\underline{\Lambda}_i$, we can compute $[\underline{\mathbf{G}}_i, \overline{\mathbf{G}}_i]$ with respect to the input region. Therefore, this method can be used to compute either the global Lipschitz constant (when the input region is the entire input domain) or the local Lipschitz constant (when the input region is a d-neighbourhood).

Computation of Lower and Upper Bounds

Let $\eta(\mathbf{x}, L_p, d)$ be a d-neighbourhood centred around $\mathbf{x}$, we can first use the above method to compute $[\underline{\mathbf{G}}_i, \overline{\mathbf{G}}_i]$. Then, the upper bound and lower bounds of the j-th dimension are as follows:

$$\begin{aligned} \mathbf{up}_i(j) &= \mathbf{v}_i(j) + \overline{\mathbf{G}}_i(j) \max_{\mathbf{x}' \in \eta(\mathbf{x}, L_p, d)} |\mathbf{x}(j) - \mathbf{x}'(j)| \\ \mathbf{lo}_i(j) &= \mathbf{v}_i(j) - \underline{\mathbf{G}}_i(j) \max_{\mathbf{x}' \in \eta(\mathbf{x}, L_p, d)} |\mathbf{x}(j) - \mathbf{x}'(j)| \end{aligned} \tag{21.17}$$

Chapter 22
Robustness Verification via Reachability Analysis

Chapter 23
Conclusion

Part V
Enhancement to Robustness and Generalization

Chapter 24
Robustness Enhancement through Min-Max Optimisation

(edit!!!) Deep neural networks (DNNs) can be easily fooled to confidently make incorrect predictions by adding small and human-imperceptible perturbations to their input [20, 84, 92]. The study of adversarial defenses, aiming to ultimately eliminate adversarial threat [40], has brought about significant advances in the past few years, with various techniques developed, including input denoising [4], adversarial detection [48], gradient regularization [86], feature squeezing [95], defensive distillation [61] and adversarial training [49, 61]. Among various techniques, adversarial training is known to be the most effective one [3].

Adversarial training considers adversarial examples during the training. Consider a training dataset $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^{m}$ with $m \in \mathbb{N}$ samples drawn from a distribution D, where $\mathbf{x}_i \in \mathbb{R}^d$ is an example in the d-dimensional input space and y_i is its ground-truth label, adversarial training [49] updates the minimization objective of the training scheme from the usual one as follows

$$J = \mathop{\mathbb{E}}_{(\mathbf{x},y)\sim D} \mathcal{L}(f_\theta(\mathbf{x}), y) \tag{24.1}$$

to

$$J_{\text{adv}} = \mathop{\mathbb{E}}_{(\mathbf{x},y)\sim D} \left[\max_{||\mathbf{x}'-\mathbf{x}||_p \leq \epsilon} \mathcal{L}(f_\theta(\mathbf{x}'), y) \right], \tag{24.2}$$

where $\mathbf{x}'$ is the adversarial example within the ϵ-ball (bounded by an ℓ_p-norm) centered at clean example $\mathbf{x}$, $f_\theta(\cdot)$ is the DNN with parameter θ, and $\mathcal{L}(\cdot)$ is the standard classification loss (e.g., the cross-entropy loss). Therefore, adversarial training is formulated as a min-max optimization problem.

[Below are old Contents in the deep learning chapter, check to see if they are needed, or integrate with the above]

On the opposite side of adversarial attacks [6, 85], researchers also show huge interests in designing various defence techniques, which are to either identify or reduce adversarial examples so that the decision of the DNN can be more robust. Until now, the developments of attack and defence techniques have been seen as an "arm-race". For example, most defences against attacks in the white-box setting,

including [25, 51, 52, 61], have been demonstrated to be vulnerable to e.g., iterative optimisation-based attacks [7, 8].

24.0.1 Adversarial Training

Adversarial training is one of the most notable defence methods, which was first proposed by [19]. It can improve the robustness of DNNs against adversarial attacks by retraining the model on adversarial examples. Its basic idea can be expressed as below:

$$\theta^* = \arg\min_{\theta} \mathop{\mathbb{E}}_{(\mathbf{x},y)\sim D} \mathcal{L}(\mathbf{x}; y; f_\theta). \tag{24.3}$$

This is improved in [49] by assuming that all neighbours within the ϵ-ball should have the same class label, i.e., local robustness. Technically, this is done by changing the optimisation problem by requiring that for a given ϵ-ball (represented as a d-Neighbourhood), to solve

$$\theta^* = \arg\min_{\theta} \mathop{\mathbb{E}}_{(\mathbf{x},y)\sim D} \left[\max_{\delta\in[-\epsilon,\epsilon]^{s_1}} \mathcal{L}(\mathbf{x}+\delta; y; f_\theta) \right]. \tag{24.4}$$

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*. [49] adopted Projected Gradient Descent (PGD) to approximately solve the inner maximization problem. Please see the template code in Section C for the adversarial training.

Later on, to defeat the iterative attacks, [56] proposed to use a cascade adversarial method which can produce adversarial images in every mini-batch. Namely, at each batch, it performs a separate adversarial training by putting the adversarial images (produced in that batch) into the training dataset. Moreover, [87] introduces ensemble adversarial training, which augments training data with perturbations transferred from other models.

Chapter 25
Generalisation Enhancement through PAC Bayesian Theory

Given a prior distribution over the parameters θ, which is selected before seeing a training dataset, a posterior distribution on θ will depend on both, the training dataset and a specific learning algorithm. The PAC-Bayesian framework [50] bounds the generalisation error with respect to the Kullback-Leibler (KL) divergence [39] between the posterior and the prior distributions.

Consider a training dataset S with $m \in \mathbb{N}$ samples drawn from a distribution D. Given a learning algorithm (e.g., a classifier) f_θ with prior and posterior distributions P and Q on the parameters θ respectively, for any $\delta > 0$, with probability $1 - \delta$ over the draw of training data, we have that [17, 50]

$$\mathbb{E}_{\theta\sim Q}[\mathcal{L}_D(f_\theta)] \leq \mathbb{E}_{\theta\sim Q}[\mathcal{L}_S(f_\theta)] + \sqrt{\frac{\mathrm{KL}(Q||P) + \log\frac{m}{\delta}}{2(m-1)}}, \tag{25.1}$$

where $\mathbb{E}_{\theta\sim Q}[\mathcal{L}_D(f_\theta)]$ is the expected loss on D, $\mathbb{E}_{\theta\sim Q}[\mathcal{L}_S(f_\theta)]$ is the empirical loss on S, and their difference yields the generalisation error. Eq. (25.1) outlines the role KL divergence plays in the upper bound of the generalisation error. In particular, a smaller KL term will help tighten the generalisation error bound. Assume that P and Q are Gaussian distributions with $P = \mathcal{N}(\mu_P, \Sigma_P)$ and $Q = \mathcal{N}(\mu_Q, \Sigma_Q)$, then the KL-term can be written as follows:

$$\begin{aligned}&\mathrm{KL}(\mathcal{N}(\mu_Q, \Sigma_Q)||\mathcal{N}(\mu_P, \Sigma_P))\\ &= \frac{1}{2}\Big[\mathrm{tr}(\Sigma_P^{-1}\Sigma_Q) + (\mu_Q - \mu_P)^\top\Sigma_P^{-1}(\mu_Q - \mu_P) - k + \ln\frac{\det\Sigma_P}{\det\Sigma_Q}\Big],\end{aligned} \tag{25.2}$$

where k is the number of parameters in θ. Below, [35] incorporates weight correlation into this framework to tighten the bound. Given weight matrix $w_l \in \mathbb{R}^{N_{l-1}\times N_l}$ of the l-th layer, the average weight correlation is defined as

$$\rho(N_l) = \frac{1}{N_l(N_l-1)}\sum_{\substack{i,j=1\\ i\neq j}}^{N_l}\frac{|w_{li}^T w_{lj}|}{||w_{li}||_2||w_{lj}||_2}, \tag{25.3}$$

where w_{li} and w_{lj} are i-th and j-th column of the matrix w_l, corresponding to the i-th and j-th neuron at l-th layer, respectively. Intuitively, $\rho(w_l)$ is the average cosine similarity between weight vectors of any two neurons at the l-th layer.

Given the above $\rho(w_l)$, [35] introduces a correlation matrix $\Sigma_{\rho(w_l)} \in \mathbb{R}^{N_l \times N_l}$, with diagonal elements being 1, and off-diagonal ones all $\rho(w_l)$. Then, the posterior corvariance matrix can be represented as $\Sigma_{Q_{w_l}} = \Sigma_{\rho(w_l)} \otimes \sigma_l^2 I_{N_{l-1}}$, where $\otimes$ is Kronecker product. Let $g(w) = \sum g(w_l)$ where $g(w_l)$ defined by:

$$g(w_l) = -(N_l - 1)N_{l-1} \ln(1 - \rho(w_l)) - N_{l-1} \ln(1 + (N_l - 1)\rho(w_l)). \tag{25.4}$$

Then the KL term w.r.t. the l-th layer can be given by

$$\mathrm{KL}(Q||P)_l = \frac{||\theta_l^F - \theta_l^0||_{\mathrm{Fr}}^2}{2\sigma_l^2} + \ln \frac{\det(\Sigma_{P_w})_l \cdot \det(\Sigma_{P_b})_l}{\det(\Sigma_{Q_w})_l \cdot \det(\Sigma_{Q_b})_l} = \frac{||\theta_l^F - \theta_l^0||_{\mathrm{Fr}}^2}{2\sigma_l^2} + g(w_l). \tag{25.5}$$

Further, when $\sigma_l^2 = \sigma^2$ for all l, we have $\mathrm{KL}(Q||P) = \sum_{l=1}^{L} \mathrm{KL}(Q||P)_l$. Given the above derivation, [35] concludes that the KL term in Eq. (25.5) is positively correlated to $g(w_l)$. Naturally, $g(w_l)$ can be considered as a regulariser term in the training function.

The training of a neural network is seen as a process of optimising over an objective function $J(\theta; X, y)$, where X is the input data, y is the corresponding label, and θ is the parameter. [35] adds a penalty term $g(w) = \sum_l g(w_l)$, which is a function of $\rho(w)$, to the objective function J, and denote the regularised objective function by $\tilde{J}$:

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha g(w), \tag{25.6}$$

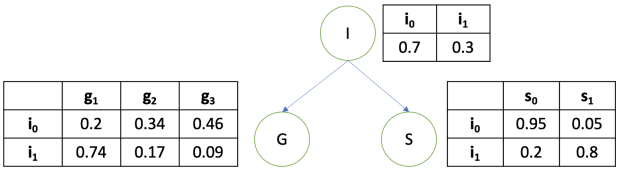
where $\alpha \in [0, \infty)$ is a hyper-parameter that balances the relative contribution of the $g(w_l)$ penalty term.

Part VI
Probabilistic Graph Models for Feature Robustness

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., [5] 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

Fig. 25.1: A simple graphical model of three nodes

Figure 25.1 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,

$$\text{Probabilistic Graphical Model} = \text{Graphical Structure} + \text{Multivariate Statistics} \tag{25.7}$$

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 $\mathcal{V}$.

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.

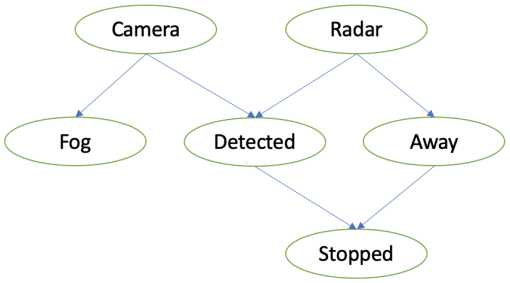


Fig. 25.2: A simple Bayesian network for safety analysis on vehicle stopping upon pedestrian detection

Figure 25.2 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 25.3. 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 appearing in the imagery input. This is intuitive, because the foggy condition may affect the ability of camera capturing the pedestrian. Similar

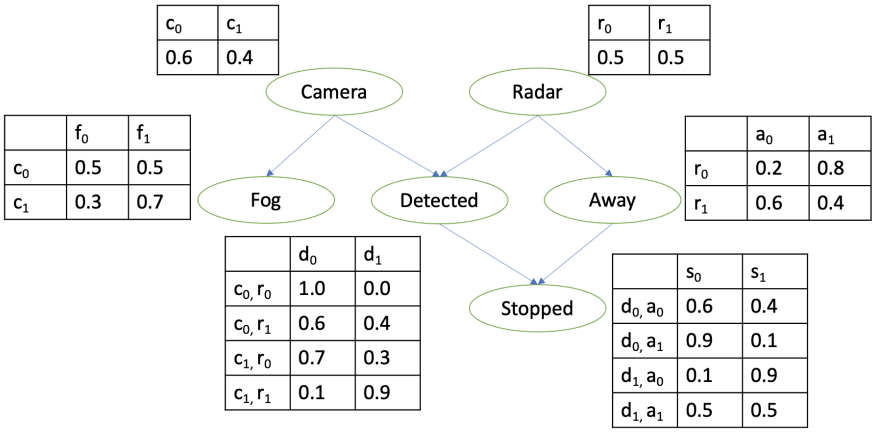


Fig. 25.3: Probabilistic table of the graphical model in Figure 25.2

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.

Chapter 26
I-Maps

This section studies the conditional independences in G.

26.1 Naive Bayes and Joint Probability

First of all, we explain the difference between usual classification and Naive Bayes classifier. As shown in Figure 26.1, 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 as another graphical model

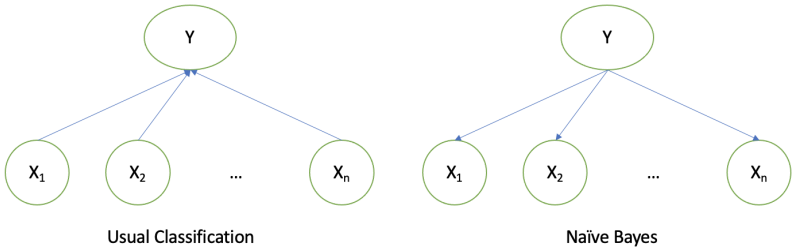


Fig. 26.1: Naive Bayes as a probabilistic graphical model

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

$$P(X_1, ..., X_n, Y) = P(Y) \prod_{i \in \{1..n\}} P(X_i|Y) \tag{26.1}$$

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

$$P(\mathcal{V}) = \prod_{V \in \mathcal{V}} P(V|Pa(V)) \tag{26.2}$$

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.

26.2 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 A.3 has presented a few examples on how to do the calculation.

26.3 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.,
>
> $$X \perp NonDesc(X)|Pa(X) \tag{26.3}$$

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 25.2, we have the following local conditional independences that can be read directly from the graph:

$$\begin{aligned} I(G) = \{ & (Fog \perp Detected, Radar, Away, Stopped|Camera), \\ & (Camera \perp Radar, Away), \\ & (Radar \perp Camera, Fog), \\ & (Detected \perp Away, Fog|Camera, Radar), \\ & (Away \perp Camera, Fog, Detected|Radar), \\ & (Stopped \perp Fog, Camera, Radar|Detected, Away)\} \end{aligned} \tag{26.4}$$

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 28, we will introduce a comprehensive method – d-separation – that is able to infer all possible conditional independencies from a graph.

26.4 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 26.2. Then, we define

Definition 26.1 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 26.1 Consider the joint probability table P as in Table A.1, and the three graphs in Figure 26.2. we note that $I(P) = \{X \perp Y\}$, $I(G_0) = \{X \perp Y\}$, and $I(G_1) =$

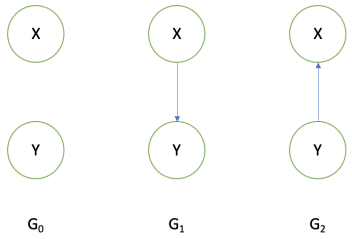


Fig. 26.2: Three simple, two-node graphs

$I(G_2) = \emptyset$. 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 A.2, 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 26.1 *If G is an I-map of P, then*

$$P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i|Pa(X_i)) \tag{26.5}$$

This justifies our conjecture at Equation (26.2).

26.5 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.

Chapter 27
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.

27.1 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 25.3, 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

$$P(s_0) \tag{27.1}$$

which is the probability of the car does not stop. This can be computed by having

$$\begin{aligned} P(s_0) &= \sum_{C,R,F,D,A} P(C,R,F,D,A,S=s_0) \\ &= \sum_{i=0}^{1}\sum_{j=0}^{1}\sum_{k=0}^{1}\sum_{l=0}^{1}\sum_{m=0}^{1} P(C=c_i,R=r_j,F=f_k,D=d_l,A=a_m,S=s_0) \\ &\approx 0.62 \end{aligned} \tag{27.2}$$

Then, we may consider a conditional probability such as

$$P(s_0|c_1) = \frac{P(s_0,c_1)}{P(c_1)} \approx 0.51 \tag{27.3}$$

which says that if we know that the camera captured the pedestrian, the probability of not stopped decreased to 0.51. Similarly, if consider radar, we have

$$P(s_0|r_1) = \frac{P(s_0, c_1)}{P(r_1)} \approx 0.44 \tag{27.4}$$

If both the camera and the radar are considered, we have

$$P(s_0|c_1, r_1) = \frac{P(s_0, c_1, r_1)}{P(c_1, r_1)} \approx 0.31 \tag{27.5}$$

27.2 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:

$$P(c_1) = 0.4 \tag{27.6}$$

After observing that the car stopped, she might infer as follows, which shows that the probability increased to 0.51.

$$P(c_1|s_1) = \frac{P(c_1, s_1)}{P(s_1)} \approx 0.51 \tag{27.7}$$

Intuitively, this may suggest that the specific camera installed on this car may perform above average.

27.3 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

$$P(r_1|d_1) \approx 0.83 \tag{27.8}$$

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

$$P(r_1|d_1, c_1) \approx 0.75 \tag{27.9}$$

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.

27.4 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))
```

Chapter 28
D-Separation

From Section 26, 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 28.1 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.

28.1 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 28.1.

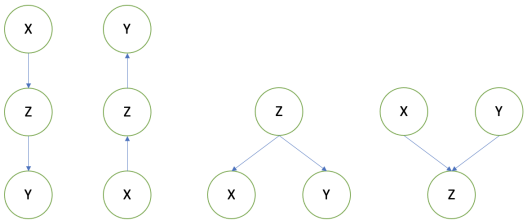


Fig. 28.1: Four patterns

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 28.2,

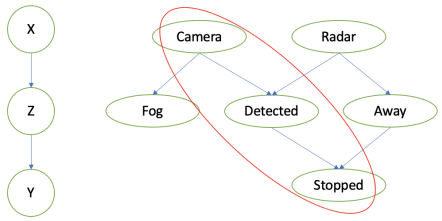


Fig. 28.2: Indirect Causal Effect

Indirect Evidential Effect $Y \rightarrow Z \rightarrow X$

Similarly, evidence X cannot influence the cause Y if Z is observed.

Common Cause $X \leftarrow Z \rightarrow Y$

Once Z is observed, one of the effects cannot influence the other. Figure 28.3 presents a case of common cause in our running example.

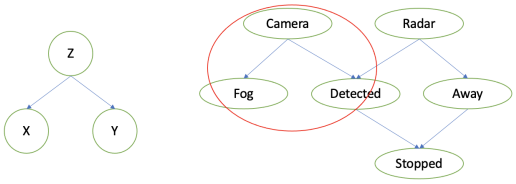


Fig. 28.3: Common Cause

Common Effect $X \rightarrow Z \leftarrow Y$

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 28.4 presents a case of common effect in our running example.

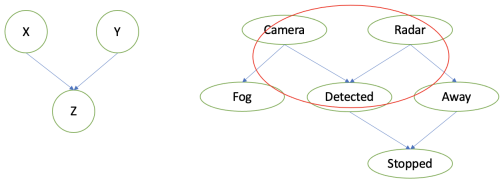


Fig. 28.4: Common Effect

28.2 General Case: Active Trail and D-Separation

Summarising the above discussion on local canonical cases (or v-structures), we have that

- Causal trail, $X \rightarrow Z \rightarrow Y$, 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, $X \leftarrow Z \rightarrow Y$, is active if and only if Z is not observed.
- Common effect, $X \rightarrow Z \leftarrow Y$, 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 13. 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.

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 14. Simply speaking, it requires all local triplets on the path to be inactive to make the path inactive.

Algorithm 13: ***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}
```

Algorithm 14: ***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 28.2 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.

Chapter 29
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.

29.1 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 29.1 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 29.1. Are X and Y independent? Actually, if we follow the exact computation, they are dependent. But we suspect that 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

X	Y	times
head	head	27
head	tail	22
tail	head	25
tail	head	26

Table 29.1: A simple coin tossing example

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.

29.2 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

$$d_I(D) = \frac{1}{|D|} \sum_{x_i, x_j} P(x_i, x_j) \log \frac{P(x_i, x_j)}{P(x_i)P(x_j)} \tag{29.1}$$

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

$$R_{d,t}(D) = \begin{cases} \text{Accept} & d(D) \leq t \\ \text{Reject} & d(D) > t \end{cases} \tag{29.2}$$

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 (29.1); 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

$$Score_L(G, D) = \sum_D \sum_{i=1}^{n} \log P(x_i | pa(x_i)) \tag{29.3}$$

  which can be seen as the loss of using graph G to predict D.
- Bayesian Score

$$Score_B(G, D) = \log P(D|G) + \log P(G) \tag{29.4}$$

  which, in addition to the log-likelihood score $\log P(D|G)$, it also consider the prior $P(G)$.
- Bayes score with penalty term

$$Score_{BIC}(G, D) = L(G, D) - \frac{\log |D|}{2} ||G|| \tag{29.5}$$

  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.

29.3 Practicals

The following code uses the **pomegranate** package to automatically learn structures from a simple dataset.

```
from pomegranate import BayesianNetwork
import seaborn, time, numpy, matplotlib
seaborn.set_style('whitegrid')

import pandas as pd
X = pd.DataFrame({'1':[0,0,0,1,0], '2':[0,0,1,0,0], '3'
    :[1,1,0,0,1], '4':[0,1,0,1,1]})
X = X.to_numpy()

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()

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()

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()
```

Chapter 30
Abstraction of Neural Network as Probabilistic Graphical Model

In this section, we construct a Bayesian network out of a trained neural network. In the end, the Bayesian network captures the distribution of neuron valuations in terms of latent features encoded in each neural network layers, as well as their causal relationships.

30.1 Extraction of Hidden Features

Assume that some feature extraction technique such as PCA and ICA has been used to analyse the neuron activation vector $\mathbf{v}_i$ of layer i that are induced by a given *training set* D_{train}. This produces a set of feature mappings $L_i = \{\lambda_{i,j}\}_{j\in\{1,\ldots,t_i\}}$ for t_i features, such that each $\lambda_{i,j}\colon \mathbb{L}_i \to \mathbb{F}_{i,j}$ maps the vector space $\mathbb{L}_i$ of neuron valuation into the j-th component of the feature space $\mathbb{F}_{i,j}$.

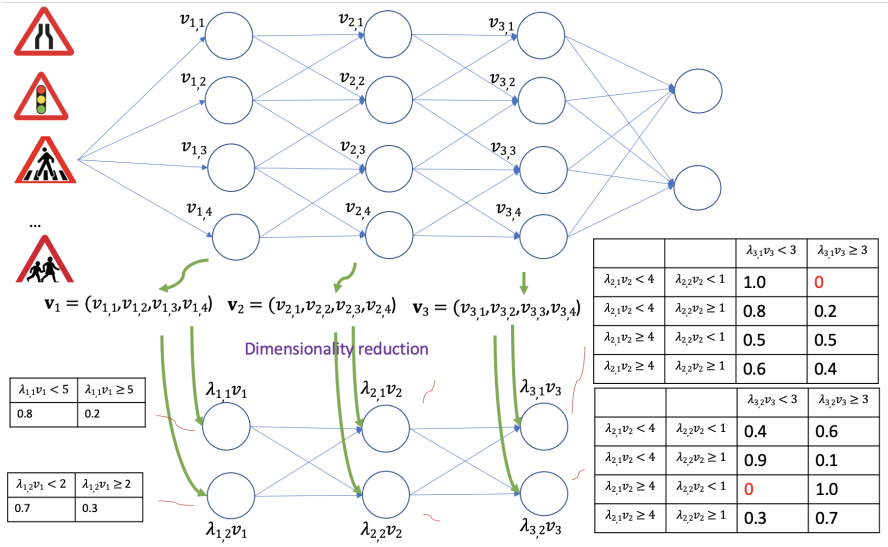
Actually, L_i is such that the neuron values $\mathbf{v}_i$ for any input $\mathbf{x} \in D_{train}$, can be transformed into a t_i-dimensional vector

$$\langle \lambda_{i,1}\mathbf{v}_i, \ldots, \lambda_{i,t_i}\mathbf{v}_i \rangle \in \mathbb{F}_i \tag{30.1}$$

where $\lambda_{i,j}\mathbf{v}_i$ represents the j-th component of the value obtained after mapping the instance $\mathbf{x}$ into the feature space. Figure 30.1 gives an illustrative diagram of reducing $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ to features. In particular, each $\mathbf{v}_i$ is reduced to two features $\lambda_{i,1}\mathbf{v}_i$ and $\lambda_{i,2}\mathbf{v}_i$.

30.2 Discretisation of Hidden Feature Space

The feature extraction techniques result in mappings $\lambda_{i,j}$ that range over a continuous and potentially infinite domain. Yet, Bayesian network-based abstraction technique relies on the construction of *probability tables*, where each entry associates a set of



$\lambda_{1,1}v_1 < 5$	$\lambda_{1,1}v_1 \geq 5$
0.8	0.2

$\lambda_{1,2}v_1 < 2$	$\lambda_{1,2}v_1 \geq 2$
0.7	0.3

		$\lambda_{3,1}v_3 < 3$	$\lambda_{3,1}v_3 \geq 3$
$\lambda_{2,1}v_2 < 4$	$\lambda_{2,2}v_2 < 1$	1.0	0
$\lambda_{2,1}v_2 < 4$	$\lambda_{2,2}v_2 \geq 1$	0.8	0.2
$\lambda_{2,1}v_2 \geq 4$	$\lambda_{2,2}v_2 < 1$	0.5	0.5
$\lambda_{2,1}v_2 \geq 4$	$\lambda_{2,2}v_2 \geq 1$	0.6	0.4

		$\lambda_{3,2}v_3 < 3$	$\lambda_{3,2}v_3 \geq 3$
$\lambda_{2,1}v_2 < 4$	$\lambda_{2,2}v_2 < 1$	0.4	0.6
$\lambda_{2,1}v_2 < 4$	$\lambda_{2,2}v_2 \geq 1$	0.9	0.1
$\lambda_{2,1}v_2 \geq 4$	$\lambda_{2,2}v_2 < 1$	0	1.0
$\lambda_{2,1}v_2 \geq 4$	$\lambda_{2,2}v_2 \geq 1$	0.3	0.7

Fig. 30.1: Reducing Neural Networks to Bayesian Networks

distinct latent feature values with a probability. For this construction to be relevant, we therefore *discretise* each latent feature component into a *finite* set of sub-spaces.

30.3 Construction of Bayesian Network Abstraction

The abstraction that we construct primarily represents the *probabilistic distribution* of the set of latent feature values induced by a set $\mathbf{X}$ of test instances. In other words, given an input $\mathbf{x} \in \mathbf{X}$, the abstraction allows us to estimate the probability that $\mathbf{x}$ induces a given combination of values for the latent features that have been learned by the neural network.

Thanks to the layered and acyclic nature of the neural networks that we consider, we can directly characterise the *causal relationship* between the sets of neuron values in various layers *w.r.t.* a series of inputs as well. In other words, given an input $\mathbf{x} \in \mathbf{X}$, one can in principle estimate the conditional probability of each neuron value at layer i *w.r.t.* the probability of every combination of neuron values at layer $i-1$. By lifting the above relationship from individual neuron values to latent feature intervals, we seek to capture *causal semantic relations* that link the features at each layer: in a layer i, and with an input $\mathbf{x}$, the *probability* that a latent feature valuation belongs to a given interval in the corresponding feature space is *dependent* on probabilities pertained to latent feature intervals at layer $i-1$.

30.4 Preserved Property

First of all, we show that the constructed Bayesian network is an abstraction of the neural network. Given a finite set $\mathbf{X}$ of inputs, an abstraction constructs a set $\mathbf{X}' \supseteq \mathbf{X}$ that generalises $\mathbf{X}$ to more elements [26]. Given an input $\mathbf{x}$ and a Bayesian network $\lambda(\mathbf{X})$, we are able to check the probability of $\mathbf{x}$ on $\lambda(\mathbf{X})$, i.e., $\lambda(\mathbf{X})(\mathbf{x})$. We say that $\mathbf{x}$ is included in $\lambda(\mathbf{X})$ if $\lambda(\mathbf{X})(\mathbf{x}) > 0$. The following lemma suggests that every sample in $\mathbf{X}$ is included in $\lambda(\mathbf{X})$:

Lemma 30.1 *All inputs $\boldsymbol{x}$ in the dataset $\boldsymbol{X}$ are included in the $\lambda(\boldsymbol{X})$ with probability greater than 0.*

Let $\mathbf{X}'$ be the set of inputs that satisfy $\lambda(\mathbf{X})(\mathbf{x}) > 0$. This lemma suggests that $\mathbf{X}' \supset \mathbf{X}$. Therefore, $\lambda(\mathbf{X})$ defines an abstraction of the dataset $\mathbf{X}$. This abstraction also suggests that the abstraction assumption – *i.e.,* inputs that are outliers *w.r.t.* the abstraction are also outliers *w.r.t.* the original neural network – is reasonable because $\mathbf{X}' \supset \mathbf{X}$.

In addition to this simple property, [5, 1] also consider other analysis techniques based on the abstracted Bayesian network.

Part VII
Looking Further

This part will discuss other topics related to the machine learning safety but have not been covered in other parts. This includes other deep learning models including deep reinforcement learning and other safety analysis techniques including testing techniques and safety assurance.

Chapter 31
Deep Reinforcement Learning

Unlike the convolutional neural network we introduced in Part III which relies on labelled data for supervised learning, reinforcement learning is a learning process in which the learning agent is trained through trial and error. Reinforcement learning is usually formalised as the finding of an optimal strategy in a Markov decision process (MDP). In other words, it is typical to model the interaction of an intelligent agent with its environment as an MDP, and then apply some algorithms (e.g., value iteration, policy iteration) to compute an optimal strategy for the intelligent agent.

To utilise traditional MDP algorithm, it is needed to have several components well defined, including the states, the actions, the transition relation, and the reward function. However, for a real-world application, such components may not be easily defined, for example, the transition relation may not be definable as it can be impossible to have a complete definition of the environment. Moreover, some components such as the states may be of very high dimensional, which will make the traditional MDP algorithms fail to work due to the time and memory limitations. To deal with these problems, deep reinforcement learning (DRL) combines reinforcement learning and deep learning to enable our working with unstructured, high-dimensional data without manual engineering of the components.

Another major difference from the convolutional neural network is the definition of safety properties. While for convolutional neural network the safety properties are closely related to the misclassification of individual input instances, for reinforcement learning the safety properties are more appropriate to be defined on the *sequential inputs*. In this chapter, we will reuse the verification tools for convolutional neural network to work with deep reinforcement learning.

We remark that, in this chapter, we assume that the reward function is well defined, and based on this, to consider the safety properties. Other important factors related to the definition of rewards, such as the side effect and reward hacking [2], are not considered.

31.1 Interaction of Agent with Environment

We use discounted infinite-horizon MDP to model the interaction of an agent with the environment E. An MDP is a 5-tuple $\mathcal{M}^E = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{P}(s'|s,a)$ is a probabilistic transition, $\mathcal{R}(s,a) \in \mathbb{R}_{\geq 0}$ is a reward function, and $\gamma \in [0,1)$ is a discount factor. A (deterministic) policy is $\pi: \mathcal{S} \rightarrow \mathcal{A}$ that maps from states to actions. We consider DDPG [47, 83, 53] for a reinforcement learning agent. DDPG returns a deterministic policy.

Based on $\mathcal{M}^E$, a policy π induces a trajectory distribution $\rho^{\pi,E}(\zeta)$ where $\zeta = (s_0, a_0, s_1, a_1, ...)$ denotes a random trajectory. The state-action value function of π is defined as $Q^\pi(s,a) = \mathbb{E}_{\zeta \sim \rho^{\pi,E}}[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t)]$ and the state value function of π is $V^\pi(s) = Q^\pi(s, \pi(s))$.

Example 31.1 We consider a reinforcement learning driven robot that navigates, and avoids collisions, in a complex environment where there are static and dynamic objects/obstacles. As stated in Section 31.1, the robot can be modelled as an MDP. At each time t, it has its observation of the laser sensors from the environment, namely state s_t, i.e.,

$$s_t = (o_t^1, o_t^2, \cdots, o_t^n)^T \tag{31.1}$$

where $o_t^1, o_t^2, \cdots, o_t^n$ are sensor signals at time t. As usual, the sensors can only scan the environment within a certain distance, for example, it is within 3.15 metres in Turtlebot Waffle Pi [68] for a distance sensor.

An action $a_t \in \mathcal{A}$ consists of several decision variables. With the PID controller on the robot, we consider two action variables, representing line velocity and angle velocity, respectively, i.e., $a_t = (v_t^{line}, v_t^{angle})^T$. At each time t, the DRL actor network outputs an action pair $(v_t^{line}, v_t^{angle})$ from the action set $\mathcal{A}$.

The objective of the robot is to avoid the obstacles and reach a goal area. On every state s_t, the sensory input o_t^i can be utilised to e.g., predict the distance to the obstacles and the goal area when they are close enough (within 3.15 metres). The environment imposes a reward function r on both the states (w.r.t. the distance to obstacles) and the actions (w.r.t. the acceleration in linear or angular speed).

31.2 Safety Properties through Probabilistic Computational Tree Logic

Probabilistic model checking [43] has been used to analyse quantitative properties of systems across a variety of application domains. It involves the construction of a probabilistic model, e.g., DTMC or MDP, that formally represents the behaviour of a system over time. The properties of interest are usually specified with, e.g., LTL or PCTL. Then, via model checkers, a systematic exploration and analysis is performed to check if a claimed property holds. In this paper, we adopt DTMC and PCTL whose

definitions are as follows. We remark that, the application of a policy π over an MDP induces a DTMC.

Definition 31.1 (DTMC) Let AP be a set of atomic propositions. A DTMC is a tuple $(S, s_0, \mathbf{P}, L)$, where S is a (finite) set of states, $s_0 \in S$ is an initial state, $\mathbf{P}\colon S \times S \to [0,1]$ is a probabilistic transition matrix such that $\sum_{s' \in S} \mathbf{P}(s,s') = 1$ for all $s \in S$, and $L\colon S \to 2^{AP}$ is a labelling function assigning each state with a set of atomic propositions.

Definition 31.2 (DTMC Reward Structure) A reward structure for DTMC $D = (S, s_0, \mathbf{P}, L)$ is a tuple $r = (r_S, r_T)$ where $r_S\colon S \to \mathbb{R}_{\geq 0}$ is a state reward function and $r_T\colon S \times S \to \mathbb{R}_{\geq 0}$ is a transition reward function.

Definition 31.3 (PCTL) The syntax of PCTL is defined by *state formulae* ϕ, *path formulae* ψ and *reward formulae* μ.

$$\begin{aligned}
\phi &::= true \mid ap \mid \phi \wedge \phi \mid \neg\phi \mid P_{\bowtie p}(\psi) \mid R^r_{\bowtie q}(\mu) \\
\psi &::= \phi \mid \phi\ U\ \phi \\
\mu &::= C^{\leq t} \mid \Diamond\, \phi
\end{aligned}$$

where $ap \in AP, p \in [0,1], q \in \mathbb{R}_{\geq 0}, t \in \mathbb{N}, \bowtie \in \{<, \leq, >, \geq\}$ and r is a reward structure. The temporal operator is called "next", and U is called "until". We write $\Diamond\, \phi$ for $true\, U\, \phi$, and call it "eventually". Operator $C^{\leq t}$ is "bounded cumulative reward", expressing the reward accumulated over t steps. Formula $R^r_{\bowtie q}(\Diamond\, \phi)$ expresses "reachability reward", the reward accumulated up until the first time a state satisfying ϕ.

Given $D = (S, s_0, \mathbf{P}, L)$ and $r = (r_S, r_T)$, the satisfaction of state formula ϕ on a state $s \in S$ is defined as:

$$\begin{aligned}
&s \models true; \quad s \models ap \Leftrightarrow ap \in L(s); \quad s \models \neg\phi \Leftrightarrow s \not\models \phi; \\
&s \models \phi_1 \wedge \phi_2 \Leftrightarrow s \models \phi_1 \text{ and } s \models \phi_2; \\
&s \models \mathcal{P}_{\bowtie p}(\psi) \Leftrightarrow Pr(s \models \psi) \bowtie p; \\
&s \models \mathcal{R}^r_{\bowtie q}(\mu) \Leftrightarrow \mathbb{E}[rew^r(\mu)] \bowtie q,
\end{aligned}$$

where $Pr(s \models \psi) \bowtie p$ concerns the probability of the set of paths satisfying ψ starting in s. Given a path η, if write $\eta[i]$ for its i-th state and $\eta[0]$ the initial state, then

$$rew^r(C^{\leq t})(\eta) = \sum_{j=0}^{k-1} (r_S(\eta[j]) + r_T(\eta[j], \eta[j+1]))$$

$$rew^r(\Diamond\phi)(\eta) = \begin{cases} \infty & \forall j \in \mathbb{N}(\eta[j] \not\models \phi) \\ rew^r(C^{\leq ind(\eta,\phi)})(\eta) & \text{otherwise} \end{cases}$$

where $ind(\eta, \phi) = \min\{j | \eta[j] \models \phi\}$ denotes the index of the first occurrence of ϕ on path η. Moreover, the satisfaction relations for a path formula ψ on a path η is defined as:

$$\eta \models \phi \Leftrightarrow \eta[1] \models \phi$$
$$\eta \models \phi_1 \, U \, \phi_2 \Leftrightarrow \exists j \geq 0 (\eta[j] \models \phi_2 \wedge \forall k < j (\eta[k] \models \phi_1))$$

Very often, it is of interest to know the actual probability that a path formula is satisfied, rather than just whether or not the probability meets a required threshold since this can provide a notion of margins as well as benchmarks for comparisons following later updates. So, the PCTL definition can be extended to allow *numerical queries* of the form $\mathcal{P}_{=?}(\psi)$ or $\mathcal{R}^r_{=?}(\psi)$ [43]. After formalising the system behaviors and properties in DTMC and PCTL respectively, automated tools have been developed to solve the verification problem, e.g., PRISM [42] and STORM [13].

We remark that, PCTL can be utilised to describe safety-related properties for e.g., the robot navigation example Example 31.1 as discussed in [15].

31.3 Training a DRL agent

The model-free DDPG algorithm [47] is applied for the training of a DRL policy for the robot. Typically, the DRL policies are trained in a simulation environment before applied to the real world [11]. That is because of the unbearable costs of having real-world (negative) examples for training in real world [97]. The objective of the DDPG algorithm for an intelligent agent is to learn a policy: $\pi: \mathcal{S} \rightarrow \mathcal{A}$, which maximises the expectation of the best reward over time N:

$$J = \mathbb{E}(\sum_{t=0}^{N-1} \gamma^t r_t | s_t = s_0) \tag{31.2}$$

where s_0 is the initial state; r_t is the reward based on state-action pair at time slot t; γ is the discount factor which is applied to reduce the effect of future reward. To simplify the notation, we let $G_t = \sum_{t=0}^{N-1} \gamma^t r_t$.

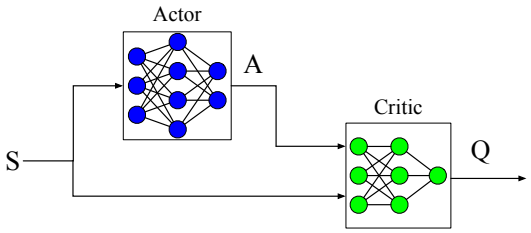


Fig. 31.1: Structure of DDPG.

The DDPG algorithm has two different neural networks, actor networks $\mu(s_t|\theta^\mu)$ and critic networks $Q(s_t, a_t|\theta^Q)$, which are illustrated in Fig. 31.1. θ^μ and θ^Q are the weights of the actor and critic network, respectively. Due to the non-linearity

of the neural networks, the DDPG algorithm can deal with the continues states and continues actions, which are more realistic to autonomous systems. Actor network is used to yield a deterministic action value a_t, it takes the observation of environment as the inputs, and outputs the decided actions of the robot. Critic network is used to approximate Q value, which is use to judge the state-action pair is good or not. It takes the observation from environment and the action value from actor networks, and then outputs the approximated $Q(s_t, a_t)$ values. In the algorithm, the critic network is trained to minimise the loss function based on the stochastic gradient descent [47]:

$$\mathcal{L}(\theta^Q) = \mathbb{E}[(y_t - Q(s_t, a_t|\theta^Q))^2] \tag{31.3}$$

where $y_t = r_t(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_t|\theta^\mu)|\theta^Q)$ is the approximated Q value based on current state s_t and previous parameters θ^μ, θ^Q of two neural networks. The actor network is updated by the policy gradient with following equation [47]:

$$\nabla_{\theta^\mu} J^{\theta^\mu} = \mathbb{E}[\nabla_a Q(s, a|\theta^Q)|_{a=\mu(s|\theta^\mu)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)] \tag{31.4}$$

To breaking harmful correlations and learn more from individual tuples multiple times, an experience replay buffer is applied.

31.4 Verification

Although at any specific time a DRL agent – like the classifier – also returns an action according to the state, the correctness of the action is not solely dependent on the state. Instead, considering its training mechanism, the correctness of the action at any specific time depends on the expected long-term accumulated rewards. For this reason, the verification of a DRL agent also needs to consider not only the current state but also the long-term rewards (and therefore the future states).

Considering a typical DRL agent which has two neural networks $f_1 : \mathcal{S} \to \mathcal{A}$ and $f_2 : \mathcal{S} \times \mathcal{A} \to \mathcal{N}$, representing the actor and the critic agents, respectively. Intuitively, $f_1(s)$ returns the action a that needs to be taken on a state s, while $f_2(s, a)$ returns the expected rewards of taking action a on state s. Assume that we have a verification tool g that, given f_1 and a constraint $C_\mathcal{S}$ in $\mathcal{S}$, outputs an (over-approximated) reachable set in $\mathcal{A}$. That is, we can have

$$g(f_1, C_\mathcal{S}) \subseteq \mathcal{A} \tag{31.5}$$

as the reachable set of actions when given a set of states expressed with constraint $C_\mathcal{S}$. Such verification tool is available in our previous works [71, 93, 46].

In the following, depending on whether learning an environment agent is possible, we have two different verification approaches.

Learning Environment Agent

To conduct verification, we may have an environment agent $E: \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{P}(\mathcal{S})$, which returns a distribution of next states when given an action on the current state. Such environment can be learned as in [21]. Similar as Equation (31.5), we have

$$g(E, C_{\mathcal{S}} \times C_{\mathcal{A}}) \tag{31.6}$$

as the reachable set of next states when given a set of states expressed with $C_{\mathcal{S}}$ and a set of actions expressed with $C_{\mathcal{A}}$.

Based on the above, once we have E and f_1, and a set $C_{\mathcal{S}}$ of constraints on the current states, we can compute

$$C'_{\mathcal{S}} = g(E, C_{\mathcal{S}} \times g(f_1, C_{\mathcal{S}})) \tag{31.7}$$

as the set of reachable next states. This step can be repeated until obtaining $C_{\mathcal{S}}^{(k)}$ after $k > 1$ steps. We can then check whether $(C_{\mathcal{S}}, C_{\mathcal{S}}^{(k)})$ satisfies the constraint C as defined with e.g., PCTL formulas.

Splitting Constraints

When learning an environment agent is infeasible, we can consider an alternative approach. We need two supplementary functions. The first function h_1 is, given a constraint $C_{\mathcal{S}}$, to split $C_{\mathcal{S}}$ into a set of constraints, i.e.,

$$h_1(C_{\mathcal{S}}, g, f_1) = \{C_{\mathcal{S}}^{a_1}, ..., C_{\mathcal{S}}^{a_m}\} \tag{31.8}$$

such that $C_{\mathcal{S}} = C_{\mathcal{S}}^{a_1} \cup ... \cup C_{\mathcal{S}}^{a_m}$ and, for all $1 \leq i \leq m$, $g(f_1, C_{\mathcal{S}}^{a_i}) = \{a_i\}$. Intuitively, after splitting, all states in a constraint $C_{\mathcal{S}}^{a_i}$ take the same action a_i. We write $a(C_{\mathcal{S}}^{a_i}) = a_i$. The second function h_2 is, given constraints $C_{\mathcal{S}}$ and an action a, to return an overapproximation to the next states, i.e.,

$$h_2(C_{\mathcal{S}}, a) \supset \bigcup_{s \models C_{\mathcal{S}}, s \in \mathcal{S}} support(E(s, a)) \tag{31.9}$$

where $s \models C_{\mathcal{S}}$ means that the state s satisfies the constraint $C_{\mathcal{S}}$, and $support(D)$ returns the support of a distribution D.

Now, given a constraint $C_{\mathcal{S}}$ on the current states, we determine the safety of $C_{\mathcal{S}}$, written as $C_{\mathcal{S}} \models C$, with

$$\bigwedge_{C \in h_1(C_{\mathcal{S}}, g, f_1)} h_2(C, a(C)) \models C \tag{31.10}$$

Note that, Equation (31.10) is a recursive definition, i.e., $h_2(C, a(C)) \models C$ can also be computed in this way. By recursively computing Equation (31.10) for k steps, we can have the verification result.

We remark that, h_1 can be obtained by applying binary search together with the verification tool g, and h_2 can be obtained by a genetic algorithm to search for a convex hull over the next states.

Chapter 32
Testing Techniques

Verification techniques, as discussed in Part III and Chapter 31, are to ascertain – with mathematical proof – whether a property holds on a mathematical model. The soundness and completeness required by the mathematical proof results in the scalability problem that verification algorithms can only work with either small models (e.g., the MILP-based method as in Chapter 21) or limited number of input dimensions (e.g., the reachability analysis as in Chapter 22). In practice, when working with real-world systems where the machine learning models are large in nature, other techniques have to be considered for the certification purpose.

Similar to traditional software testing against software verification, neural network testing provides a certification methodology with a balance between completeness and efficiency. In established industries, e.g., avionics and automotive, the needs for software testing have been settled in various standards such as DO-178C and MISRA. However, due to the lack of logical structures and system specification, it is less straightforward on how to extend such standards to work with systems with neural network components. In the following, we discuss some existing neural network testing techniques. The readers are referred to the survey [31] for more discussion.

32.1 A General Testing Framework

Assume that according to the model f and a safety property ϕ, we are able to define a set of test objectives $\mathcal{R}$. We will discuss later in Section 32.2 a few existing covering methods cov to define $\mathcal{R}$ for convolutional neural networks.

Definition 32.1 (Test Suite) Given a neural network f, a test suite $\mathcal{T}$ is a finite set of input instances, i.e., $\mathcal{T} \subseteq \mathcal{D}$. Each instance is called a test case.

Ideally, given the set of test objectives $\mathcal{R}$ with respect to some covering method cov, we run a test case generation algorithm (to be introduced in Section 32.3) to find a test suite $\mathcal{T}$ such that

$$\forall \alpha \in \mathcal{R} \exists \mathbf{x} \in \mathcal{T} : cov(\alpha, \mathbf{x}) \tag{32.1}$$

where $cov(\alpha, \mathbf{x})$ intuitively means that the test objective α is satisfied under the test case $\mathbf{x}$. Intuitively, Equation (32.1) means that every test objective is covered by some of the test cases. In practice, we might want to compute the degree to which the test objectives are satisfied by a test suite $\mathcal{T}$.

Definition 32.2 (Test Criterion) Given a neural network f, a covering method cov, a set $\mathcal{R}$ of test objectives, and a test suite $\mathcal{T}$, the test criterion $M_{cov}(\mathcal{R}, \mathcal{T})$ is as follows:

$$M_{cov}(\mathcal{R}, \mathcal{T}) = \frac{|\{\alpha \in \mathcal{R} | \exists \mathbf{x} \in \mathcal{T} : cov(\alpha, \mathbf{x})\}|}{|\mathcal{R}|} \tag{32.2}$$

Intuitively, it computes the percentage of the test objectives that are covered by test cases in $\mathcal{T}$ w.r.t. the covering method cov.

32.2 Coverage Metrics for Neural Networks

Research in software engineering has resulted in a broad range of approaches to testing software. Please refer to [103, 34, 78] for comprehensive reviews. In white-box testing, the structure of a program is exploited to (perhaps automatically) generate test cases. Structural coverage criteria (or metrics) define a set of test objectives to be covered, guiding the generation of test cases and evaluating the completeness of a test suite. E.g., a test suite with 100% statement coverage exercises all statements of the program at least once. While it is arguable whether this ensures functional correctness, high coverage is able to increase users' confidence (or trust) in the testing results [103]. Structural coverage analysis and testing are also used as a means of assessment in a number of safety-critical scenarios, and criteria such as statement and modified condition/decision coverage (MC/DC) are applicable measures with respect to different criticality levels. MC/DC was developed by NASA[23] and has been widely adopted. It is used in avionics software development guidance to ensure adequate testing of applications with the highest criticality [70].

Let $\mathcal{R}$ be the set of test objectives to be covered. For different structure coverage, we can define different sets of test objectives. In [63], $\mathcal{R}$ is instantiated as the set of statuses of hidden neurons. That is, for the set $\mathcal{H}_i$ of hidden neurons with ReLU activation functions at layer i, we let

$$\mathcal{R}_{\text{neuron coverage}} = \bigcup_{i=2}^{K-1} \mathcal{H}_i \tag{32.3}$$

Intuitively, combining with the general testing framework in Section 32.1, it requires to find a test suite $\mathcal{T}$ such that for every hidden ReLU neuron, there is a test case $t \in \mathcal{T}$ who can activate it. As another example, in [80], $\mathcal{R}$ is instantiated as the set of causal relationships between feature pairs. That is,

$$\mathcal{R}_{\text{MC/DC coverage}} = \bigcup_{i=2}^{K-2} \{(h_1, h_2) \mid h_1 \in \mathcal{H}_i, h_2 \in \mathcal{H}_{i+1}\} \tag{32.4}$$

Intuitively, combining with the general testing framework in Section 32.1, it requires to find a number of test pairs such that each $(h_1, h_2) \in \mathcal{R}_{\text{MC/DC coverage}}$, h_2 can be independently activated by h_1 [79].

32.3 Test Case Generation

Once a coverage metric is determined, it is needed to develop a test case generation algorithm to produce a set of test cases. Existing methods include e.g., input mutation [91], fuzzing [58], genetic algorithm [29], symbolic execution [81], and gradient ascent [82].

32.4 Discussion

In addition to testing techniques which originate from the software engineering area, there are other techniques that might be useful to analyse the safety of neural networks. Statistical evaluation applies statistical methods in order to gain insights into the verification problem we concern. In addition to the purpose of determining the existence of failures in the deep learning model, statistical evaluation assesses the satisfiability of a property in a probabilistic way, by e.g., aggregating sampling results. The aggregated evaluation result may have probabilistic guarantee, in the form of e.g., the probability of failure rate lower than a threshold l is greater than $1 - \epsilon$, for some small constant ϵ.

For the robustness, sampling methods, such as [90], are to summarise property-related statistics from the samples. While sampling methods can have probabilistic guarantees via e.g., Chebyshev's inequality, it is still under investigation on how to associate test coverage metrics with probabilistic guarantee. For the generalisation error, other than the empirical approach of using a set of test data to evaluate, recent efforts on complexity measure [10, 35] suggest that it is possible to estimate generalisation error – with theoretical bound – by only considering the weights of the deep learning without resorting to the test dataset.

Chapter 33
Safety Assurance

In Part III, Chapter 31, and Chapter 32, we have introduced verification and testing techniques for convolutional neural network (CNN) and deep reinforcement learning (DRL). These techniques are to work with individual safety properties, such as the robustness of a data instance $\mathbf{x}$ (for CNN) or the safety of a policy π on a given initial state s_0 (for DRL). However, considering that deep learning models usually serve as components of a large autonomous system, the safety of the deep learning models is related to how it is used in the system. For example, if a CNN is used as a perception component for e.g., object detection in a self-driving car, there will be a set D_{op} of data instances that may appear in operational time, all of which needs to be verified. If a DRL agent is used as a control component for e.g., navigation in a mobile robot, there will be a set of possible trajectories that may appear in operational time, all of which needs to be verified.

In this chapter, we introduce a principled approach to utilise evidence produced by verification techniques about low-level safety properties (e.g., robustness of individual instances) to reason about high-level safety claims, such as "the perception component of the self-driving car can correctly classify the next 1,000 instances with probability higher than 99%". These high-level safety claims are required in various industrial standards for software used in safety-critical systems. Methodologically, the approach is based on the safety argument, which provides a link between the safety evidence and a safety claim, showing that the safety evidence is sufficient to support the claim.

33.1 Reliability Assessment for Convolutional Neural Networks

Assume that, we are working with a verification technique g, which, given a network f and a constraint C, returns the probability of the inputs within C being classified correctly. The constraint C can be e.g., a norm ball with $\mathbf{x}$ as the centre to denote the possible perturbations.

Also, as mentioned above, we assume that there are a set D_{op} of operational data instances. We partition the input domain $\mathcal{D}$ into m cells, subject to the r-separation

property [66]. These cells are disjoint and altogether form the entire input domain. Let p_{op} be the empirical distribution of the cells estimated with the dataset D_{op}. Then, we can learn a generative model G_θ over parameters θ such that

$$\theta^* = \arg\min_\theta \mathrm{KL}(G_\theta, p_{op}) \tag{33.1}$$

where $\mathrm{KL}(\cdot,\cdot)$ is the KL divergence between two distributions.

Based on the above, we can estimate the reliability (defined as the probability of failure in classifying the next input) as

$$\text{Reliability}(f) = \sum_{i=1}^{m} G_\theta(C_i)(1 - g(f, C_i)) \tag{33.2}$$

Intuitively, $G_\theta(C_i)$ returns the probability density of the cell i represented as the constraint C_i, and $1 - g(f, C_i)$ returns the failure rate of the neural network f working on inputs satisfying the constraint C_i.

[99, 101] show how to develop a principled safety argument to justify the reliability claim by aggregating evidence from either formal verification or statistical evaluation with Bayesian inference [77]. Other than learning a generative model, there are other methods to learn the distribution of cells [100].

33.2 Reliability Assessment for Deep Reinforcement Learning

Because DRL is working on the trajectories, we assume a verification technique g', as discussed in Section 31.4, that, given a policy π and an initial state s, returns whether it is safe. We can also assume that g' returns the probability of safety, because a Boolean answer can also be converted into a Dirac probability.

Similar as Section 33.1, we partition the set of possible initial states into m subsets, each of which is represented as a constraint C_i, for $i = 1..m$. Then, we can also define the empirical distribution p_{op} over the partitions, and get the model G_θ. Based on these, we can estimate the reliability (defined as the probability of failure in classifying the next input) as

$$\text{Reliability}(f) = \sum_{i=1}^{m} G_\theta(C_i)(1 - g'(E, \pi, C_i)) \tag{33.3}$$

where $G_\theta(C_i)$ returns the probability density of the partition i represented as the constraint C_i, and $1 - g'(f, \pi, C_i)$ returns the failure rate of the DRL agent f working on inputs satisfying the constraint C_i under the environment E.

Part VIII
Mathematical Foundations

Appendix A
Foundation of Probability Theory

A.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 A.1 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

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

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 A.2 The $Grade$ random variable in Example A.1 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 A.3 For the **iris** example as in Example 1.1, 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.

A.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 A.4 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 A.5 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 A.6 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 A.7 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 $2/26 = 1/13$.

A.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

$$X \perp Y \tag{A.2}$$

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

Example A.8 Consider the joint probability table as shown in Table A.1.

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

Table A.1: A simple two variable joint distribution

First of all, we note that

$$\begin{aligned} P(X = 0) &= 0.32 + 0.8 = 0.4 \\ P(X = 1) &= 0.6 \\ P(Y = 0) &= 0.08 + 0.12 = 0.2 \\ P(Y = 1) &= 0.8 \end{aligned} \tag{A.3}$$

Then, we notice that

$$\begin{aligned} 0.08 &= P(X = 0, Y = 0) = P(X = 0) * P(Y = 0) = 0.4 * 0.2 \\ 0.32 &= P(X = 0, Y = 1) = P(X = 0) * P(Y = 1) = 0.4 * 0.8 \\ 0.12 &= P(X = 1, Y = 0) = P(X = 1) * P(Y = 0) = 0.6 * 0.2 \\ 0.48 &= P(X = 1, Y = 1) = P(X = 1) * P(Y = 1) = 0.6 * 0.8 \end{aligned} \tag{A.4}$$

that is,

$$P(X, Y) = P(X) * P(Y) \tag{A.5}$$

which suggests that X and Y are independent.

Example A.9 On the other hand, for the following Table A.2, 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 A.2: 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 Z, denoted as

$$X \perp Y|Z \tag{A.6}$$

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 A.8 and A.9 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 $P(Y|Z)$.

A.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

$$P(X_{i1}, ..., X_{ik}|X_{j1} = x_{j1}, ..., X_{jl} = x_{jl}) \tag{A.7}$$

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

$$\begin{aligned} & MAP(X_1, ..., X_k | X_{l+1}, ..., X_n) \\ = & \arg\max_{x_1,...,x_k} \sum_{x_{k+1},...,x_l} P(X_1 = x_1, ..., X_l = x_l | X_{l+1} = x_{l+1}, ..., X_n = x_n) \end{aligned} \tag{A.8}$$

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 A.10 Consider a probability table for variable X_1 as in Table A.3.

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

Table A.3: Probability table for X_1

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

Example A.11 Consider a joint probability table for variables X_1 and X_2 as in Table A.4.

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 A.4: Joint probability table for X_1 and X_2

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

Appendix B
Linear algebra

B.1 Scalars, Vectors, Matrices, Tensors

Scalar

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

Example B.1 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.

B.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

$$\mathbf{A}^T_{ij} = \mathbf{A}_{ji} \text{ for all } i, j \tag{B.1}$$

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

$$T(\mathbf{x}) = \mathbf{A}\mathbf{x} \tag{B.2}$$

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

$$\mathbf{AB} = \mathbf{BA} = \mathbf{I} \tag{B.3}$$

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

B.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):

$$||\mathbf{x}||_1 = \sum_{i=1}^{n} |x_i| \tag{B.4}$$

- L_2 (Euclidean distance):

$$||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{n} x_i^2} \tag{B.5}$$

- L_∞ (Chebyshev distance):

$$||\mathbf{x}||_\infty = \max_i |x_i| \tag{B.6}$$

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 [27].

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

Definition B.1 (d-Neighbourhood) Given a data instance $\mathbf{x}$, a distance function L_p, and a distance d, we define the *d-neighbourhood* $\eta(\mathbf{x}, L_p, d)$ of $\mathbf{x}$ w.r.t. L_p as

$$\eta(\mathbf{x}, L_p, d) = \{\hat{\mathbf{x}} \mid ||\hat{\mathbf{x}} - \mathbf{x}||_p \leq d\}, \tag{B.7}$$

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

B.4 Variance, Covariance, and Covariance Matrix

Variance

measures the variation of a single random variable. Formally,

$$Var(X) = \mathbb{E}[(X - \overline{X})^2] \tag{B.8}$$

where $\overline{X} = \mathbb{E}[X]$ is the mean. If we are working with a finite set D of data samples, we have

$$\overline{X} = \frac{1}{|D|} \sum_{\mathbf{x} \in D} \mathbf{x} \qquad\qquad Var(X) = \frac{1}{|D|} \sum_{\mathbf{x} \in D} (\mathbf{x} - \bar{X})^2 \tag{B.9}$$

where $|D|$ denotes the number of samples in D.

Covariance

, based on variance, measures how much two random variables vary together. Formally,

$$Cov(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] \tag{B.10}$$

If working with a dataset D of $(\mathbf{x}, \mathbf{y})$ pairs, it is

$$Cov(X, Y) = \frac{1}{|D|} \sum_{(\mathbf{x}, \mathbf{y}) \in D} (\mathbf{x} - \bar{X})(\mathbf{y} - \bar{Y}) \tag{B.11}$$

Covariance Matrix

Generalising the above, consider a column vector $\mathbf{X} = (X_1, ..., X_n)^T$ of random variables, we can have the covariance matrix as follows:

$$\mathbf{K} = \begin{bmatrix} Cov(X_1, X_1) & Cov(X_1, X_2) & Cov(X_1, X_3) & \dots & Cov(X_1, X_n) \\ Cov(X_2, X_1) & Cov(X_2, X_2) & Cov(X_2, X_3) & \dots & Cov(X_2, X_n) \\ \dots & \dots & \dots & \dots & \dots \\ Cov(X_n, X_1) & Cov(X_n, X_2) & Cov(X_n, X_3) & \dots & Cov(X_n, X_n) \end{bmatrix} \tag{B.12}$$

Part IX
Competitions

Appendix C
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.

C.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.

C.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
import copy
```

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"
    ####################
```

```
33 #
       ##########################################################################
```

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.

```
1  'generate adversarial data, you can define your adversarial
       method'
2  def adv_attack(model, X, y, device):
3      X_adv = Variable(X.data)
4
5      #
       ##########################################################################
6      ## Note: below is the place you need to edit to implement
       your own attack algorithm
7      #
       ##########################################################################
8
9      random_noise = torch.FloatTensor(*X_adv.shape).uniform_(-0.1,
        0.1).to(device)
10     X_adv = Variable(X_adv.data + random_noise)
11
12     #
       ##########################################################################
13     ## end of attack method
14     #
       ##########################################################################
15
16     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.

```
1  '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.nll_loss(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.nll_loss(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.nll_loss(model(adv_data), target)
        loss = F.nll_loss(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))

    adv_tstloss, adv_tstacc = eval_adv_test(model, device, test_loader)
```

```
    print('Your estimated attack ability, by applying your attack
     method on your own trained model, is: {:.4f}'.format(1/
    adv_tstacc))
    print('Your estimated defence ability, by evaluating your own
     defence model over your attack, is: {:.4f}'.format(
    adv_tstacc))
    ############################################################
    ## 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)
        data_ = copy.deepcopy(data.data)
        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)
```

C.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.

C.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

$$\textbf{Score} = \begin{array}{c} \textbf{i=1} \\ \textbf{i=2} \\ \cdots \\ \textbf{i=n-1} \\ \textbf{i=n} \end{array} \begin{pmatrix} s_{11} & s_{12} & \cdots & s_{1(n-1)} & s_{1n} \\ s_{21} & s_{22} & \cdots & s_{2(n-1)} & s_{2n} \\ & & & & \\ s_{(n-1)1} & s_{(n-1)2} & \cdots & s_{(n-1)(n-1)} & s_{(n-1)n} \\ s_{n1} & s_{n2} & \cdots & s_{n(n-1)} & s_{nn} \end{pmatrix} \begin{array}{c} \\ \\ \\ \\ \\ \end{array} \\ \begin{array}{ccccc} \textbf{j=1} & \textbf{j=2} & \cdots & \textbf{j=n-1} & \textbf{j=n} \end{array} \tag{C.1}$$

for the mutual evaluation scores of using M_i to evaluate Atk_j (defined in function **adv_attack**). The score s_{ij} is the **test accuracy** obtained by using **adv_attack** function from the file competition $\_j$**.py** to attack the model from i**.pt**. From Equation (C.1), we get j's attacking ability by letting

$$AttackAbility_j = \sum_{i=1}^{n} \textbf{Score}_{i,j} \tag{C.2}$$

to be the total of the scores of j-th column. Let **AttackAbility** be the vector of $AttackAbility_j$. Moreover, we get i's defence ability by letting

$$DefenceAbility_i = \sum_{j=1}^{n} \textbf{Score}_{i,j}, \tag{C.3}$$

Let **DefenceAbility** be the vector of $DefenceAbility_i$. Then, for the vectors **AttackAbility** and **DefenceAbility**, we apply Softmax function to normalise to get

$$\sigma(\frac{1}{\textbf{AttackAbility}}), \sigma(\textbf{DefenceAbility}) \tag{C.4}$$

Then, the final score for the submission i is

$$FinalScore_i = \sigma(\sigma(\frac{1}{\textbf{AttackAbility}}) + \sigma(\textbf{DefenceAbility}))_i \tag{C.5}$$

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.

C.5 Q&A

Will the running time, memory, and CPU usage be considered in marking for the training and attack?

We only evaluate against a trained model, so no consideration will be given on the running time, memory, and CPU usage in training. For the attack, it will run on

marker's computer, so we expect it to run in a reasonable time, e.g., within 1 min for a single image.

Can we use external pip packages for purposes like hyperparameter tuning?

You can use external libraries as long as they are consistent with the libraries we suggested for this module, and can be installed easily through either pip or annaconda.

You can also use package for e.g., attack, but you have to make sure that the package will run well in normal circumstance (so that the markers can run the package).

Would it be OK to search for popular algorithm and then try to fix that to my submission?

Yes, you can. Actually, you are free to take – and adapt – any existing algorithms/implementations – this is also a skill that is nowadays quite useful in ML field. This will probably be the case for most people.

How would you make sure the competition is fair? I am worrying about that some submissions may try to overfit the test dataset.

The test dataset is not open, so nobody has access to the test dataset before competition.

Architectural search was mentioned in today's Q&A session, but I think it may be unhelpful since our target network architecture is fixed. The only thing I could come up with is to use Dropouts to deactivate some nodes to achieve a pseudo different net structure?

Architecture search might not be the most suitable technique to consider here – I mentioned it simply because there was a question I was asked.

Automated hyperparameter tunning was also mentioned in today's Q&A. Also, you mentioned several sessions ago that we should make sure to use some widely used reliable libraries. Given that I am using ray tune currently, do you have any recommendations on this?

It might be OK to try, if you want to exercise on hyper-parameter tuning. I do not have personal preference on the libraries, and you can select whichever you feel OK with. Just to remind you: for the submission, we only look at the trained weights, so will not be able to consider how you tune the parameters to get the weights.

How will the score of attack/defense be measured? Do we use loss or accuracy or loss & accuracy to evaluate the result?

Please refer to the $p\_distance$ function of the template code for a good understanding of this question.

How could you deal with the connection problem when downloading dataset through python IDLE?

Please use CMD (windows) or Terminal (mac) to download the dataset.

Glossary

$\boldsymbol{f}$ Machine Learning Model

$\boldsymbol{D_{train}}$ Training Dataset

$\boldsymbol{D_{test}}$ Test Dataset

$\mathcal{D}$ Input Domain

$\boldsymbol{C}$ A set of labels

$\mathbf{x}$ A data instance

$\boldsymbol{X_i}$ The i-th feature of data instance

$\boldsymbol{y}$ A scalar label of the data instance $\mathbf{x}$

$\hat{\boldsymbol{y}}$ The predictive label of the data instance $\mathbf{x}$ by a classifier

$\mathcal{L}$ Loss function

$\boldsymbol{h}$ Ground-truth function

Index

References

References

1. Amany Alshareef, Nicolas Berthier, Sven Schewe, and Xiaowei Huang. Quantifying the importance of latent features in neural networks. In *SafeAI2022*, 2022.
2. Dario Amodei, Chris Olah, Jacob Steinhardt, Paul F. Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *CoRR*, abs/1606.06565, 2016.
3. Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *arXiv preprint arXiv:1802.00420*, 2018.
4. Yang Bai, Yan Feng, Yisen Wang, Tao Dai, Shu-Tao Xia, and Yong Jiang. Hilbert-based generative defense for adversarial examples. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4784–4793, 2019.
5. 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.
6. Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 387–402. Springer, 2013.
7. Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 3–14. ACM, 2017.
8. Nicholas Carlini and David Wagner. Magnet and" efficient defenses against adversarial attacks" are not robust to adversarial examples. *arXiv preprint arXiv:1711.08478*, 2017.
9. Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP), IEEE Symposium on*, pages 39–57, 2017.
10. Niladri S Chatterji, Behnam Neyshabur, and Hanie Sedghi. The intriguing role of module criticality in the generalization of deep networks. *ICLR2020*, 2020.
11. Paul Christiano, Zain Shah, Igor Mordatch, Jonas Schneider, Trevor Blackwell, Joshua Tobin, Pieter Abbeel, and Wojciech Zaremba. Transfer from simulation to real world through learning deep inverse dynamics model. *arXiv preprint arXiv:1610.03518*, 2016.
12. Jacson Rodrigues Correia da Silva, Rodrigo Ferreira Berriel, Claudine Badue, Alberto Ferreira de Souza, and Thiago Oliveira-Santos. Copycat CNN: stealing knowledge by persuading confession with random non-labeled data. In *2018 International Joint Conference on Neural Networks, IJCNN 2018, Rio de Janeiro, Brazil, July 8-13, 2018*, pages 1–8. IEEE, 2018.
13. Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A Storm is coming: A modern probabilistic model checker. In Rupak Majumdar and Viktor Kunčak,

editors, *Computer Aided Verification*, volume 10427 of *LNCS*, pages 592–600, Cham, 2017. Springer.
14. Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 321–338, Santa Clara, CA, August 2019. USENIX Association.
15. Yi Dong, Xingyu Zhao, and Xiaowei Huang. Dependability analysis of deep reinforcement learning based robotics and autonomous systems. *CoRR*, abs/2109.06523, 2021.
16. Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.
17. Gintare Karolina Dziugaite and Daniel M Roy. Computing nonvacuous generalization bounds for deep (stochastic) neural networks with many more parameters than training data. *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2017.
18. Logan Engstrom, Dimitris Tsipras, Ludwig Schmidt, and Aleksander Madry. A rotation and a translation suffice: Fooling cnns with simple transformations. *arXiv preprint arXiv:1712.02779*, 2017.
19. Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *CoRR*, abs/1412.6572, 2014.
20. Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
21. David Ha and Jürgen Schmidhuber. World models. *CoRR*, abs/1803.10122, 2018.
22. Jamie Hayes and George Danezis. Learning universal adversarial perturbations with generative models. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 43–49. IEEE, 2018.
23. Kelly Hayhurst, Dan Veerhusen, John Chilenski, and Leanna Rierson. A practical tutorial on modified condition/decision coverage. Technical report, NASA, 2001.
24. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
25. Dan Hendrycks and Kevin Gimpel. Early methods for detecting adversarial images. *arXiv preprint arXiv:1608.00530*, 2016.
26. Thomas A. Henzinger, Anna Lukina, and Christian Schilling. Outside the box: Abstraction-based monitoring of neural networks. *CoRR*, abs/1911.09032, 2019.
27. 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.
28. W. Ronny Huang, Jonas Geiping, Liam Fowl, Gavin Taylor, and Tom Goldstein. Metapoison: Practical general-purpose clean-label data poisoning. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 12080–12091. Curran Associates, Inc., 2020.
29. Wei Huang, Youcheng Sun, Xingyu Zhao, James Sharp, Wenjie Ruan, Jie Meng, and Xiaowei Huang. Coverage-guided testing for recurrent neural networks. *IEEE Transactions on Reliability*, pages 1–16, 2021.
30. Wei Huang, Xingyu Zhao, and Xiaowei Huang. Embedding and extraction of knowledge in tree ensemble classifiers. *Machine Learning*, 2021.
31. Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. *Computer Science Review*, 37:100270, 2020.
32. Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *International Conference on Computer Aided Verification*, pages 3–29. Springer, 2017.
33. Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15–17, 1976.
34. Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

35. Gaojie Jin, Xinping Yi, Liang Zhang, Lijun Zhang, Sven Schewe, and Xiaowei Huang. How does weight correlation affect the generalisation ability of deep neural networks. In *NeurIPS'20*, 2020.
36. Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *European Conference on Computer Vision*, pages 694–711. Springer, 2016.
37. Alex Kantchelian, J. D. Tygar, and Anthony D. Joseph. Evasion and hardening of tree ensemble classifiers. In *Proceedings of the 33nd International Conference on Machine Learning*, volume 48, pages 2387–2396, 2016.
38. Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
39. S. Kullback and R.A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, 1951.
40. Alexey Kurakin, Ian Goodfellow, Samy Bengio, et al. Adversarial examples in the physical world, 2016.
41. Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *CoRR*, abs/1607.02533, 2016.
42. Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *LNCS*, pages 585–591, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
43. Marta Kwiatkowska, Gethin Norman, and David Parker. Probabilistic Model Checking: Advances and Applications. In Rolf Drechsler, editor, *Formal System Verification: State-of the-Art and Future Trends*, pages 73–121. Springer, Cham, 2018.
44. 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.
45. Melanie Lefkowitz. Professor's perceptron paved the way for ai – 60 years too soon, 2019.
46. Jianlin Li, Jiangchao Liu, Pengfei Yang, Liqian Chen, Xiaowei Huang, and Lijun Zhang. Analyzing deep neural networks with symbolic propagation: Towards higher precision and faster verification. In *SAS2019*, 2019.
47. Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *ICLR'16*, 2016.
48. Xingjun Ma, Bo Li, Yisen Wang, Sarah M Erfani, Sudanthi Wijewickrema, Michael E Houle, Grant Schoenebeck, Dawn Song, and James Bailey. Characterizing adversarial subspaces using local intrinsic dimensionality. *arXiv preprint arXiv:1801.02613*, 2018.
49. Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.
50. David A McAllester. PAC-bayesian model averaging. In *Proceedings of the twelfth annual conference on Computational learning theory*, pages 164–170, 1999.
51. Dongyu Meng and Hao Chen. Magnet: a two-pronged defense against adversarial examples. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 135–147. ACM, 2017.
52. Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischoff. On detecting adversarial perturbations. *arXiv preprint arXiv:1702.04267*, 2017.
53. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
54. Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 86–94, 2017.
55. Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2574–2582, 2016.

56. Taesik Na, Jong Hwan Ko, and Saibal Mukhopadhyay. Cascade adversarial machine learning regularized with a unified embedding. In *International Conference on Learning Representations (ICLR)*, 2018.
57. Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 807–814, 2010.
58. Augustus Odena and Ian Goodfellow. TensorFuzz: Debugging neural networks with coverage-guided fuzzing. *arXiv preprint arXiv:1807.10875*, 2018.
59. Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff nets: Stealing functionality of black-box models. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 4954–4963. Computer Vision Foundation / IEEE, 2019.
60. Nicolas Papernot, Fartash Faghri, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Alexey Kurakin, Cihang Xie, Yash Sharma, Tom Brown, Aurko Roy, Alexander Matyasko, Vahid Behzadan, Karen Hambardzumyan, Zhishuai Zhang, Yi-Lin Juang, Zhi Li, Ryan Sheatsley, Abhibhav Garg, Jonathan Uesato, Willi Gierke, Yinpeng Dong, David Berthelot, Paul Hendricks, Jonas Rauber, and Rujun Long. Technical report on the cleverhans v2.1.0 adversarial examples library. *arXiv preprint arXiv:1610.00768*, 2018.
61. Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 582–597. IEEE, 2016.
62. Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 372–387. IEEE, 2016.
63. Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18. ACM, 2017.
64. Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Towards practical verification of machine learning: The case of computer vision systems. *arXiv preprint arXiv:1712.01785*, 2017.
65. 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.
66. Roberto Pietrantuono, Peter Popov, and Stefano Russo. Reliability assessment of service-based software under operational profile uncertainty. *Reliability Engineering & System Safety*, 204:107193, 2020.
67. Omid Poursaeed, Isay Katsman, Bicheng Gao, and Serge J. Belongie. Generative adversarial perturbations. In *Conference on Computer Vision and Pattern Recognition*, 2018.
68. Robotis. Robotis(2019) turtlebot3 – e-manual, waffle pi. [Online] https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/. (Accessed on 02 August 2021).
69. Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.
70. RTCA. Do-178c, software considerations in airborne systems and equipment certification. 2011.
71. Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. Reachability analysis of deep neural networks with provable guarantees. In *IJCAI*, pages 2651–2659, 2018.
72. Aniruddha Saha, Akshayvarun Subramanya, and Hamed Pirsiavash. Hidden trigger backdoor attacks. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 11957–11965. AAAI Press, 2020.
73. Ahmed Salem, Yang Zhang, Mathias Humbert, Pascal Berrang, Mario Fritz, and Michael Backes. Ml-leaks: Model and data independent membership inference attacks and defenses on

machine learning models. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.
74. Ali Shafahi, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 6106–6116, Red Hook, NY, USA, 2018. Curran Associates Inc.
75. Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 3–18. IEEE Computer Society, 2017.
76. 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.
77. Lorenzo Strigini and Andrey Povyakalo. Software fault-freeness and reliability predictions. In *SafeComp2013*, pages 106–117, 2013.
78. Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. A survey on data-flow testing. *ACM Computing Surveys*, 50(1):5:1–5:35, March 2017.
79. Youcheng Sun, Xiaowei Huang, and Daniel Kroening. Testing deep neural networks. *CoRR*, abs/1803.04792, 2018.
80. Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Shap, Matthew Hill, and Rob Ashmore. Structural test coverage criteria for deep neural networks. 2018.
81. Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Concolic testing for deep neural networks. In *Automated Software Engineering (ASE), 33rd IEEE/ACM International Conference on*, 2018.
82. Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Deepconcolic: Testing and debugging deep neural networks. In *41st ACM/IEEE International Conference on Software Engineering (ICSE2019)*, 2018.
83. Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
84. Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.
85. 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.
86. Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017.
87. Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble Adversarial Training: Attacks and Defenses. In *International Conference on Learning Representations*, 2018.
88. Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 601–618, USA, 2016. USENIX Association.
89. Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 36–52. IEEE Computer Society, 2018.
90. T.-W. Weng, H. Zhang, P.-Y. Chen, J. Yi, D. Su, Y. Gao, C.-J. Hsieh, and L. Daniel. Evaluating the Robustness of Neural Networks: An Extreme Value Theory Approach. In *ICLR2018*, 2018.
91. Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. Feature-guided black-box safety testing of deep neural networks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 408–426. Springer, 2018.
92. Dongxian Wu, Yisen Wang, Shu-Tao Xia, James Bailey, and Xingjun Ma. Skip connections matter: On the transferability of adversarial examples generated with resnets. *arXiv preprint arXiv:2002.05990*, 2020.

93. 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.
94. Chaowei Xiao, Jun-Yan Zhu, Bo Li, Warren He, Mingyan Liu, and Dawn Song. Spatially transformed adversarial examples. *arXiv preprint arXiv:1801.02612*, 2018.
95. Weilin Xu, David Evans, and Yanjun Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. *arXiv preprint arXiv:1704.01155*, 2017.
96. Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. Privacy risk in machine learning: Analyzing the connection to overfitting. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 268–282, 2018.
97. Yang Yu. Towards sample efficient reinforcement learning. In *IJCAI*, pages 5739–5743, 2018.
98. Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013.
99. Xingyu Zhao, Alec Banks, James Sharp, Valentin Robu, David Flynn, Michael Fisher, and Xiaowei Huang. A safety framework for critical systems utilising deep neural networks. In *SafeComp2020*, pages 244–259, 2020.
100. Xingyu Zhao, Wei Huang, Alec Banks, Victoria Cox, David Flynn, Sven Schewe, and Xiaowei Huang. Assessing reliability of deep learning through robustness evaluation and operational testing. In *SafeComp2021*, 2021.
101. Xingyu Zhao, Wei Huang, Vibhav Bharti, Yi Dong, Victoria Cox, Alec Banks, Sen Wang, Sven Schewe, and Xiaowei Huang. Reliability assessment and safety arguments for machine learning components in assuring learning-enabled autonomous systems. *CoRR*, abs/2112.00646, 2021.
102. Chen Zhu, W. Ronny Huang, Hengduo Li, Gavin Taylor, Christoph Studer, and Tom Goldstein. Transferable clean-label poisoning attacks on deep neural nets. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7614–7623. PMLR, 09–15 Jun 2019.
103. Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.