ABSTRACT

YU, YAHE. Machine Learning Approach to Biological Applications: STI Strategies for HIV and Evolutionary Genetics. (Under the direction of Hien Tran.)

With the rapid development of artificial intelligence in last decades, many biological problems can be solved by using artificial intelligence methods. In this thesis, we mainly focus on two applications.

In the first chapter, we use reinforcement methods to find optimal strategies for HIV patient. HIV stands for human immunodeficiency virus. HIV infection was an acute fatal disease in the 1980s, but it is a manageable chronic disease now. The HIV infected patients, regardless of how long they've had the virus or how they got infected, can have an undetectable viral load and enjoy a long and healthy life by taking the antiretroviral treatment (ART). Structured treatment interruption (STI) is a new therapy for HIV infected patients. STI may be used to reduce side effects of medications, to enhance a medication's effectiveness when restarted, or as a step towards stopping treatment altogether. But how to make a proper STI strategy for HIV patients is still a difficult problem. In this thesis, we introduce the XGBoost regression method into fitted Q iteration algorithm to design STI treatment for HIV patients. The XGBoost based fitted Q iteration converges faster than conventional extra-trees based fitted Q iteration on HIV problems. By using an XGBoost based fitted Q iteration algorithm, we can obtain the optimal STI strategy with fewer training data compared with extra-trees based fitted Q iteration. Finally, comparing with extra-trees based fitted Q, XGBoost based fitted Q is much more computationally efficient.

In the second chapter, we use deep learning methods to find the exponential growth virus infection samples. Being able to detect exponential growth samples is of paramount importance. We can use this to predict an epidemic, or check a virus adaptation to the drug pressure, i.e, drug resistance, or check the immune escape within a person. However, it is very difficult to find the exponential growth samples in a very big sequence of data. The demographic history leads to differences in the shapes of phylogenetic trees, as well as the pairwise difference matrices. We transfer all pairwise difference matrices from the simulated sequence data into heat maps. By using two UNet++ networks and a softmax classifier, we can successfully find all the exponential blocks, which ranged from size 5 to 125, contained in the heat maps. Finally, a probability that a sequencing sample contains an exponential block will also be provided.

© Copyright 2020 by Yahe Yu

All Rights Reserved

Machine Learning Approach to Biological Applications:
STI Strategies for HIV and Evolutionary Genetics

by
Yahe Yu

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Applied Mathematics

Raleigh, North Carolina

2020

APPROVED BY:

Kevin Flores	Tien Khai Nguyen
Ryan Murray	Hien Tran Chair of Advisory Committee

DEDICATION

This thesis is dedicated to my family: my parents, Guoqiang Yu and Xiaoli Ping, who always love and support me, my wife, Yaru Wang, who always encourages and believes in me, and my brother, Pengyi Yu, who always makes me happy. Also, I would like to dedicate this thesis in memory of a great mathematician, Dr. H.T. Banks.

BIOGRAPHY

Yahe Yu was born in 1991, Ruzhou, Henan province, China and grew up there. He received his B.S. degree in Information and Computational Science from the department of Mathematics in 2014, and a B.S. degree in Information Management and System from the department of Economics and Management in 2015 at Dalian University of Technology. In 2015, he started his graduate study in the department of Mathematics at North Carolina State University. In 2016, he transferred from the master program to the Ph.D. program. In 2017, he received his M.S. degree in Applied Mathematics. From 2017, he started doing research on reinforcement learning and deep learning under the guidance of Professor Hien Tran.

ACKNOWLEDGEMENTS

First of all, I would like to give my most sincere gratitude to my advisor, Dr. Hien Tran. He is the best advisor I have ever met. He is always encouraging and motivating me. It was he who always gives me hope and suggestions when I encounter difficult problems in my research. His guidance helped me a lot when I was doing my research and writing this thesis.

I would like to thank the rest of my thesis committee: Dr. H.T. Banks, Dr. Kevin Flores, Dr. Tien Khai Nguyen and Dr. Ryan Murray, for their kind support, encouragement and advice on my research. It is my honor to have them in my committee.

Also, I would also like to thank my friends, Jiabin Yu and Yan Jun, who help and encourage me when I was taking courses and doing my research; Jiapeng Zhu, who always takes care of me and helps cutting the hair for me; Ran Bi, who is my roommate and helps me a lot in my daily life.

Last but not least, I would like to thank everyone that I met here. It was them who make up the whole picture of my life.

TABLE OF CONTENTS

LIST OF TABLES . . . . . **vii**

LIST OF FIGURES . . . . . **viii**

Chapter 1 USING REINFORCEMENT LEARNING METHODS TO FIND THE OPTIMAL STI STRATEGIES FOR HIV PATIENTS **1**

1.1 Introduction 1
1.2 Tree-Based Methods 3
1.2.1 Decision tree 3
1.2.2 Random forest 8
1.2.3 Extremely randomized trees 8
1.2.4 XGBoost 10
1.3 Reinforcement Learning 13
1.3.1 Markov decision processes 13
1.3.2 Dynamic programming 15
1.3.3 Exploitation and exploration 17
1.3.4 Model-based and model-free reinforcement learning 18
1.3.5 XGBoost based fitted Q iteration algorithm 19
1.4 HIV Infection and STI Treatment 22
1.4.1 Introduction of HIV 22
1.4.2 Nonlinear ODE model of HIV infection dynamics 23
1.4.3 Optimal STI treatment formulation 25
1.5 Data Generation and Experiments 27
1.6 Numerical Results 30
1.6.1 Baseline results 30
1.6.2 Fitted Q iteration algorithm results 31
1.7 Contributions 40
1.8 Future Work 41

Chapter 2 IMAGE SEGMENTATION FOR EVOLUTIONARY GENETICS **43**

2.1 Introduction 43
2.2 Neural Network 44
2.2.1 Single-layer perceptron 44
2.2.2 Multi-layer perceptron 45
2.2.3 Activation functions 46
2.2.4 Backpropagation algorithms 48
2.3 Convolutional Neural Networks 52
2.3.1 Convolutional layer 52
2.3.2 Pooling layer 55
2.3.3 Fully connected layer 55
2.3.4 Dropout 56
2.3.5 Loss function 57

2.3.6 Evaluation metrics . . . 58
2.4 U-Net and UNet++ . . . 58
2.4.1 U-Net . . . 58
2.4.2 UNet++ . . . 59
2.5 Evolutionary Genetics . . . 61
2.6 Experiments and Results . . . 66
2.6.1 UNet++ for large exponential block samples . . . 66
2.6.2 UNet++ for small exponential block samples . . . 71
2.6.3 Ensemble of UNet++ . . . 72
2.6.4 Classifier . . . 75
2.7 Conclusion . . . 77
2.8 Contributions . . . 78
2.9 Future Work . . . 80

BIBLIOGRAPHY . . . **81**

LIST OF TABLES

Table 1.1 Dataset for Play Tennis. 5
Table 1.2 The parameters used in the model (1.41). The superscripts * denote estimated from human data and ** denote those estimated from macaque data. 24

Table 2.1 Segmentation results for UNet++ for large exponential block samples, UNet++ for small exponential block samples and Ensemble of UNet++. 78

LIST OF FIGURES

Figure 1.1 Partitioning of the root node based on the "Outlook" attribute. . . . 6
Figure 1.2 Decision tree based on the Dataset for Play Tennis. 7
Figure 1.3 Diagram of a random decision forest. 9
Figure 1.4 The six-tuples state of the simulated patients started with "non-healthy" state in 1000 days by following with the no treatment(– –) strategy ($\epsilon_1 = 0, \epsilon_2 = 0$), and with the fully treatment (–) strategy ($\epsilon_1 = 0.7, \epsilon_2 = 0.3$). 32
Figure 1.5 The first STI strategy computed by the extra-trees based fitted Q iteration 33
Figure 1.6 The first STI strategy computed by the XGBoost based fitted Q iteration 33
Figure 1.7 The six-tuples state of the simulated patients started with "non-healthy" state in 1000 days by following with the first STI strategy computed by the XGBoost based fitted Q iteration, and with the first STI strategy computed by the extra-trees based fitted Q iteration. . 34
Figure 1.8 The seventh STI strategy computed by the extra-trees based fitted Q iteration 35
Figure 1.9 The seventh STI strategy computed by the XGBoost based fitted Q iteration 36
Figure 1.10 The six-tuples state of the simulated patients started with "non-healthy" state in 1000 days by following with the seventh STI strategy computed by the XGBoost based fitted Q iteration, and with the seventh STI strategy computed by the extra-trees based fitted Q iteration. 37
Figure 1.11 The optimal STI strategy computed by the extra-trees based fitted Q iteration 38
Figure 1.12 The optimal STI strategy computed by the XGBoost based fitted Q iteration 38
Figure 1.13 The six-tuples state of the simulated patients started with "non-healthy" state in 1000 days by following with the optimal STI strategy computed by the XGBoost based fitted Q iteration, and with the optimal STI strategy computed by the extra-trees based fitted Q iteration. 39
Figure 1.14 The influence of the number of simulated patients contained in the fitted Q iteration algorithms on the total discounted reward (in the first 1000 days) with respect to the computed strategies. 41

Figure 2.1 Single-layer perceptron with bias [Agg18]. 44
Figure 2.2 Multi-layer perceptron with bias [Agg18]. 45
Figure 2.3 Step function. 46
Figure 2.4 Sigmoid function. 47
Figure 2.5 Tanh function. 47
Figure 2.6 ReLU function. 48

Figure 2.7 The example of a simple neural network. 49
Figure 2.8 An example of convolution operation between a $7 \times 7 \times 1$ input and a $3 \times 3 \times 1$ filter with stride of 1 [Agg18]. 53
Figure 2.9 An example of applying convolution operation on a real image [Fra17]. 54
Figure 2.10 An example of padding [Agg18]. 55
Figure 2.11 An example of maximum pooling with a 3×3 filter, and stride equals to 1 and 2 [Agg18]. 56
Figure 2.12 Dropout Neural Net Model:(a) is the standard neural net; (b) is the neural net after applying dropout [Sri14]. 57
Figure 2.13 The structure of U-Net [Ron15]. 59
Figure 2.14 The structure of UNet++ [Zho18]. 60
Figure 2.15 Idealized phylogenetic trees of the (A) exponential growth virus population and (B) constant growth virus population and the corresponding population size [Vol13]. 62
Figure 2.16 The phylogenetic trees with mutations of (A) exponential growth virus population and (B) constant growth virus population, respectively. 62
Figure 2.17 The heat maps for the normalized matrices A and B, respectively. . 64
Figure 2.18 The heat maps contain exponential block. 64
Figure 2.19 Some examples of heat maps in the simulated data set: graph (a) and (b) is the heat map and mask map which contains a exponential black with size 5; (c) and (d) is the heat map and mask map which contains a exponential black with size 50; (e) and (f) is the heat map and mask map which contains a exponential black with size 100. . . 67
Figure 2.20 The IoU accuracy of the UNet++ of different training sets with common difference 1, 2, 3, 5, 10, 15, 20, 25, respectively. 69
Figure 2.21 The IoU accuracy of the UNet++ for large exponential block samples. 70
Figure 2.22 The worst cases for identifying the size 5 exponential blocks in the test samples (Red rectangle are the true exponential block area, the yellow areas are the predicted results). 71
Figure 2.23 The performance of the UNet++ for small exponential block samples. 72
Figure 2.24 The performance of the ensemble of UNet++. 73
Figure 2.25 The predicted results for both UNet++ networks: (a) Sample with size 72 exponential block; (b) Sample with size 29 exponential block; (c) Sample with size 10 exponential block; (d) Sample with size 5 exponential block (Red rectangle are the true exponential block area, the yellow areas are the predicted results). 74
Figure 2.26 The samples with worst prediction of ensemble UNet++: (a) Sample with size 5 exponential block; (b) Sample with size 5 exponential block(a) Sample with size 33 exponential block. 76
Figure 2.27 The structure of the classifier. 77
Figure 2.28 Confusion matrix for the classifier. 78
Figure 2.29 The final results of ensemble UNet++. 79

CHAPTER

1

USING REINFORCEMENT LEARNING METHODS TO FIND THE OPTIMAL STI STRATEGIES FOR HIV PATIENTS

1.1 Introduction

The traditional way for designing the medical treatment strategies is based on the standardized application of clinical studies and experience, and mainly relied on trial-and-error to guide decisions for individual patients. The clinicians need to make a series of sequential treatment strategy for many diseases, especially the chronic and acute diseases. The strategy is to decide when to give the treatment, change the treatment and stop the treatment. Usually it is difficult for clinicians to make those strategies. The clinicians mainly relied on their general clinical practice and intuition, the patient's overall medical history, as well as the observed responses to different treatments.

HIV stands for human immunodeficiency virus. The virus can weakens a person's immune system by attacking cells in the immune system that fight diseases and infections. The cells being killed in the immune system are called T-helper cells, also referred to as CD4+ T-lymphocytes. After the virus killed the CD4+ cells, it will make more copies of itself.

The virus will gradually weaken the infected person's immune system. If the HIV virus is left untreated, it will cause the acquired immunodeficiency syndrome (AIDS). AIDS is the end-stage disease of HIV [Wei93]. HIV is found in semen, blood, vaginal and anal fluids, and breastmilk. It is mainly passed on through unprotected sex (without a condom), sharing needles or syringes and during pregnancy, birth, or breastfeeding.

Despite significant recent advances in medical technology, there is still no effective way to cure the HIV. But the HIV infected patient can have an undetectable viral load and enjoys a long and healthy life by taking the antiretroviral treatment (ART). However, ART is not a cure for HIV. It only keeps HIV under control, so it will not affect the infected people's normal life. ART is a combination of drugs that is commonly used to treats HIV. Usually, ART contains a combination of 3 or more drugs to have the greatest chance of lowering the amount of HIV in infected patient. The drugs used to treat HIV can be divided into two categories: protease inhibitors (PIs) and reverse transcriptase inhibitors (RTIs). RTIs can prevent HIV RNA converted into DNA. So RTIs can block the integration of the viral code into the target cell [Ada04]. Meanwhile, PIs can prevent the normal structuring and cutting of the viral proteins before they are released from the host cell. PIs works in the final stage of the viral life. Thus PIs can dramatically reduce the number of infectious virus particles that released by the infected cell.

Nowadays, ART is recommended for all people with HIV, regardless of how long they've had the virus or how healthy they are. But the flaws of this treatment are obvious. Patients experience many common side effects when they take the ART. Also, some of them have the poor adherence to the drugs. In the long term use, it can increase the risk of myocardial infarction [AHDDSG03]. Another disadvantage of ART is the cost of ART is very expensive. The per-person lifetime cost is very high, which is around 77,300 dollars with the three-drugs therapy [Fre01]. So how to properly and efficiently administer the treatment is still an open problem that is worth to explore. Structured treatment interruption (STI) is an alternative treatment strategy designed to overcome these problems [Lor00]. STI is one of the most controversial and interesting topics in HIV medicine. The key part of STI is cycled on and off (turn on and off) the drug therapy on a fixed schedule (long or short) in order to reduce a patient's exposure to antiretroviral agents [Rui00].

The Berlin patient, Timothy Ray Brown, considered to be the first person cured of HIV/AIDS. This patient was treated soon after he was diagnosed with HIV infection. However, the treatment was interrupted on two occasions due to the intercurrent conditions. After the third introduction of therapy, treatment was terminated and the Berlin patient had continual suppression of viral replication in the absence of any ART [Jes14]. After this, there has been a growing interest in finding the optimal STI strategy for HIV infection

circa 2000. In 2000, Lori et al. designed the structured treatment interruptions to control HIV-1 infection [Lor00]. In 2004, Adams et al. developed a nonlinear ODE model for HIV infection by investigating the single drug control [Ada05]. In 2005, Adams et al. formulated a dynamic model with compartments including target cells, infected cells, virus, and immune response that is subject to multiple drug treatments as control inputs [Ada04]. They found the suboptimal STI strategy for HIV infected patients by using the ideas from dynamic programming. In 2006, Ernst et al. applied a reinforcement learning method, which is called fitted Q iteration algorithm, to find the optimal STI strategies for HIV [Ern06]. In this chapter, we will apply a new method, called XGBoost based fitted Q iteration, to find the optimal STI strategy for HIV patients.

1.2 Tree-Based Methods

Tree-based methods are one of the mostly used machine learning methods. They can be used to solve both classification and regression problems. They are conceptually simple but powerful. They have the high accuracy, stability and ease of interpretation. There are many tree-based methods, including decision tree, random forest, extra-trees and gradient boosting trees, etc. We will introduce them in this section.

1.2.1 Decision tree

Decision tree is a very commonly used machine learning method and the basic of the tree based methods. It can be used to both classification and regression problems. The goal of the decision tree is to create a model that predicts the value of a target variable based on several input variables.

In general, a decision tree contains a root node, several internal nodes and several leaf nodes. Leaf nodes correspond to decision results, while each of the other nodes corresponds to an attribute test. The set of samples contained in each node is partitioned into child nodes based on the results of the attribute test. The root node contains the entire set of samples. The path from the root node to each leaf node corresponds to a sequence of decision tests. The purpose of decision tree learning is to produce a decision tree with strong generalization capability, i.e., the ability to handle unseen samples.

The generation of a decision tree is a recursive process. In the basic decision tree algorithm, there are three situations that cause recursion to stop: (1) the current node contains samples that all belong to the same category and do not need to be partitioned; (2) the current set of attributes is empty or all samples take the same value on all attributes and

cannot be partitioned; (3) the set of samples contained in the current node is empty and cannot be partitioned.

The key to decision tree learning is how to choose the optimal partitioning attribute. As the partitioning process continues, we want the decision tree's branching nodes to contain samples that belong to the same category as much as possible, i.e., the purity of each node as high as possible.

Information entropy is one of the most commonly used metrics to measure the purity of a sample set. Assuming that the proportion of samples of the kth class in the current sample set D is p_k $(k = 1, 2, \ldots, n)$, the information entropy of D is defined as:

$$\mathrm{Ent}(D) = -\sum_{k=1}^{n} p_k \log_2 p_k, \tag{1.1}$$

where n is the total number of classes containing in the entire data set.

Assuming that the discrete attribute **a** has M possible values $\{a_1, a_2, \ldots, a_M\}$, if we use attribute **a** to divide the current sample set D, we will have M branch nodes, where the m-th branch node contains all samples in D that have a value of a_m on attribute **a**. We denoted the set of samples contained in the m-th branch node as D_m. We can use equation (1.1) to compute the information entropy of D_m. Given that different branch nodes contain different numbers of samples, we assign weights to each branch node $\frac{|D_m|}{|D|}$ (where $|\cdot|$ is the cardinality of a set). The more samples a branch node have, the more weight it will have. The gain of information obtained by dividing the sample set D by the attribute **a** can be calculated by:

$$\mathrm{Gain}(D, \mathbf{a}) = \mathrm{Ent}(D) - \sum_{m=1}^{M} \frac{|D_m|}{|D|} \mathrm{Ent}(D_m). \tag{1.2}$$

In general, the greater the information gain of an attribute, the greater the purity gain obtained by using that attribute for partitioning. Thus, we can use the information gain to select the decision tree's partitioning attributes. The famous ID3 algorithm [Qui86] used information gain to select the partitioning attribute.

We will give an example to show how to create a decision tree. The data set is called "Dataset for Play Tennis" [Wit02]. Table 1.1 shows the entire data set. The data set contains 14 training samples. Each sample contains 4 features: Outlook, Temperature, Humidity and Windy. There are two classes in the data set: play tennis and do not play tennis.

When the decision start, the root node contains the entire set of samples. The proportion of play tennis is $p_1 = \frac{9}{14}$. The proportion of do not play tennis is $p_2 = \frac{5}{14}$. So the information

Table 1.1 Dataset for Play Tennis.

Day	Outlook	Temperature	Humidity	Windy	Play
1	sunny	hot	high	false	no
2	sunny	hot	high	true	no
3	overcast	hot	high	false	yes
4	rainy	mild	high	false	yes
5	rainy	cool	normal	false	yes
6	rainy	cool	normal	true	no
7	overcast	cool	normal	true	yes
8	sunny	mild	high	false	no
9	sunny	cool	normal	false	yes
10	rainy	mild	normal	false	yes
11	sunny	mild	normal	true	yes
12	overcast	mild	high	true	yes
13	overcast	hot	normal	false	yes
14	rainy	mild	high	true	no

entropy of the root node is:

$$\text{Ent}(D) = -\sum_{k=1}^{2} p_k \log_2 p_k = -\left(\frac{9}{14}\log_2\frac{9}{14} + \frac{5}{14}\log_2\frac{5}{14}\right) = 0.940. \tag{1.3}$$

We have 4 attributes in our current data set. So we need to calculate the information gain for each attribute. We will use the attribute "Outlook" as an example. In the "Outlook" attribute, there are 3 values: {sunny, rainy, overcast}. If we use this feature to divide D, we will get 3 subsets: D_1(Outlook = sunny), D_2(Outlook = rainy) and D_3(Outlook = overcast).

The subset D_1 contains the samples with day $\{1,2,8,9,11\}$, of which the proportion of playing tennis is $p_1 = \frac{2}{5}$. The proportion of do not play tennis is $p_2 = \frac{3}{5}$. So the information entropy of D_1 is

$$\text{Ent}(D_1) = -\sum_{k=1}^{2} p_k \log_2 p_k = -\left(\frac{2}{5}\log_2\frac{2}{5} + \frac{3}{5}\log_2\frac{3}{5}\right) = 0.97. \tag{1.4}$$

The subset D_2 contains the samples with day $\{4,5,6,10,14\}$, of which the proportion of playing tennis is $p_1 = \frac{3}{5}$. The proportion of do not play tennis is $p_2 = \frac{2}{5}$. So the information entropy of D_2 is

$$\text{Ent}(D_2) = -\sum_{k=1}^{2} p_k \log_2 p_k = -\left(\frac{3}{5}\log_2\frac{3}{5} + \frac{2}{5}\log_2\frac{2}{5}\right) = 0.97. \tag{1.5}$$

The subset D_3 contains the samples with day $\{3,7,12,13\}$, of which the proportion of playing tennis is $p_1 = \frac{4}{4}$. The proportion of do not play tennis is $p_2 = \frac{0}{4}$. So the information entropy of D_3 is

$$\text{Ent}(D_3) = -\sum_{k=1}^{2} p_k \log_2 p_k = -\left(\frac{4}{4}\log_2\frac{4}{4} + \frac{0}{4}\log_2\frac{0}{4}\right) = 0. \tag{1.6}$$

The gain of information obtained by dividing the sample set D by the attribute "Outlook" can be calculated by:

$$\begin{aligned}\text{Gain}(D,\text{Outlook}) &= \text{Ent}(D) - \sum_{m=1}^{3}\frac{|D_m|}{|D|}\text{Ent}(D_m) \\ &= 0.940 - (\frac{5}{14}\times 0.970 + \frac{5}{14}\times 0.970 + \frac{4}{14}\times 0) \\ &= 0.69.\end{aligned} \tag{1.7}$$

Similarly, we can calculate the information gain of other attributes:

$$\begin{aligned}&\text{Gain}(D,\text{Temperature}) = 0.029,\\ &\text{Gain}(D,\text{Humidity}) = 0.152,\\ &\text{Gain}(D,\text{Windy}) = 0.048.\end{aligned}$$

The attribute "Outlook" has the largest information gain. Hence, we choose "Outlook" as the partitioning attribute for root node. Figure 1.1 shows the result of a partition of the root node based on the "Outlook" attribute.

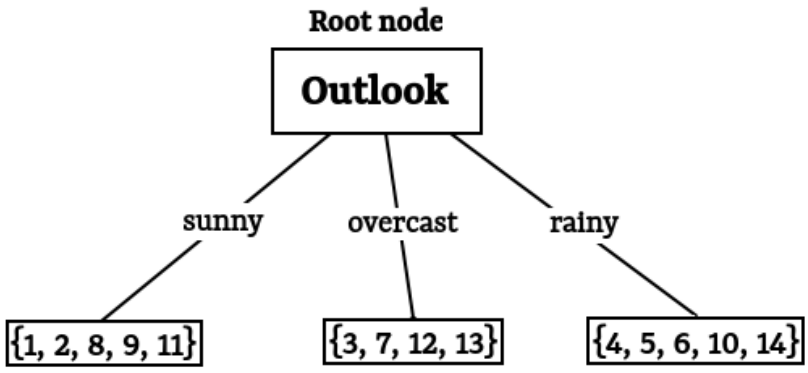


Figure 1.1 Partitioning of the root node based on the "Outlook" attribute.

Then, the decision tree learning algorithm will further partition each branch node. By

applying the same strategy to each branch node, we can partition the nodes until the three situations we descried above is reached. The whole decision tree is shown in Figure 1.2.

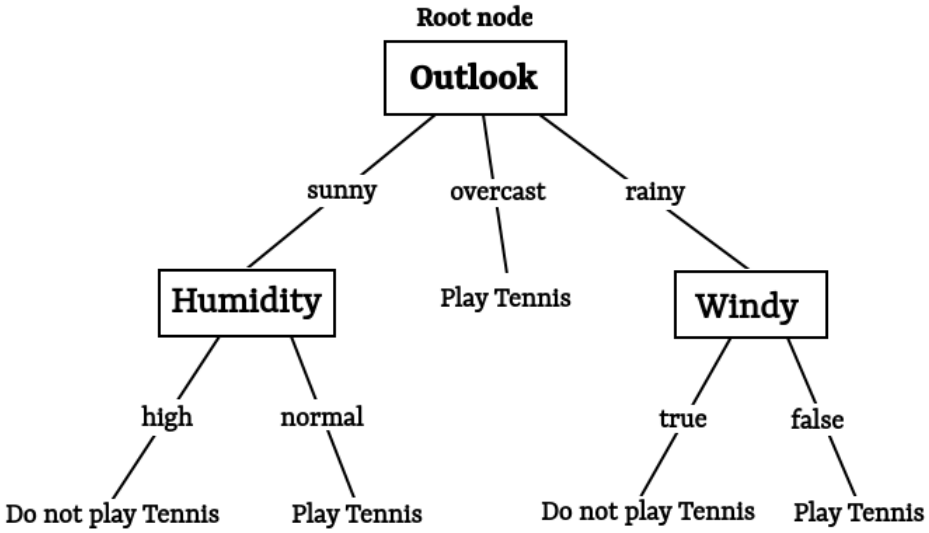


Figure 1.2 Decision tree based on the Dataset for Play Tennis.

There are also some other decision tree algorithms that use different approaches to select the partitioning attribute. The C4.5 decision tree algorithm use the gain ratio to select the partitioning attribute. The gain ratio [Qui93] is defined as:

$$\text{Gain ratio}\,(D,\mathbf{a}) = \frac{\text{Gain}(D,\mathbf{a})}{\text{IV}(\mathbf{a})}, \tag{1.8}$$

where $\text{IV}(\mathbf{a}) = -\sum_{m=1}^{M} \frac{|D_m|}{|D|} \log_2 \frac{|D_m|}{|D|}$ is the intrinsic value of the attribute $\mathbf{a}$.

Another famous decision tree algorithm is classification and regression tree (CART) algorithm [Bre84]. In this algorithm, the Gini index was used to select the partitioning attribute. The purity of dataset D can be measured by Gini value:

$$\begin{aligned} \text{Gini}(D) &= \sum_{k=1}^{n} \sum_{k' \neq k} p_k p_{k'} \\ &= 1 - \sum_{k=1}^{n} p_k^2. \end{aligned} \tag{1.9}$$

Intuitively, $\text{Gini}(D)$ reflects the probability that two randomly selected samples from the dataset D will have inconsistent category labels. Therefore, the smaller the $\text{Gini}(D)$ value is,

the higher the purity of the data set D will have. The Gini index of attribute **a** is defined as:

$$\text{Gini index}(D,a)=\sum_{m=1}^{M}\frac{|D_m|}{|D|}\text{Gini}(D_m). \tag{1.10}$$

Here, we will select the attribute that minimizes the Gini index after partitioning as the optimal partitioning attribute.

1.2.2 Random forest

Random forest algorithm is an extension of the decision tree algorithm. Random forest is an ensemble learning method for classification, regression and other tasks that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or the average prediction (regression) of the individual trees [Ho95; Ho98].

Random forest contains many decision trees. Instead of using the entire dataset to train each decision tree, we use the random bootstrap sample set to train each tree. Assuming we have n decision trees in the forest. In order to build the decision trees in the forest, we need to draw n random bootstrap sample sets from the entire set of samples. Assuming the size of all the bootstrap data sets is m. For each bootstrap data set, we randomly choose m samples from the original data set with replacement. Notice that we are allowed to select the same sample more than once in the bootstrap data set. Then we grow each decision tree by using the random bootstrap data set. The way to build each decision tree is same as the CART tree algorithm. After all the decision trees are build, we aggregate the prediction by each tree to assign the class label by majority vote or average. Figure 1.3 shows an example of random forest.

1.2.3 Extremely randomized trees

Extremely randomized trees [Geu06] was proposed by Pierre Geurts, Damien Ernst and Louis Wehenkel in 2006. It is a tree-based ensemble method for supervised classification and regression problems. The default setting of extremely randomized trees is called extra-trees.

Extra-trees is an ensemble method. It builds a forest with many trees, just like most random forest algorithms [Bre01]. Extra-trees method combines the attribute randomization of a random subspace with a totally random selection of cut-point. The big differences between extra-trees and other tree-based ensemble methods are that they split nodes by choosing cut-points fully at random and use the entire learning samples (rather than a bootstrap replica) to grow the trees.

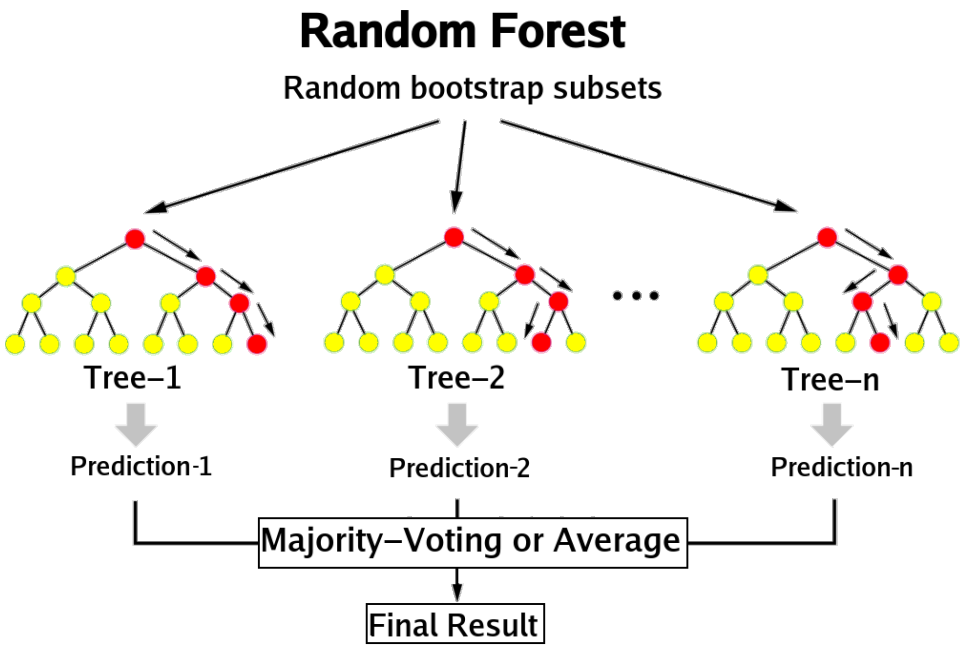


Figure 1.3 Diagram of a random decision forest.

The extra-trees method has three important parameters: K is the number of attributes randomly selected at each node, it controls the strength of the attribute randomization; $n_{\min}$ is the minimum number of elements required to split a node, it controls the degree of smoothing; and M is the number of trees to create in the forest.

We will describe the whole process of extra-tree method when applying on the data with numerical features. Given a training set which contains a set of two-tuples, one of the tuple is a vector of numerical input features and another is a scalar output. The method will generate M binary unpruned decision trees independently in the forest. Those trees can be generated parallelly since they are independent with each other. For each tree in the forest, it will use the whole training set to grow. Each tree in the forest grows by following the same rules. Assuming we select an arbitrary tree from the forest. The entire training set will be the input of the root node of the tree. From all the non constant input features, we select K features at random without replacement. We compute the maximal and minimal value of each feature in the K selected features. For each feature, a discretization threshold will be selected uniformly between the maximal and minimal values. Then a score will be calculated based on the information gain for each features. We will have K scores in total. Selected the feature with the largest score as the feature to split the data set into left and right child nodes. This procedure will be repeated for each of the left and right child node until one of the stop split conditions is satisfied. There are three stop split conditions: the

number of samples contains in this node is less than $n_{\min}$, all attributes are constant in this node and the output is constant in this node. Once all leaf nodes satisfied the stop split condition, the tree is built. For the classification tree, the output at a given leaf node is the class with the maximal number contained in the leaf node. For the regression tree, the output at a given leaf node is the mean of the output values from all the samples contained in the leaf node.

After all trees in the forest are built, we can evaluate a given tuples with only a vector of features. The tuples will go through each tree in the forest. It will have M predicted values. For the classification problem, the final prediction will be the class with the maximal number in those M predicted values. For the regression problem, the final prediction will be the arithmetic mean of the M predicted values.

1.2.4 XGBoost

XGBoost is a tree-based method that was proposed by Tianqi Chen in 2016. It is based on the framework of gradient boosting. It is widely used by data scientists to achieve state-of-the-art results on many machine learning challenges and runs more than ten times faster than existing popular solutions on a single machine [Che16]. XGBoost stands for "Extreme Gradient Boosting". It is also a tree-based ensemble method designed for supervised learning problems.

Just like most of the tree-based ensemble methods, XGBoost needs to build an ensemble of decision trees. A set of classification and regression trees (CART) are contained in the tree ensemble model. But unlike the extra-trees, instead of using the arithmetic mean of all predicted values as output, XGBoost uses the sum of all predictions from trees in the forest. Given a training set which contains a set of two-tuples, one of the tuple is input features x_i and another is a scalar output y_i. Let t be a function of CART in the function space $\mathscr{T}$ which is the set of all possible CARTs and be defined as:

$$t(x) = w_{q(x)}, w \in \mathbb{R}^T, q : \mathbb{R}^d \to \{1, 2, \cdots, T\}, \tag{1.11}$$

where T is the number of leaves, w is the vector of scores on leaves, and q is the structure of the tree, i.e., the function assigning each data point to the corresponding leaf.

Unlike the way that extra-trees method builds each tree, the XGBoost builds each tree

by minimizing the objective function:

$$\begin{aligned} \text{obj} &= L(\theta) + \Omega(\theta) \\ &= \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(t_k), \end{aligned} \tag{1.12}$$

where n is the number of samples, K is the number of trees in the tree ensemble model, L is the training loss function, Ω is the regularization term, and l is a differentiable convex loss function that measures the difference between the prediction $\hat{y}_i$ and data y_i. The training loss function is used to measure the error of the tree ensemble model and is usually defined by the mean square error (MSE), which is given by:

$$L(\theta) = \sum_{i} (y_i - \hat{y}_i)^2. \tag{1.13}$$

The regularization term Ω is used to control the complexity of the tree ensemble model. It can help us to avoid the overfitting problem. It is defined as:

$$\Omega(t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2. \tag{1.14}$$

The additive strategy is applied in XGBoost method to find the parameters of trees. The key point of additive strategy is to fix what we learned and add one new tree at a time. Assuming that, at time m, we have m trees in the forest and the prediction values of them are $\hat{y}_i^m$. Then we can rewrite $\hat{y}_i^m$ as:

$$\hat{y}_i^m = \sum_{k=1}^{m} t_k(x_i) = \sum_{k=1}^{m-1} t_k(x_i) + t_m(x_i) = \hat{y}_i^{m-1} + t_m(x_i). \tag{1.15}$$

Then, the objective function at step m is

$$\begin{aligned} \text{obj}^m &= \sum_{i=1}^{n} l(y_i, \hat{y}_i^m) + \sum_{i=1}^{m} \Omega(t_i) \\ &= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{m-1} + t_m(x_i)) + \Omega(t_m) + \text{constant}. \end{aligned} \tag{1.16}$$

The loss function can be written as a function with a first order term and a quadratic term. But for other loss functions, it is hard to get such a nice form. So, we expand the loss function

up to the second order by the Taylor expansion:

$$\text{obj}^m \approx \sum_{i=1}^{n} [l(y_i, \hat{y}_i^{m-1}) + g_i t_m(x_i) + \frac{1}{2} h_i t_m^2(x_i)] + \Omega(t_m) + \text{constant}, \tag{1.17}$$

where the g_i and h_i are

$$g_i = \partial_{\hat{y}_i^{m-1}} l\left(y_i, \hat{y}_i^{m-1}\right) \tag{1.18}$$

$$h_i = \partial^2_{\hat{y}_i^{m-1}} l\left(y_i, \hat{y}_i^{m-1}\right). \tag{1.19}$$

If we remove all the constants in Equation (1.17), the specific objective function at step m is

$$\begin{aligned} \text{obj}^m &\approx \sum_{i=1}^{n} \left[g_i w_{q(x_i)} + \frac{1}{2} h_i w^2_{q(x_i)} \right] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \\ &= \sum_{j=1}^{T} \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T \\ &= \sum_{j=1}^{T} \left[G_j w_j + \frac{1}{2} \left(H_j + \lambda \right) w_j^2 \right] + \gamma T, \end{aligned} \tag{1.20}$$

where $I_j = \left\{ i \mid q(x_i) = j \right\}$ is the set of indices of data points assigned to the j-th leaf, $G_j = \sum_{i \in I_j} g_i$ and $H_j = \sum_{i \in I_j} h_i$. In the last step of the Equation (1.20), we changed the objective equation into a quadratic form. So the best w_j for a given structure $q(x)$ is:

$$w_j^* = -\frac{G_j}{H_j + \lambda}. \tag{1.21}$$

The minimal value of the objective function at step m is:

$$\text{obj}^* = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T. \tag{1.22}$$

Equation (1.22) can be used to measure how good a tree structure $q(x)$ is.

Since we know how to measure how good a tree is, we can use the greedy strategy to grow the tree. For a new tree in the forest, it will start with 0 depth. For each leaf node of the new tree, we choose the split with the highest gain scores, where the gain score is defined by:

$$\text{Gain} = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma. \tag{1.23}$$

We repeat this procedure until the stopping condition is reached.

So, the final output of the tree ensemble model in XGBoost is:

$$\hat{y}_i = \sum_{k=1}^{K} t_k(x_i), t_k \in \mathscr{T}. \tag{1.24}$$

1.3 Reinforcement Learning

In this section, an overview of the general reinforcement learning and tree-based methods framework will be given.

Reinforcement learning problem is a category of sequential decision making problem. In reinforcement learning problem, we are trying to train an agent to learn what to do, how to map situations to actions, so as to maximize a numerical reward signal in a given environment. Each action taken by the agent will give an effect to the environment. Once the environment received an action from the agent, it will send a single number (reward) back to the agent. Notice, in here, the reward may not be given immediately, and it may be a delayed reward. The problem of reinforcement learning uses the idea from dynamical systems theory, more specifically, the optimal control of Markov decision processes [Sut18].

1.3.1 Markov decision processes

Markov decision processes (MDPs) are a classical formalization of sequential decision making. The key components of a Markov decision processes are: a set of state space X, a set of action space U, a set of reward R, a reward function $r(x,u)$ and a transition function $f(x,u)$. When applying an action $u \in U$ on a state $x \in X$ at time t, the agent receives a numerical reward $r = r(x,u)$. Then the environment will move to a new state x' based on the action u and the state transition function $p(x,u,x')$. Most of the MDPs are finite dimensional problems, which mean that both X and U are finite sets. However, there are some MDPs problems with continuous or countably infinite state or action sets whose exact solutions are only possible in special cases. Here, we mainly focus on the finite MDPs problems.

The agent is the learner and decision maker in MDPs. Given a time point t, the goal of the agent is to find an optimal policy $\mu(x_t, u_t)$ that can maximize the expected discounted reward:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{1.25}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the discount rate. The policy $\mu : X \to U$ is a mapping from a state space X to an action space U.

If the agent follows a fixed policy μ and starts with a state x, then we denoted it as $V_\mu(x)$, the so-called it value function. $V_\mu(x)$ is the expected return when starting in x and following μ thereafter, it is defined as:

$$V_\mu(x) = E_\mu\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|x_t = x\right], \tag{1.26}$$

where $E_\mu[\cdot]$ denotes the expected value of a random variable given that the agent follows policy μ and t is any given time step. So the optimal expected return for a state x can be written as:

$$V_*(x) = \max_\mu V_\mu(x) = \max_\mu E_\mu\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|x_t = x\right]. \tag{1.27}$$

Then, we can use the Bellman equation [Dix90] to write $V_*(x)$ in the recursive form:

$$V_*(x) = \max_{u\in U} E\left[r(x,u) + \gamma V_*(x_{t+1})|x_t = x, u_t = u\right], \tag{1.28}$$

for all $x \in X$. From the definition above, the optimal policy $\mu_*(x)$ is given by:

$$\mu_*(x) = \operatorname*{argmax}_{u\in U}\left[r(x,u) + \gamma V_*(x_{t+1})|x_t = x, u_t = u\right]. \tag{1.29}$$

Similarly, we can define the value of taking action u in the state x under a policy μ, denoted by $Q_\mu(x,u)$, called Q-function, as the expected return starting from x, taking the action u, then following policy μ thereafter:

$$Q_\mu(x,u) = E_\mu\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|x_t = x, u_t = u\right]. \tag{1.30}$$

The optimal Q-function is given by:

$$Q_*(x,u) = \max_\mu Q_\mu(x,u) = \max_\mu E_\mu\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|x_t = x, u_t = u\right]. \tag{1.31}$$

Similarly, we can rewrite the Q-function in the recursive form:

$$Q_*(x,u) = E\left[r(x,u) + \gamma \max_{u'} Q_*(x_{t+1}, u')|x_t = x, u_t = u\right]. \tag{1.32}$$

Once we get the Q-function, the optimal value function can be expressed as:

$$V_*(x) = \max_{u\in U} Q_*(x,u) \tag{1.33}$$

and the optimal policy is given by:

$$\mu_*(x) = \underset{u \in U}{\operatorname{argmax}} Q_*(x, u). \tag{1.34}$$

As we shall see, after we obtain the Q-function, we can compute the best optimal policy without requiring knowledges of the reward and state transition functions [Sut18]. But in some cases, the Q-function may be difficult to obtain.

1.3.2 Dynamic programming

In this section, we assume that the environment is a finite MDP. It means that the state space X, action space U and the reward space R are all finite sets. Once we know the state transition probabilities $p(x, u, x')$ and the reward function $r(x, u)$, we can use dynamic programming methods to compute the optimal value function $V_*(x)$. The two most basic approaches are called policy iteration and value iteration [Bel57].

1.3.2.1 Policy Iteration

Policy iteration is a basic method for solving finite MDPs. It contains two distinct phases of policy evaluation and policy improvement. The details of policy iteration are shown in Algorithm 1. The idea of policy evaluation is to use the recursive formulation of the value function (1.28) to compute the value function V_μ for an arbitrary policy μ. Our final goal is to find an optimal policy μ_*. So, by applying policy improvement after policy evaluation, we can find a better policy to update our current policy μ.

Algorithm 1: Policy Iteration

Input: A fully-specified finite MDP: (X, U, p, R, γ).

begin Initialize V_0 and μ_0 arbitrarily

repeat

Policy Evaluation

repeat

For each x in X do

$$V_{k+1}(x) \leftarrow \sum_{x' \in X} p(x, \mu_k(x), x') \big[r(x, \mu_k(x)) + \gamma V_k(x') \big]$$

until *The convergence criteria is satisfied*;

Policy Improvement

For each $x \in X$ do

$$\mu_{k+1}(x) \leftarrow \underset{u \in U}{\operatorname{argmax}} \sum_{x' \in X} p(x, u, x') \big[r(x, u) + \gamma V_{k+1}(x') \big]$$

until $\mu_{k+1}(x) = \mu_k(x), \forall x \in X$;

end

Output: Optimal policy $\mu_*(x)$

Because a finite MDP has only a finite number of policies, that is $|U|^{|X|}$ possible policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations. In the policy iteration, each policy is guaranteed to be a strict improvement compared with the previous one, unless the policy is already optimal. Policy iteration often converges in surprisingly few iterations [Sut18]. But each iteration involves complex computation, because it needs to sweep through the state set in each iteration [Put14].

1.3.2.2 Value Iteration

In the policy iteration, we don't need to wait for the exact convergence to obtain the optimal policy. In fact, the policy evaluation step of the policy iteration can be truncated in several ways without losing the convergence guaranteed by the policy iteration [Sut18]. We called this approach the value iteration. The value iteration is another commonly used method of computing an optimal MDP policy and its value function. The details of the value iteration are shown in Algorithm 2.

Algorithm 2: Value Iteration

Input: A fully-specified finite MDP: (X, U, p, R, γ).

begin Initialize V_0 arbitrarily

repeat

For each x in X do

$$V_{k+1}(x) \leftarrow \max_{u \in U} \sum_{x' \in X} p(x,u,x') \big[r(x,u) + \gamma V_k(x') \big]$$

until *The convergence criteria is satisfied*;

end

Output: An approximate of value function $\hat{V}(x)$; a deterministic policy $\mu \approx \mu_*(x)$ such that

$$\mu(x) = \operatorname*{argmax}_{u \in U} \sum_{x' \in X} p(x,u,x') \big[r(x,u) + \gamma \hat{V}(x') \big]$$

The iteration algorithm can be terminated when some convergence criteria is satisfied. The most common way is:

$$\max_{u \in U} |V_{k+1}(x) - V_k(x)| < \epsilon, \tag{1.35}$$

where $\epsilon > 0$ is an error tolerance. The error tolerance is problem dependent. Theoretical results show that if equation (1.35) is satisfied in a finite MDP problem, the difference between the current approximate value function and the optimal value function will satisfied [Wil93]:

$$\max_{u \in U} |V_*(x) - V_{k+1}(x)| < \frac{2\epsilon\gamma}{1-\gamma}. \tag{1.36}$$

In practice, it is often the case that the optimal policy is discovered well before the value function converges [Kae96]. In summary, the value iteration is a flexible approach to compute the value function and the optimal policy.

1.3.3 Exploitation and exploration

The trade-off between exploitation and exploration is one of the challenges that comes up in reinforcement learning. The exploitation means that the agent is urgent to maximize the expected reward based on the value function it has already learned by now. The exploration means that the agent needs to explore more situations in the state-action space to make better action selections. The agent can not make the best decision by only doing exploitation or exploration. The exploitation and exploration must be well balanced if the agent

wants to make the optimal decision to maximize the expected reward. This is the so called exploitation-exploration dilemma. This dilemma has been intensively studied by many researchers for many decades. It remains unresolved. But the common way for balancing them is to implement a stochastic policy that gives extra weight to the best known action to the agent.

Among those, ϵ-greedy policy is the most simple one. In this case, when the agent need to choose an action from the action space, it will take a random action from the action space with a small probability ϵ, otherwise, it will choose the action associated with highest expected reward based on the value function (or Q-function) it has already learned with probability $1-\epsilon$. The advantage of ϵ-greedy methods is that it can theoretically ensure the convergence of the value function and Q-function [Sut18].

Another commonly used stochastic policy is the softmax policy. If the agent follows the softmax policy, it will choose actions according to a modified Boltzmann distribution:

$$\mu(x) = \frac{e^{\frac{Q(x,u)}{\tau}}}{\sum_{u \in U} e^{\frac{Q(x,u)}{\tau}}}, \tag{1.37}$$

where τ is a positive parameter called the temperature [Sut18]. High temperatures cause the actions to be all equiprobable. Low temperatures cause a greater difference in selection probability for actions that differ in their value estimates, as $\tau \to 0$, the policy approaches the simple greedy policy.

Whether ϵ-greedy policy or softmax policy is better mainly depends on the specific applications. Both policies have only one parameter that must be set.

1.3.4 Model-based and model-free reinforcement learning

The reinforcement learning methods can be divided into two categories: model-based and model-free reinforcement learning. Model-based reinforcement learning methods require a model of the environment, such as dynamic programming and probabilistic inference for learning control, etc. While the model-free reinforcement learning methods do not require the knowledge of the model of the environment, i.e., the transition function or the reward function is not required. The common model-free reinforcement learning methods are Q-learning, temporal difference method, etc.

The primary component of the model-based methods is planning. Calculating a policy based on a model is called planning [Wie12]. Based on the model of the environment, the agent will make plans. Because the agent can interact with the model of environment before it really implements an action, this can dramatically improve the speed of sampling. A lot

of models behave linearly at least in the local proximity and require very few samples to learn them. Hence, once the transition function and the reward function are known, we can plan the optimal controls without further sampling. But the disadvantage of model-based methods is interacting with environment will make the problem much more complex. Also, the problem of model-bias exists in model-based methods.

The primary component of the model-free methods is learning. The agent learning the transition and reward model from interaction with the environment is called learning. Contrary to the model-based methods, the model-free methods need to do more sampling to gather more statistical knowledge about the unknown model. Because of the environment is unknown, so the agent needs to try all kind of different actions to have more explorations. Hence, the balance between exploitation and exploration is important here.

1.3.5 XGBoost based fitted Q iteration algorithm

Fitted Q iteration algorithm [Ern05] was proposed by Ernst et al. in 2005. Their work inspired by the online Q-learning method [Wat89]. By using the idea of fitted value iteration [Gor99] and based on the framework of kernel-based reinforcement learning [Orm02], they reformulated the Q-function determination problem as a sequence of kernel-based regression problems. Also, they introduced several tree-based regression algorithms, like Kd-tree [Ben75], pruned CART tree [Bre84], tree bagging [Bre96], extra-trees [Geu06] and totally randomized trees [Geu06], into the fitted Q iteration algorithm. Because the fitted Q iteration algorithm requires that the regression method must be able to model any Q-function during the iterations, the non-parametric methods such as tree based methods have a great flexibility on modeling those Q-functions [Ern05]. Also, tree-based methods can adapt to high dimensional space and have higher computational efficiency compared to kernel-based methods. Kalyanakrishnan & Stone shown that the fitted Q iteration algorithm can make efficient use of training data [Kal07]. Since in the medical applications data may be hard to collect, this feature of fitted Q iteration is very important in those applications. Also, the fitted Q iteration algorithm is well suitable for the reinforcement problems with continuous state and action spaces [Ant08].

The regular regression methods used in fitted Q iteration algorithm are kernel-based methods and tree-based methods. There is still other options here, for example using neural-network architectures as the regression. Martin Riedmiller in 2005, is the first researcher to successfully introduce the multi-layer perceptron to the fitted Q algorithm [Rie05]. There is also an application that used neural-network based fitted Q iteration to find the optimal drug strategy for anemia management [Mal11]. Recently, researchers are mainly focusing

on using deep neural networks for the Q learning.

However, we will focus on a state of the art tree-based method, which is called XGBoost [Che16]. XGBoost was proposed by Tianqi Chen in 2015. XGBoost, which is based on the CART algorithm, added the regularized terms into the loss function. It has been dominating applied machine learning and Kaggle competitions for structured or tabular data for a long time. It is an implementation of gradient boosted decision trees designed for speed and performance. It is scalable in all scenarios, which is one of the most important features. Also, it runs more than ten times faster than existing popular solutions on a single machine [Che16]. Most of the tree based method can find the split points when the data points are of equal weights. However, they are not equipped to handle weighted data. XGBoost has a distributed weighted quantile sketch algorithm to effectively handle weighted data. Compared with other tree-based method, XGBoost has a big improvement. Based on the great features of XGBoost, we expected that it will have a better performance compared with other regression methods in fitted Q iteration. In this section, for the first time, we introduced the XGBoost into fitted Q iteration algorithm. We called this method the XGBoost based fitted Q iteration algorithm.

The input of the XGBoost based fitted Q iteration algorithm is a set of four-tuples, that is (x_t, u_t, r_t, x_{t+1}). We denote the set of four-tuples by $\mathscr{F} = \{(x_t^l, u_t^l, r_t^l, x_{t+1}^l)\}_{l=1}^{\#\mathscr{F}}$. Among the four-tuples, x_t is the current system state at time t, u_t is the control action that the agent will take at time t, r_t is the instantaneous reward obtained when the agent takes the action u_t and x_{t+1} is the new state of the system at the time $t+1$. The recurrence equation used in this algorithm is:

$$Q_N(x,u) = r(x,u) + \gamma \max_{u' \in U} Q_{N-1}(f(x,u),u'), \quad \forall N \in \mathbb{N}_1, \tag{1.38}$$

where $Q_N : X \times U \to \mathbb{R}$ is the sequence function with $Q_0(x,u) \equiv 0$. The sequence function Q_N converges to the optimal Q-function (1.31) in infinity norm as $N \to \infty$ [Ern05]. At each iteration N, we compute the estimate function $\hat{Q}_N$ of the true Q_N function by iteratively using the recurrence equation:

$$\hat{Q}_N(x_t,u_t) = r_t + \gamma \max_{u' \in U} \hat{Q}_{N-1}(x_{t+1},u'). \tag{1.39}$$

Since we already have some numerical estimates of the sequence function Q_N, we can transfer the reinforcement learning problem here to be a batch supervised learning problem. Any regression method can be used here to approximate the sequence function Q_N.

In the XGBoost based fitted Q iteration algorithm, the dynamic transition function $f(x,u)$ and the reward function $r(x,u)$ are not necessary. As long as we have the set of

system trajectories $\mathscr{F} = \{(x_t^l, u_t^l, r_t^l, x_{t+1}^l)\}_{l=1}^{\#\mathscr{F}}$, we can use equation (1.38) to approximate the optimal Q-function $Q_*(x,u)$.

The details of the XGBoost based fitted Q iteration algorithm are shown in Algorithm 3.

Algorithm 3: XGBoost based fitted Q iteration algorithm

Input: A set of four-tuples system trajectories $\mathscr{F} = \{(x_t^l, u_t^l, r_t^l, x_{t+1}^l)\}_{l=1}^{\#\mathscr{F}}$ and the XGBoost regression algorithm

begin Set $N = 0$ and $\hat{Q}_N$ equal to zero everywhere on $X \times U$ for all $N \in \mathbb{N}_0$.

repeat

- $N \leftarrow N + 1$.
- Build the training set $TS = \{(i^l, o^l)\}_{l=1}^{\#\mathscr{F}}$ based on the function $\hat{Q}_{N-1}$ and on the all four-tuples as following:

$$\left\{\left(\overbrace{(x_t^l, u_t^l)}^{i^l}, \underbrace{r_t^l + \gamma \max_{u \in U} \hat{Q}_{N-1}(x_{t+1}^l, u)}_{o^l}\right)\right\}_{l=1}^{\#\mathscr{F}}$$

- Apply the XGBoost regression algorithm to the training set TS to get the function $\hat{Q}_N(x,u)$ which is the approximation of sequence function $Q_N(x,u)$.

until *Stopping conditions are reached*;

end

Output: The last estimate function $\hat{Q}_N(x,u)$, which is an approximate of the optimal Q-function $Q_*(x,u)$

Once we obtain the estimate function $\hat{Q}_N(x,u)$, the estimate of the optimal policy can be computed by:

$$\mu_N(x) = \operatorname*{argmax}_{u \in U} \hat{Q}_N(x,u). \tag{1.40}$$

The policy $\mu_N(x)$ is an estimation of the true optimal policy $\mu_*(x)$, which is also the final result of the XGBoost based fitted Q iteration algorithm.

The stop condition decides at which iteration the process is to be terminated. A simple way is to define a maximum number of iterations. Another approach is to assume that the infinite norm of the reward function $r(x,u)$ in MDP is bounded by some positive constant B_r. Also with the discount γ given, we can fix N to get a desired level of accuracy [Ern05]. Another way to set up the stop condition is to check on the distance between the mean

of all values of $\hat{Q}_N$ and $\hat{Q}_{N-1}$. When the distance between them is small enough, we can terminate the iteration. But for some supervised learning algorithms, there is no guarantee that the sequence of $\hat{Q}_N$ functions converges. So this stop condition may not work in those cases.

The convergence of the fitted Q iteration depends on the regression method used in the algorithm. If you use the kernel-based method as the regression method, such as the k-nearest-neighbors method, partition and multi-partition methods, locally weighted averaging, linear, and multi-linear interpolation, you can guarantee the convergence in fitted Q iteration algorithm [Ern05]. If you use tree-based regression method, you need to keep the structures of trees in your regression method unchanged through out all iterations. For Kd-Tree method, the structure of trees stays the same at every iteration in fitted Q iteration algorithm. But for random forest, totally randomized trees and extra-trees, the structure of trees changes in every iteration. So it can not guarantee the convergence of the fitted Q iteration algorithm. In order to guarantee the convergence, we need to build the trees in the first iteration and then keep their structures and only refresh predictions at terminal nodes at subsequent iterations. The tree structures are therefore kept constant from one iteration to the other and this will ensure convergence.

Unfortunately, because the way XGBoost builds the trees, it is totally different with other tree-based methods. For the XGBoost based fitted Q iteration algorithm, there is no theoretical proof for the convergence of this method. This is an open problem that needs to be studied in the future. However, contrary to many parametric approximation schemes, XGBoost does not lead to divergence to infinity problem [Ern05]. And the good thing is that, even though the XGBoost based fitted Q iteration algorithm can not strictly guarantee the convergence of the sequence functions Q_N, it performs better than extra-trees based fitted Q iteration in HIV problem, numerically.

1.4 HIV Infection and STI Treatment

1.4.1 Introduction of HIV

HIV stands for the human immunodeficiency viruses. HIV infection is an acute fatal disease with first cases reported in the United State in June 1981. Today, it is a manageable chronic disease [Tea07]. The antiretroviral treatment (ART) is the recommended way to reduce the amount of virus (or viral load) in the HIV infected patient's blood and body fluids. ART is usually taken as a combination of 3 or more drugs. Structured treatment interruption (STI) is an alternative therapy for HIV infected patients [Gul02]. STI may be used to reduce side

effects of medications, to enhance a medication's effectiveness when restarted, or as a step towards stopping treatment altogether. But how to design a proper STI strategy for HIV patients is still a difficult problem.

1.4.2 Nonlinear ODE model of HIV infection dynamics

The basic model for many mathematical studies of HIV-1 dynamics is proposed by [Per96]. In 1997, Wein et al. developed a model to track the dynamics of uninfected and infected CD4+ T-cells and viral loads while allowing for virus mutations [Wei97]. In 2001, Callaway et al. examined several models to gain insight into the mechanisms responsible for sustained low viral loads [Cal02]. In 2005, Adams et al. build a nonlinear ODE model for HIV infection by investigating single drug (RTI) control [Ada05]. After that, Adams et al. updated the ODE model by investigating the multidrugs therapies-structured treatment interruptions (STI), containing RTIs and PIs. The system of ODEs describing the compartmental infection dynamics is given by [Ada04]:

$$\begin{aligned}
\text{Type 1 target: } \dot{T}_1 &= \lambda_1 - d_1 T_1 - (1-\epsilon_1) k_1 V T_1 \\
\text{Type 2 target: } \dot{T}_2 &= \lambda_2 - d_2 T_2 - (1-f\epsilon_1) k_2 V T_2 \\
\text{Type 1 infected: } \dot{T}_1^* &= (1-\epsilon_1) k_1 V T_1 - \delta T_1^* - m_1 E T_1^* \\
\text{Type 2 infected: } \dot{T}_2^* &= (1-f\epsilon_1) k_2 V T_2 - \delta T_2^* - m_2 E T_2^* \\
\text{Virus: } \dot{V} &= (1-\epsilon_2) N_T \delta (T_1^* + T_2^*) - c V \\
&\quad - [(1-\epsilon_1)\rho_1 k_1 T_1 + (1-f\epsilon_1)\rho_2 k_2 T_2] V \\
\text{Immune effectors: } \dot{E} &= \lambda_E + \frac{b_E (T_1^* + T_2^*)}{(T_1^* + T_2^*) + K_b} E - \frac{d_E (T_1^* + T_2^*)}{(T_1^* + T_2^*) + K_d} E - \delta_E E
\end{aligned} \tag{1.41}$$

with specified initial values for $T_1, T_2, T_1^*, T_2^*, V, E$ at time $t = t_0$. This ODE model contain six variables : T_1 is the number of healthy CD4$^+$ T-lymphocytes, T_2 is the number of healthy macrophages, T_1^* is the number of infected CD4$^+$ T-lymphocytes, T_2^* is the number of infected macrophages, V is the number of free virus particles, E is the number of HIV-specific cytotoxic T-cells. Those 6 variables are key components observed in clinical data sets. The details of all parameters used in this model are shown in Table 1.2 [Ada04]. It gives the definitions as well as numerical values for the parameters in the HIV model.

Table 1.2 The parameters used in the model (1.41). The superscripts * denote estimated from human data and ** denote those estimated from macaque data.

Parameter	Value	Units	Description
d_1	0.01**	$\frac{1}{day}$	target cell type 1 death rate
d_2	0.01**	$\frac{1}{day}$	target cell type 2 death rate
λ_1	10000	$\frac{cells}{mL \cdot day}$	target cell type 1 production (source) rate
λ_2	31.98	$\frac{cells}{mL \cdot day}$	target cell type 2 production (source) rate
ϵ_1	$\in [0,1)$	—	efficacy of reverse transcriptase inhibitor
ϵ_2	$\in [0,1)$	—	efficacy of protease inhibitor
k_1	8.0×10^{-7}	$\frac{mL}{virions \cdot day}$	population 1 infection rate
k_2	1.0×10^{-4}	$\frac{mL}{virions \cdot day}$	population 2 infection rate
f	$0.34 (\in [0,1))$	—	treatment efficacy reduction in population 2
δ	0.7*	$\frac{1}{day}$	infected cell death rate
m_1	1.0×10^{-5}	$\frac{mL}{cells \cdot day}$	immune-induced clearance rate for population 1
m_2	1.0×10^{-5}	$\frac{mL}{cells \cdot day}$	immune-induced clearance rate for population 2
N_T	100*	$\frac{virions}{cell}$	virions produced per infected cell
c	13*	$\frac{1}{day}$	virus natural death rate
ρ_1	1	$\frac{virions}{cell}$	average number virions infecting a type 1 cell
ρ_2	1	$\frac{virions}{cell}$	average number virions infecting a type 2 cell
λ_E	1	$\frac{cells}{mL \cdot day}$	immune effector production (source) rate
b_E	0.3	$\frac{1}{day}$	maximum birth rate for immune effectors
d_E	0.25	$\frac{1}{day}$	maximum death rate for immune effectors
K_b	100	$\frac{cells}{mL}$	saturation constant for immune effector birth
K_d	500	$\frac{cells}{mL}$	saturation constant for immune effector death
δ_E	0.1*	$\frac{1}{day}$	natural death rate for immune effectors

The functions ϵ_1, ϵ_2, respectively, represented the cycle on and off for RTI and PI. When RTI is cycling on, the value of ϵ_1 is 0.7; otherwise, the value of ϵ_1 is 0. When PI is cycling on, the value of ϵ_2 is 0.3; otherwise, the value of ϵ_2 is 0. In total, we have four different combination of treatments in this model: RTI and PI on, only RTI on, only PI on, RTI and PI off.

In the absence of treatment (meaning that we keep RTI and PI off all the time), the system of ordinary differential equations has three physical equilibrium points (here we ignore the non physical points for which one or more variables are negative). First, the

unstable equilibrium point is:

$$(T_1, T_2, T_1^*, T_2^*, V, E) = (10^6, 3198, 0, 0, 0, 10),$$

which represents an uninfected state. The other two equilibrium points are stable. One is the "healthy" locally stable equilibrium point:

$$(T_1, T_2, T_1^*, T_2^*, V, E) = (967839, 621, 76, 6, 415, 353108),$$

which has a small viral load, a high $CD4^+$ T-lymphocytes count and a high HIV-specific cytotoxic T-cells count. The other is the "non-healthy" locally stable equilibrium point:

$$(T_1, T_2, T_1^*, T_2^*, V, E) = (163573, 5, 11945, 46, 63919, 24),$$

for which T-cells are depleted and the viral load is very high. We will use the "non-healthy" point as the initial state of a patient when we generate the simulation data.

As we can see from the Table 1.2, the range of both the state variables and parameter values varies over several orders of magnitude, this will cause difficulty with the regression method when we applied the fitted Q algorithm. So we applied the $\log_{10}$ transformed system to both state variables and parameter values (except for ϵ_1 and ϵ_2) [Att17]. The new state variables are defined as:

$$x_i = \log_{10} \bar{x}_i(t, q), \tag{1.42}$$

where $\bar{x}_i = (T_1, T_2, T_1^*, T_2^*, V, E)$ represents the original, non-log transformed state. Also, for the parameter values we defined as $q \to \log_{10}(\bar{q})$, where $\bar{q}$ is the original parameter values. So the new system of equations is given by:

$$\dot{x}_i(t, q) = \frac{10^{-x_i}}{\ln 10} f_i(10^{x_i(q)}, 10^q), \tag{1.43}$$

where f_i is the right hand side of the original ODE system. By using this transformation, we can change both state variables and parameter values into similar magnitude. This will improve the accuracy of the regression. Also, it can decrease the computation time of the regression method and improves the speed of converge of the optimal strategy.

1.4.3 Optimal STI treatment formulation

The key components of the reinforcement learning are: the set of state space X, a set of action space U, a set of reward R, a reward function $r(x, u)$, and a transition function

$f(x,u)$. We will define these important components of the HIV problem in the followings.

In the HIV problem, we monitor each patient every five days. The state space X of the HIV problem is defined as:

$$X = \{(T_1, T_2, T_1^*, T_2^*, V, E) | T_1, T_2, T_1^*, T_2^*, V, E \in \mathbb{R}^+\}.$$

That is, the state of the patient can be summarized by the six-tuples $(T_1, T_2, T_1^*, T_2^*, V, E)$. We monitor the patient every five days by collecting the six-tuples quantities.

The action space of the HIV problem is defined as:

$$\begin{aligned} U &= \{\text{RTI \& PI on, only RTI on, only PI on, RTI \& PI off}\} \\ &= \{(\epsilon_1, \epsilon_2) | (0.7, 0.3), (0.7, 0), (0, 0.3), (0, 0)\}. \end{aligned}$$

We have 4 actions in our problem, that is, RTI and PI on, only RTI on, only PI on, RTI and PI off.

From the model described by (1.43), based on the $\log_{10}$ transformed ODEs system, we extracted and defined the instantaneous reward function $r(x,u)$ by:

$$r(x,u) = -Q10^V - R_1\epsilon_1^2 - R_2\epsilon_2^2 + S10^E, \tag{1.44}$$

where $x \in X$, $u \in U$; $Q = 0.1, R_1 = 20000, R_2 = 20000, S = 1000$; E is the number of HIV-specific cytotoxic T-cells, V is the number of free virus particles. When $\epsilon_1 = 0.7$, meaning the patient received the RTI treatment. When $\epsilon_2 = 0.3$, meaning the patient received the PI treatment.

We can treat the ODEs system of HIV model as the transition function. If we are given a current state of patient x and an action u applied on the current state, then we can solve the ODEs system to get the next state of patient x'.

Assuming that the patient is following the control policy μ, we can define the discounted infinite horizon cost function (also called total discount reward function) associated with μ by

$$J_\mu^\infty(x) = \lim_{N\to\infty} \sum_{t=0}^{N} \gamma^t r(x_t, \mu(x_t)), \tag{1.45}$$

where N is the number of state points for the patient, $\gamma \in [0,1]$ is the discount factor (when $\gamma = 0$, meaning that we only consider the current reward; when $\gamma = 1$, meaning that we treat the current reward and future rewards equally). The median time for patients to stop the STI treatment is around 6.5 months [Lis99; Ros00]. The rewards obtained in the first half year is much more important compared with other time periods. Hence, we set $\gamma = 0.98$ in

the HIV problem.

In summary, the final goal of optimal treatment for HIV patients is, by using the reinforcement learning method, to find an optimal strategy μ_* that maximizes $J_\mu^\infty(x)$ for all $x \in X$. Equation (1.34) is used to obtain the optimal STI strategy.

1.5 Data Generation and Experiments

In this thesis, we used fitted Q iteration algorithm to find the optimal STI strategies for HIV patients. Another approach is called the value iteration. The reason why we do not choose the value iteration is that value iteration is a model based method. It needs a model of the environment, such as a dynamical model and transition probabilities, etc. But in clinical settings, we only have clinical data sets of HIV patients. In order to allow our model to handle real clinical data in the future, we choose the fitted Q iteration algorithm.

In the fitted Q iteration algorithm the set of system trajectories $\mathscr{F} = \{(x_t^l, u_t^l, r_t^l, x_{t+1}^l)\}_{l=1}^{\#\mathscr{F}}$ is used to train our agent to learn the optimal strategy. However, in real life, the HIV clinical data is very hard to collect. In this work, we will simulate the real life clinical data to train our algorithm. It is noted that the nonlinear ODE model (1.41) was previously validated from real-life clinical data. In order to reduce the computation time and improve the speed of convergence, we applied the $\log_{10}$ transformed system the ODE model (1.41). We will use this $\log_{10}$ transformed ODE model to generate the simulation data.

In the data simulation process, we will monitor each simulated patient. Each simulated patient will start with the "non-healthy" state in the $\log_{10}$ transformed ODE model. The state of each simulated patient will be recorded every 5 days, i.e., the six-tuples $(T_1, T_2, T_1^*, T_2^*, V, E)$ will be recorded every 5 days. We will monitor each simulated patient for 1000 days. This means that for each simulated patient, we will have a set which contains 200 six-tuples states. We will generate the simulation data iteratively.

To generate the first data set, we choose 30 simulated patients in "non-healthy" steady-state. For each simulated patient, the state of his/her will be recorded every five days. At the same time, a new type of treatment will randomly selected in the action space U and applied to this patient. With each action that we choose, we can get an instantaneous reward by the reward function. We keep monitoring the simulated patient for 1000 days. In the end, by monitoring each patient for 1000 days (i.e., one trajectory), we can get a set $\{(x_t, u_t, r_t, x_{t+1}) | t = 0, 1, 2, \cdots, 198, 199\}$, which contains 200 samples (x_t, u_t, r_t, x_{t+1}). We called this set the trajectory set of simulated patient. Hence, by monitoring 30 simulated patients for 1000 days, we can get 30 trajectory sets, which includes 6000 samples. So, the

first data set can be written as

$$\mathcal{F}_1 = \{(x_t^l, u_t^l, r_t^l, x_{t+1}^l) | t = 0, 1, \cdots, 199\}_{l=1}^{30}.$$

Next, we need to choose the regression method in fitted Q algorithm. Here, we will use two methods: Extra-trees based fitted Q iteration and XGBoost based fitted Q iteration to compute the optimal strategy for HIV problem and to compare their performances.

For extra-trees based fitted Q iteration, in order to get the best result, we set $K = 6$, since our state variable contains six-tuples. Since we have a total of 4 different treatments to each patient, we will have 4 different sequence function $Q_N(x, u)$ according to the action $u \in U$. Thus, we need to build 4 different ensemble forests to approximate the sequence functions. For each ensemble forest, we set $M = 50$, this means that each forest contains 50 different trees. In order to get the fully developed trees, we set $n_{\min} = 2$. In the HIV problem, we set up two stop conditions for extra-trees based fitted Q iteration. First, we set up the maximum number of iterations to be 400. Second, we measure the distance between the means of all the values of $\hat{Q}_N$ and $\hat{Q}_{N-1}$, if the distance between them is less than 0.005, then we stop the iteration. If one of those stop conditions is satisfied, we stop the iteration. In order to guarantee the convergence of $\hat{Q}_N$ function, we will fix the structures of all trees in our algorithm and only refresh predictions at terminal nodes for another ten iterations when either stop condition is reached.

For XGBoost based fitted Q iteration, we used the python package of XGBoost version 0.90. There are three types of parameters in XGBoost: general parameters, booster parameters and learning task parameters. For general parameters, we choose booster as gbtree, the others stay default. For booster parameters, we set the learning rate as 0.1, the maximum tree depth for base learners as 6 and the number of trees to fit as 200, the others stay default. For learning task parameters we set all parameters as default. The stopping condition for XGBoost based fitted Q iteration is similar to the extra-trees based fitted Q iteration. The first stop conditions stay the same, that is, set the maximum number of iterations to be 400. But in order to guarantee the convergence of XGBoost based fitted Q iteration , we need to set another stop condition. The second stop condition is the distance between the means of all the values of $\hat{Q}_N$ and $\hat{Q}_{N-1}$ to be less than 0.005 and at the same time the infinity norm of all the values of $\hat{Q}_N$ and $\hat{Q}_{N-1}$ to be less than 0.05. The XGBoost based fitted Q iteration needs to satisfy two thresholds in the second stop condition simultaneously, or reaches the maximum number of iteration before it stopped.

Until now, we already set up all the methods and parameters in the fitted Q iteration algorithm. Next, we will enlarge our simulated data set and compute the optimal policy for

the HIV problem. The following steps are the same in both XGBoost based fitted Q iteration and extra-trees based fitted Q iteration. So, we just describe the details of the XGBoost based fitted Q iteration here.

We use the first data set $\mathscr{F}_1$ as the input of XGBoost based fitted Q iteration and start the iteration. When the stop condition described above was reached, we terminate the iteration. The output of XGBoost based fitted Q iteration is the estimated function $\hat{Q}_{N_1}(x,u)$, where N_1 is the number of iterations that this algorithm terminated. Hence, the first estimate of the optimal policy is:

$$\mu^1(x) = \underset{u\in U}{\operatorname{argmax}}\, \hat{Q}_{N_1}(x,u). \tag{1.46}$$

Once, we obtain the first estimate of the optimal policy, we can start the second step of the simulated data generation process. In the second step, similarly with the first step, we also choose 30 simulated patients with the "non-healthy" steady-state. Again, their states will be recorded every five days, and this process lasts for 1000 days. In the first step, we choose the treatment randomly from the action space U. However, in the second step, instead of choosing the drug cocktail randomly, we use the ϵ-greedy method to choose the treatment. This means that, for each simulated patient at each time point, there is 15% chance that we will choose the treatment randomly from the action space U, and there is 85% chance that we will use the treatment computed by the first estimate of the optimal policy $\mu^1(x)$ based on the state of the simulated patient at that time point. Assuming that the state of the simulated patient is x_t at the time point t, the treatment under the first estimate of the optimal policy $\mu^1(x)$ is $u_t = \mu^1(x_t)$. So, based on the ϵ-greedy method, at the time point t, 15% chance that we will select the treatment randomly, 85% chance that we will use the treatment $u_t = \mu^1(x_t)$. Under the ϵ-greedy method, by monitoring those 30 simulated patients for 1000 days, we have another 30 trajectory sets, the union of them is $\mathscr{F}_2^\epsilon$:

$$\mathscr{F}_2^\epsilon = \{(x_t^l, u_t^l, r_t^l, x_{t+1}^l) | t = 0,1,\cdots,199\}_{l=1}^{30}.$$

Hence, the second data set are the combination of $\mathscr{F}_1$ and $\mathscr{F}_2^\epsilon$:

$$\mathscr{F}_2 = \mathscr{F}_1 \cup \mathscr{F}_2^\epsilon = \{(x_t^l, u_t^l, r_t^l, x_{t+1}^l) | t = 0,1,\cdots,199\}_{l=1}^{60}.$$

There are 60 simulated patients and 12000 samples contained in the second data set. Once again, we applied the XGBoost based fitted Q iteration to the second data set. The output is the second estimate function $\hat{Q}_{N_2}(x,u)$, where N_2 is the number of iterations that this

algorithm terminated. Hence, the second estimate of the optimal policy is:

$$\mu^2(x) = \underset{u \in U}{\operatorname{argmax}} \hat{Q}_{N_2}(x, u). \tag{1.47}$$

In the process of the third data generation, similarly with the first and second steps, we choose another 30 simulated patients in "non-healthy" state. We also use the ϵ-greedy method here. The only difference here is that 85% chance that we will use the treatment computed by $\mu^2(x)$, instead of $\mu^1(x)$. After monitored the 30 simulated patients for 1000 days, we have another 30 trajectory sets of simulated patients, the union of them is $\mathscr{F}_3^{\epsilon}$. So, the third data set is $\mathscr{F}_3 = \mathscr{F}_2 \cup \mathscr{F}_3^{\epsilon}$.

By repeating the iteration of the data generation for 10 times, we obtain a series of data sets :$\mathscr{F}_1, \mathscr{F}_2, \cdots, \mathscr{F}_{10}$. The tenth data set is:

$$\mathscr{F}_{10} = \{(x_t^l, u_t^l, r_t^l, x_{t+1}^l) | t = 0, 1, \cdots, 199\}_{l=1}^{300}$$

in which we have 300 trajectories of simulated patients and 60000 samples. After applying the XGBoost based fitted Q iteration to the tenth data set, we obtain the tenth estimate of the optimal policy:

$$\mu^{10}(x) = \underset{u \in U}{\operatorname{argmax}} \hat{Q}_{N_{10}}(x, u). \tag{1.48}$$

The tenth estimate of the optimal policy $\mu^{10}(x)$ is the final result of the whole process, i.e., the best optimal policy.

1.6 Numerical Results

1.6.1 Baseline results

The baseline STI strategy is the patient receiving both RTI and PI treatments all the time. We will choose a simulated patient in the "non-healthy" state. The patient will be given the full treatment for 1000 days. By simulating the state equation (1.41), the states of the simulated patient in 1000 days are presented in Figure 1.4. Also, for comparison, Figure 1.4 contains the states of the simulated patient without any treatment in 1000 days.

Since the simulated patient here is starting with the "non-healthy" steady-state which is one of the three physical equilibrium points of the ODE system (1.41), the patient will stay at the "non-healthy" state if he/she does not received any treatment. The dash line in Figure 1.4 shows exactly what we described above.

The solid lines show the state of the simulated patient who received the fully treatment

during the 1000 days. Compared with the non treatment patient, the patient who received fully treatment has a much higher count of the uninfected T_1 cells, a slightly higher count of the uninfected T_2 cells, a slightly lower count of the infected T_1 cells and almost the same count of the infected T_2 cells. After first few days, the number of free virus particles V in the patient who received the fully treatment dropped dramatically. The number of immune effectors E reflects the response of the immune system. At the first 150 days, the immune system increase in responses to the increase in the HIV virus. But the number of immune effectors keep staying at a very low level after 250 days.

In summary, the baseline STI strategy (full treatment) keeps the viral load in control with a strong immune effector when it is compared with no treatment. We want to compare this baseline strategy with an optimal strategy.

1.6.2 Fitted Q iteration algorithm results

1.6.2.1 The first STI strategy

After applying the fitted Q iteration algorithm to the first data set $\mathcal{F}_1$, we can use the first estimate policy function to compute the first STI strategy. The first STI strategies computed by extra-trees based fitted Q iteration and XGBoost based fitted Q iteration are presented in Figure 1.5 and Figure 1.6, respectively.

Both STI strategies only have few days to turn off the RTI or PI treatment in the 1000 days. The turn off actions slightly improved the state of the simulated patient. As shown in Figure 1.7, compared with the baseline policy, the patients, who followed these two STI strageties, tend to have lower number of the free virus particles in their body. And the immune system of the patient, who follows the first strategy computed by XGBoost based fitted Q iteration, tends to have a long term responses to the HIV virus for around 200 days. After around 500 days, the free virus particles in both patients stay stable at a high level and correspondingly the immune effectors stay stable at a lower level.

The reason why the first STI strategy is not an optimal strategy is that the actions in the first data set were chosen randomly from the action space U. And in the first data set, we only have 30 trajectory of patients and 6000 samples. The data set is not large enough, i.e., not contains enough information for the fitted Q iteration algorithm to learn. This is the reason why we need to enlarge our simulation data set.

1.6.2.2 The seventh STI strategy

The seventh STI strategies computed by the extra-trees based fitted Q iteration and XGBoost based fitted Q iteration are presented in Figure 1.8 and Figure 1.9, respectively. There is a big

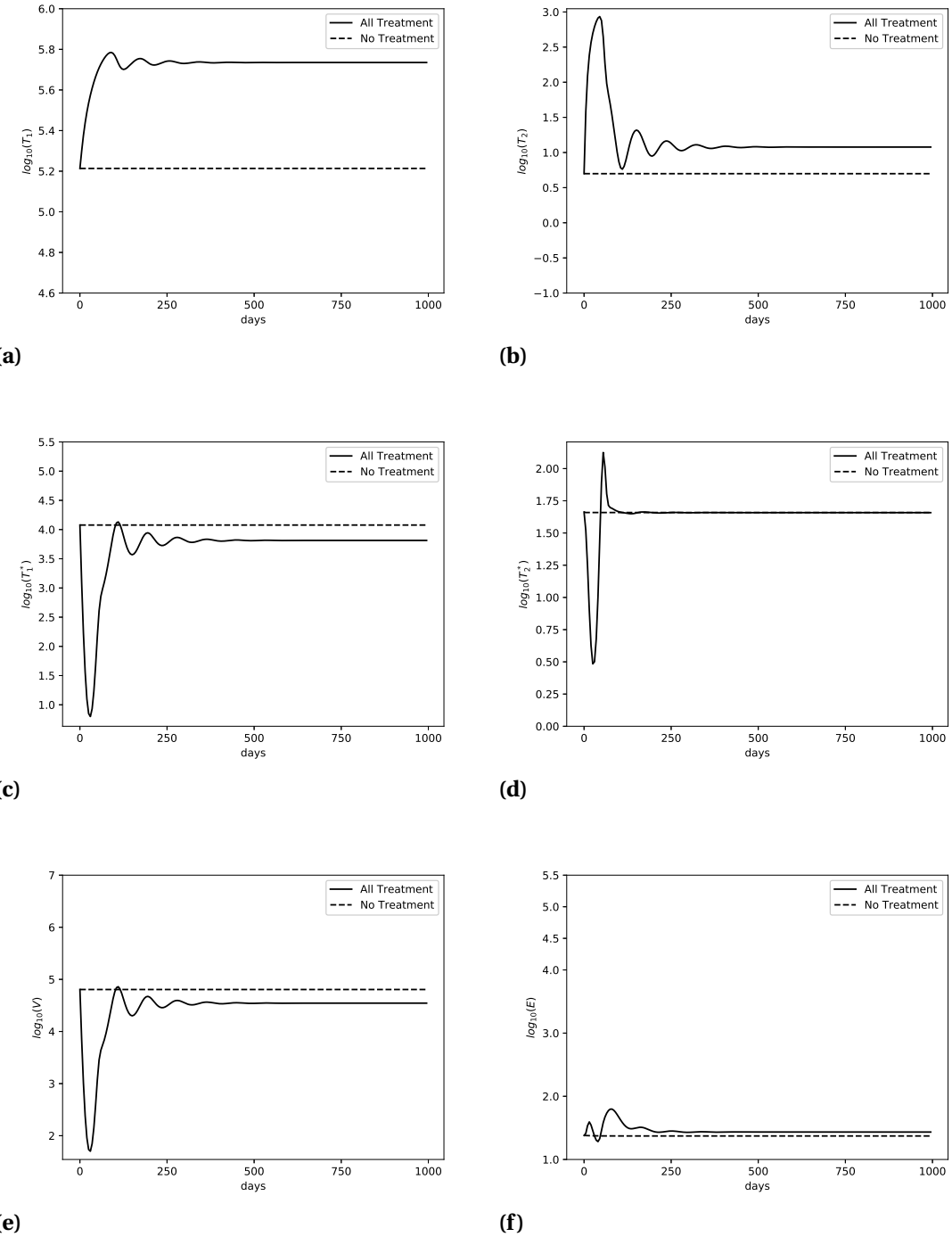
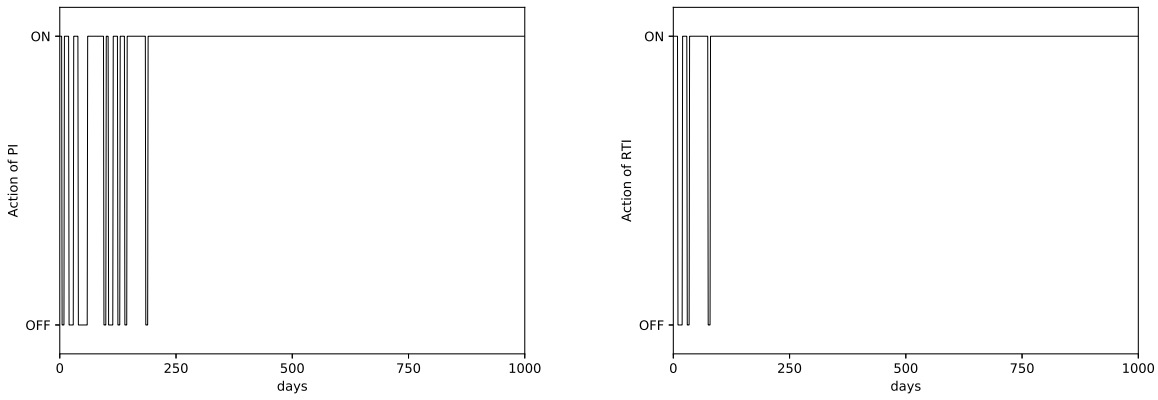
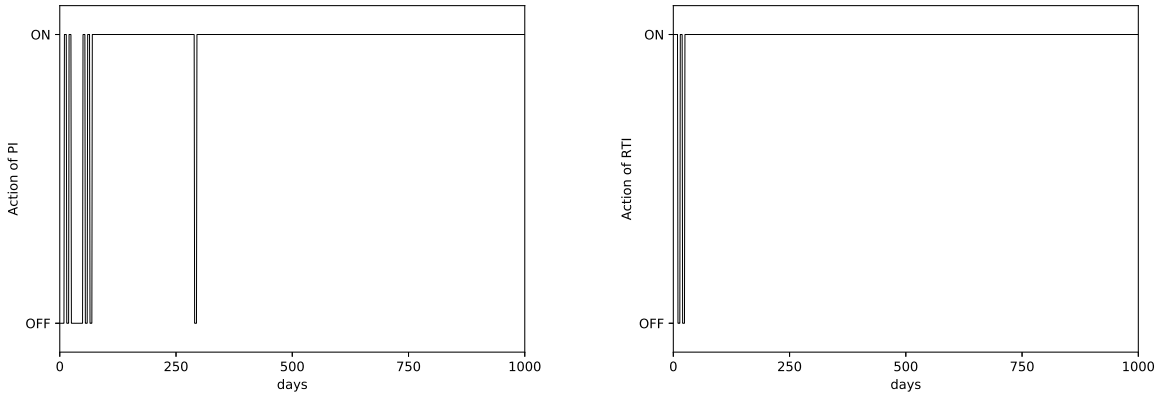


Figure 1.4 The six-tuples state of the simulated patients started with "non-healthy" state in 1000 days by following with the no treatment(– –) strategy ($\epsilon_1 = 0, \epsilon_2 = 0$), and with the fully treatment (–) strategy ($\epsilon_1 = 0.7, \epsilon_2 = 0.3$).



(a) **(b)**

Figure 1.5 The first STI strategy computed by the extra-trees based fitted Q iteration



(a) **(b)**

Figure 1.6 The first STI strategy computed by the XGBoost based fitted Q iteration

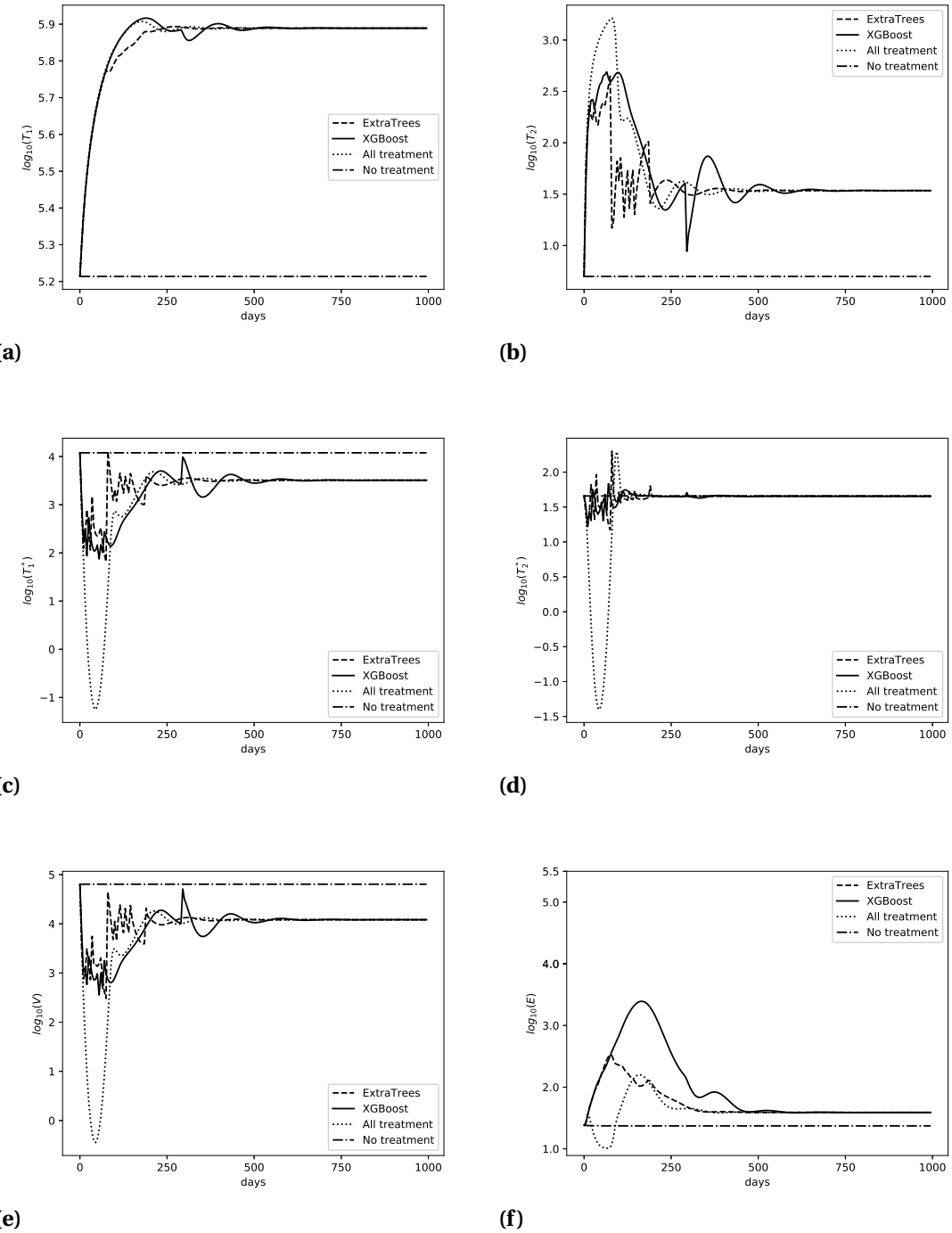
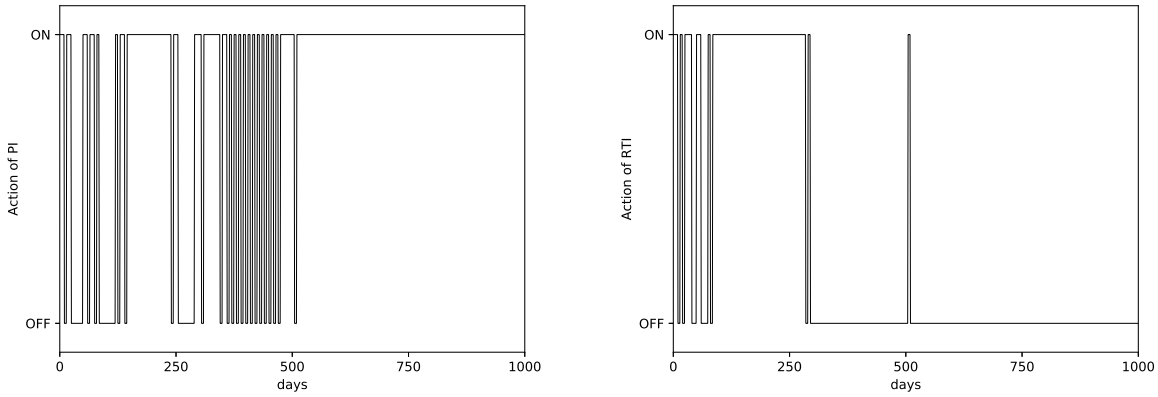


Figure 1.7 The six-tuples state of the simulated patients started with "non-healthy" state in 1000 days by following with the first STI strategy computed by the XGBoost based fitted Q iteration, and with the first STI strategy computed by the extra-trees based fitted Q iteration.

difference between the seventh STI strategy computed by extra-trees based fitted Q iteration and XGBoost based fitted Q iteration. The seventh STI strategy computed by extra-trees based fitted Q iteration can turn off the RTI treatment but needs to keep the PI treatment on all the time after around 500 days. However, the seventh STI strategy computed by XGBoost based fitted Q iteration, can turn off both RTI and PI treatment around 400 days.

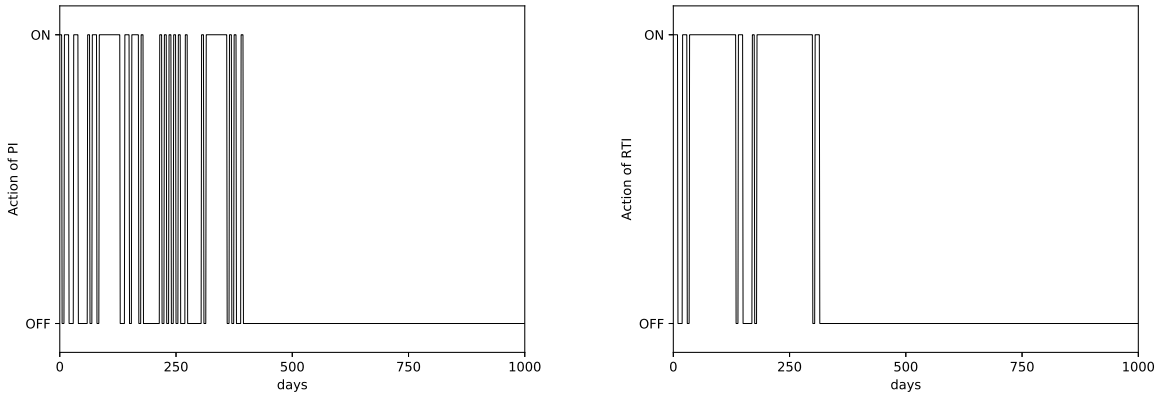


(a) **(b)**

Figure 1.8 The seventh STI strategy computed by the extra-trees based fitted Q iteration

The six-tuples state of the simulated patients started with "non-healthy" state in 1000 days by following with the seventh STI strategy computed by the XGBoost based fitted Q iteration and with the seventh STI strategy computed by the extra-trees based fitted Q iteration are shown in Figure 1.10. For both STI strategies, after 500 days, the counts of the healthy T_1 cells and the healthy T_2 cells are high. The number of the infected T_1 cells and the infected T_2 cells oscillating around a very low level, and the oscillation amplitude of the seventh STI strategy computed by the XGBoost based fitted Q iteration is much smaller compared with the one computed by the extra-trees based fitted Q iteration. The number of free virus particles oscillates around a very low level after 500 days and the oscillation amplitude of the seventh STI strategy computed by the XGBoost based fitted Q iteration is smaller. The number of the immune effectors in the patient who following the strategy computed by the XGBoost based fitted Q iteration is slightly higher than the one who following the strategy computed by the extra-trees based fitted Q iteration.

Compared with the baseline and the first STI strategies, both seventh strategies are much better. However, the seventh strategy computed by the XGBoost based fitted Q iteration



(a) **(b)**

Figure 1.9 The seventh STI strategy computed by the XGBoost based fitted Q iteration

is best in those strategies. The patient who following this strategy stays at the "healthy" steady-state without any treatments after around 400 days. While the patient, who follows the seventh strategy computed by the extra-tree based fitted Q iteration, needs to keep the PI treatment on all the time after around 500 days.

1.6.2.3 The optimal STI strategy

The final data set $\mathcal{F}_{10}$ includes 300 simulated patients with 300 trajectories of them, in total 60000 samples. This data set is big enough to contain all the information that we want the fitted Q iteration algorithm to learn. By using the best optimal policy function $\mu^{10}(x)$, we can get the optimal STI strategy for a patient who started with the "non-healthy" steady-state. The optimal STI strategies computed by extra-trees based fitted Q iteration and XGBoost based fitted Q iteration are presented in Figure 1.11 and Figure 1.12, respectively. We observe that both optimal STI strategies can turn off both RTI and PI treatments after around 400 days. The optimal STI strategy computed by the XGBoost based fitted Q iteration keeps the patient staying as a "healthy" state without any treatment after 380 days, while, it requires 400 days for the optimal STI strategy computed by the extra-trees based fitted Q iteration. Adams et al. used the optimal control theory to find the optimal STI strategy [Ada04]. The optimal strategy that they obtained could turn off both RTI and PI treatments after around 600 days. The optimal strategies that we obtained in this research could turn off the treatments after only 380 days, potentially saving treatment cost significantly.

We observe in Figure 1.13, although the STI strategy computed by the XGBoost based fitted Q iteration can turn off the both treatments 20 days ahead of the STI strategy computed

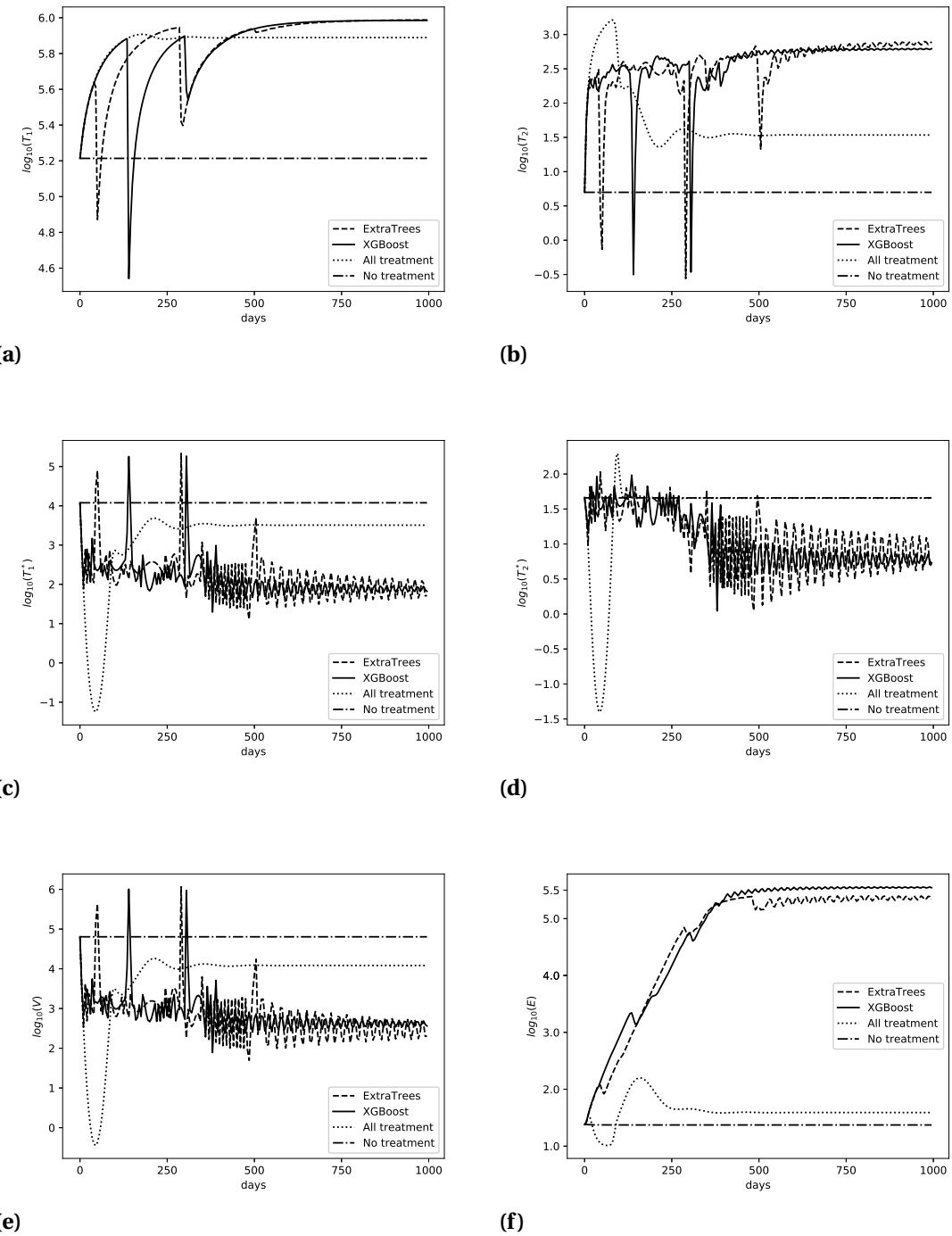
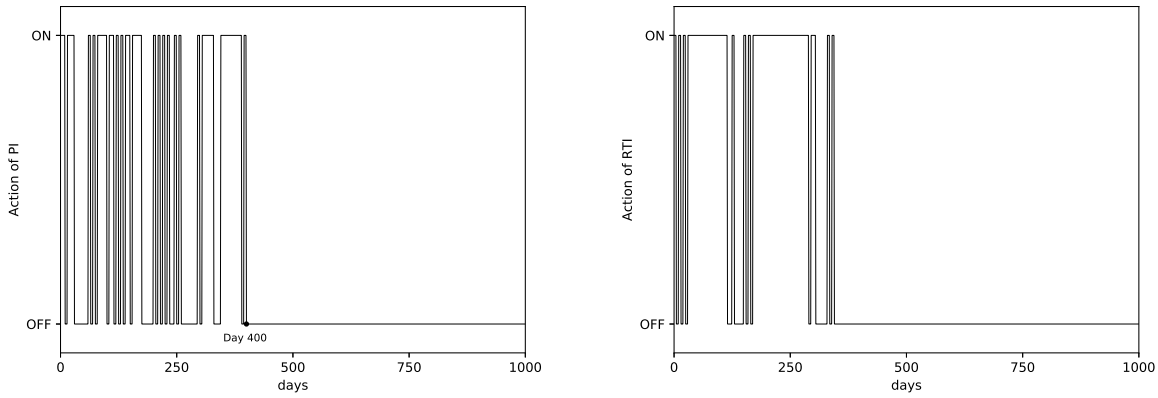
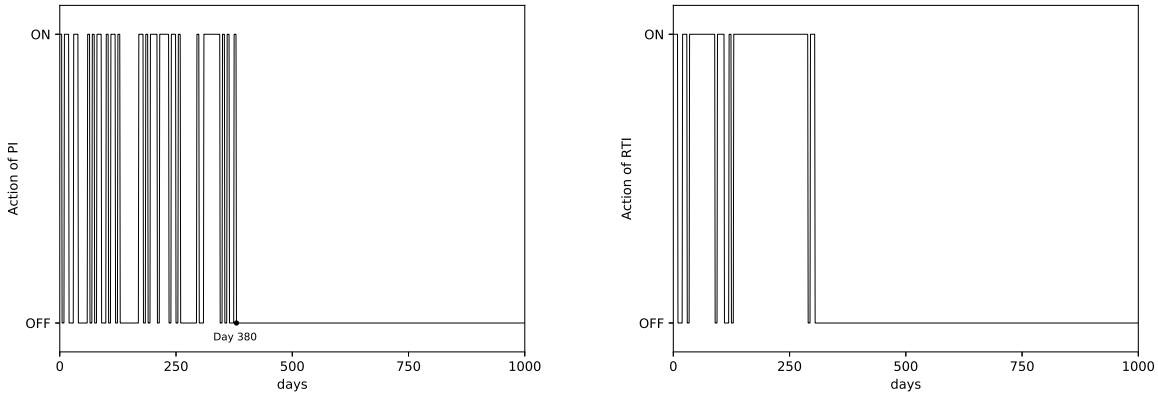


Figure 1.10 The six-tuples state of the simulated patients started with "non-healthy" state in 1000 days by following with the seventh STI strategy computed by the XGBoost based fitted Q iteration, and with the seventh STI strategy computed by the extra-trees based fitted Q iteration.



(a) **(b)**

Figure 1.11 The optimal STI strategy computed by the extra-trees based fitted Q iteration



(a) **(b)**

Figure 1.12 The optimal STI strategy computed by the XGBoost based fitted Q iteration

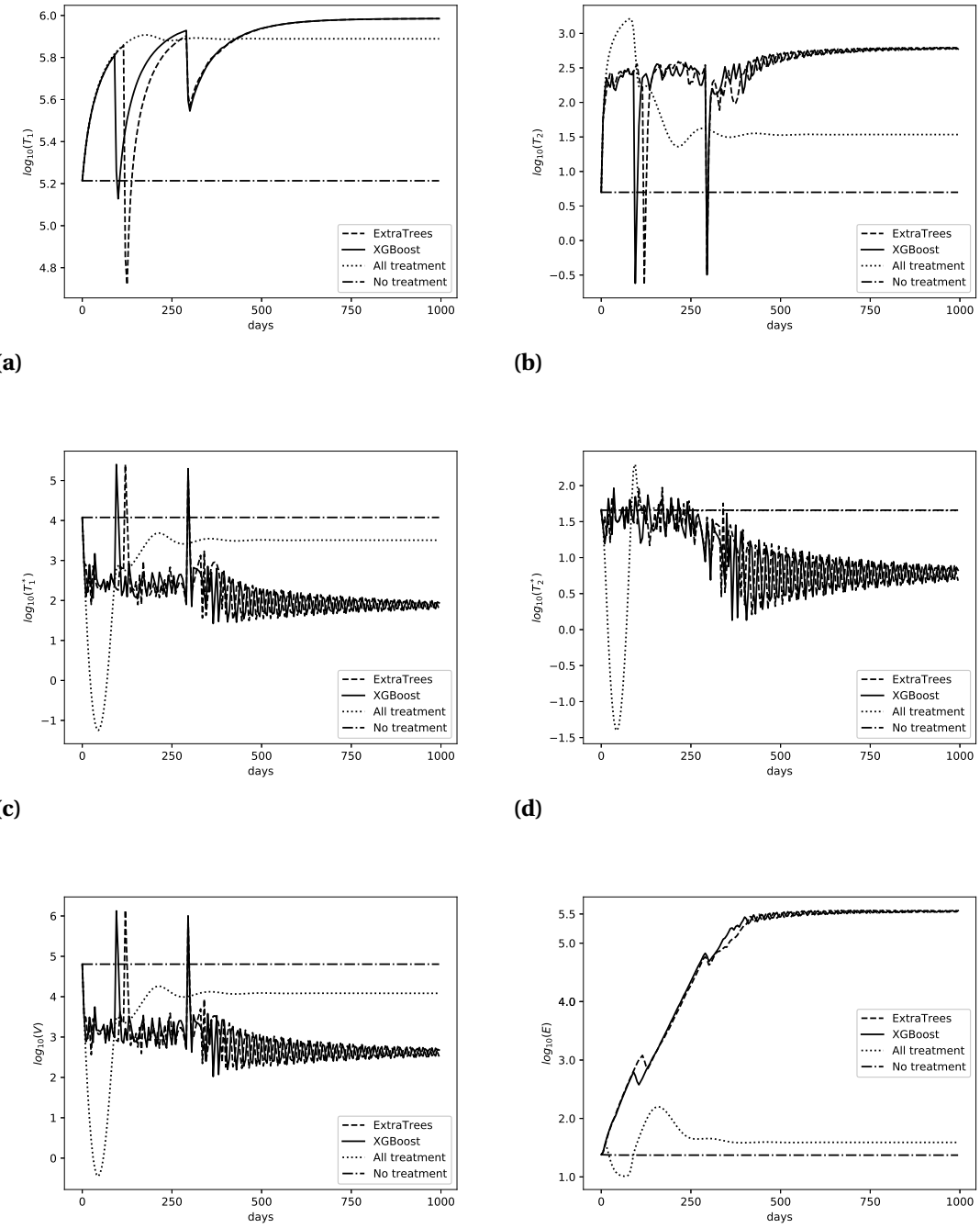


Figure 1.13 The six-tuples state of the simulated patients started with "non-healthy" state in 1000 days by following with the optimal STI strategy computed by the XGBoost based fitted Q iteration, and with the optimal STI strategy computed by the extra-trees based fitted Q iteration.

by the extra-trees based fitted Q iteration, they almost have the same performance. The patients following either of the optimal strategy can keep a very low level of viral load and a very high level of immune effectors at the end of the treatment. After around 500 days, the virus load in both patients remain less then 10^3. And the population of the infected T_1^* and T_2^* cells stays around a low level. At the same time, the counts of the health T_1 and T_2 cells keep at a high level. All of those are due to a very strong immune response. This means that both strategies can maximally reduce the number of free virus particles containing in the patient's body and can keep the immune system healthy allowing a strong immune response to the HIV virus. By following the optimal STI strategies, the patient can move from an infected state to a healthy state. However, compared with the extra-trees based fitted Q iteration, the XGBoost based fitted Q iteration obtained a better STI strategy for HIV.

Here, we define an acceptable STI strategy as the strategy that can help the patient moving from the "non-healthy" state to the "healthy" state and keep the patient staying at the "healthy" state without any RTI and PI treatment. Compared with the extra-trees based fitted Q iteration, the XGBoost based fitted Q iteration can find an acceptable STI strategy by using fewer patients data. This result is illustrated in Figure 1.14. The black squares and stars are the strategies computed by the XGBoost fitted Q iteration and the extra-trees based fitted Q iteration, respectively. The strategies above the dash line are the acceptable STI strategies. We observe that the XGBoost based fitted Q iteration can obtain an acceptable STI strategy by using only 210 patients data. However, it requires 270 patients data for the extra-trees based fitted Q iteration to get an acceptable STI strategy. The XGBoost based fitted Q iteration converges faster than the extra-trees based fitted Q iteration, because it has higher total discounted rewards by using the same number of patients data before an acceptable STI strategy was obtained.

Finally, it is noted that the computing time for the XGBoost based fitted Q iteration is around 7000 seconds. However, the computing time for the extra-trees based fitted Q iteration is around 24000 seconds. These two algorithms were running on the same desktop with the Intel 8700k processor, 32GB memory and Nvidia Geforce RTX 2080 Ti graphic card. Compared with extra-trees based fitted Q iteration, XGBoost based fitted Q iteration is much more computationally efficient.

1.7 Contributions

We are the first one who introduced the XGBoost regression method into the fitted Q iteration algorithm for finding the optimal treatment strategy for HIV patients. We have

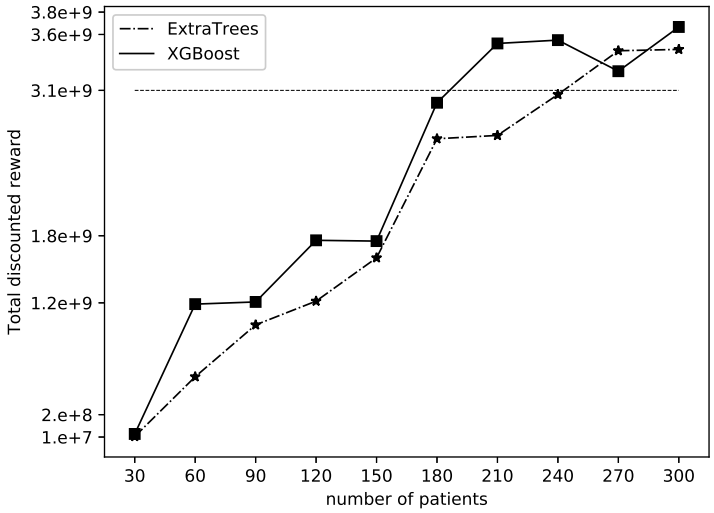


Figure 1.14 The influence of the number of simulated patients contained in the fitted Q iteration algorithms on the total discounted reward (in the first 1000 days) with respect to the computed strategies.

shown that the XGBoost based fitted Q iteration can obtain a better STI strategy compared with the extra-trees fitted Q iteration. Based on the high performance of XGBoost regression method, the XGBoost based fitted Q iteration is much more computationally efficient compared with extra-trees based fitted Q iteration.

We also proposed a new rule of stopping condition for the XGBoost based fitted Q iteration algorithm, that is, we considered both the distance defined by the mean and the infinity norm of all values of $\hat{Q}_N$ and $\hat{Q}_{N-1}$.

1.8 Future Work

Currently, we can only guarantee that the XGBoost fitted Q iteration algorithm numerically converges for HIV applications. But, there is not a theoretical proof for the convergence of the XGBoost fitted Q iteration algorithm. The theoretical convergence of the XGBoost fitted Q iteration algorithm is still an open problem. Once the convergence problems are being solved, we might apply the XGBoost fitted Q iteration algorithm to other diseases (such as radiation therapy and other chronic diseases, etc) to find the optimal treatment strategy.

In this chapter, the nonlinear ODE model of HIV infection dynamics that we used was validated by using the data collected by Rosenberg from patients studied at Massachusetts General Hospital. The real clinical data provides RNA viral load, CD4, and CD8 data for

102 patients. However, as shown in this research, the XGBoost based fitted Q iteration algorithm requires 210 patients data to obtain the optimal STI strategy. If we can collect enough clinical data in the future, it will be interesting to see if the XGBoost fitted Q iteration algorithm can maintain its high performance.

In our simulation data, the parameters were assumed to be the same for each simulated patient. However, the parameters may vary from one patient to another in clinical settings. To account for the uncertainty in model parameters, sensitivity analysis needs to be performed to study sensitivity of the optimal policy with respect to model parameters.

Finally, it is common with clinical data that, due to the range sensitivity limits of the assays used for collecting data, the clinical viral load will have upper and lower limit of quantification. The upper limit of quantification can be handled by repeatedly diluting the sample until it falls within a detection range [Att12]. However, the lower limit directly affects the observed data and a methodology is required to be able to use this data in our proposed approach for STI treatment. In machine learning, two popular approaches, Nearest Neighbor Hot-Deck and Expectation Maximization, can be employed to impute the data to gain precision. The hot-deck method imputes missing values by clustering the data and then missing values are replaced by values from similar complete samples. Expectation maximization imputes missing values by finding the maximum likelihood estimates and iterates until maximizing the log likelihood.

CHAPTER

2

IMAGE SEGMENTATION FOR EVOLUTIONARY GENETICS

2.1 Introduction

With the rapid development of artificial intelligence in last decades, many biological problems can be solved by using artificial intelligence. Image segmentation is one of the applications of artificial intelligence. There are many famous deep learning algorithms of image segmentation, such as Fully Convolutional Networks (FCN) [Lon15], U-Net [Ron15] DeepLab [Che17a], Mask R-CNN [He17] and You only look once (Yolo) [Red16], etc. Among them, U-Net is the method that focuses on the biomedical image segmentation. It performs very good on various biomedical segmentation problems. Based on the structure of U-Net, UNet++ was proposed by Zongwei Zhou in 2018 [Zho18]. UNet++ has even better performance compared with U-Net. In this chapter, we utilize the UNet++ algorithm to find the exponential blocks contained in the heat map of pairwise difference matrix, which was obtained from the sequence data of virus.

2.2 Neural Network

Neural network, also called artificial neural network (ANN), is one of the most popular machine learning techniques. Neural network simulates the mechanism of learning in biological organisms. It is a group of artificial neurons which inter connected with each other and uses a mathematical model for information processing based on a connectionism approach to computation. Neural network can be used to both supervised and unsupervised learning. It is the foundation of deep learning.

2.2.1 Single-layer perceptron

Single-layer perceptron is the simplest case of a neural network. It only has one input layer with several input nodes and one output layer with one output node. In the single-layer perceptron, a set of input data is directly mapped to an output node by using a generalized variation of a linear function. Figure 2.1 shows a structure of a single-layer perceptron with bias [Agg18].

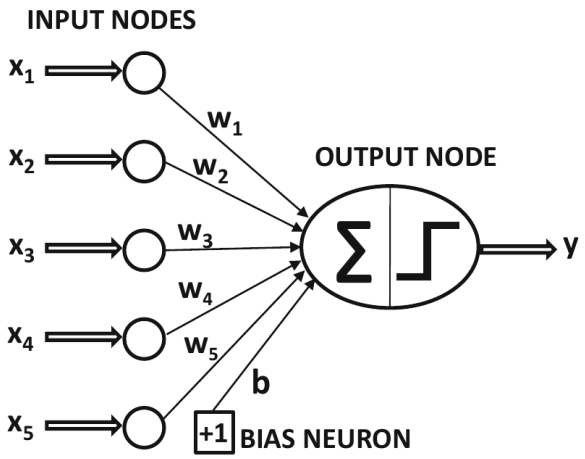


Figure 2.1 Single-layer perceptron with bias [Agg18].

Assuming that $\mathbf{x}$ is the input vector; $\mathbf{w}$ is the weight of the single layer; b is the bias and f is the activation function. The output of the single-layer perceptron with bias is:

$$y = f(\mathbf{x}^T\mathbf{w} + b). \tag{2.1}$$

Also, some single-layer perceptron does not has the bias term, so the output simply be-

comes:

$$y = f(\mathbf{x}^T \mathbf{w}). \tag{2.2}$$

2.2.2 Multi-layer perceptron

Multi-layer perceptron (MLP) is more commonly known as multi-layer neural network or artificial neural network. It can be used to both classification and regression problems. A multi-layer perceptron consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Figure 2.2 shows the structure of an example of a multi-layer perceptron with bias [Agg18]. This architecture is known as the feedforward neural network.

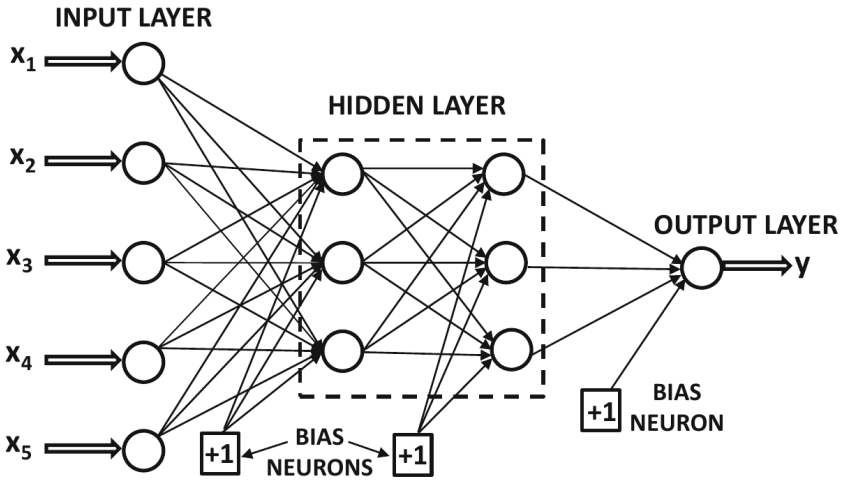


Figure 2.2 Multi-layer perceptron with bias [Agg18].

The number of hidden layers and the number of neurons contained in each hidden layer are the hyperparameters. You can choose the number of layers and number of neurons contained in each hidden layer differently for different problems. Every node in a hidden layer and output layer uses an activation function. Not like the single-layer perceptron which can only have one node in the output layer, the number of nodes contained in the output value for the multi-layer perceptron depends on the applications. The way to compute each node value and the output value is similar with the single-layer perceptron.

In a single-layer perceptron, the weights and bias are computed from the perceptron learning algorithm (PLA). In a multi-layer perceptron, the weights and bias are computed by minimizing the error function, which is defined as the difference between the network output values and the true values. This is an optimization problem that can be solved by

using a number of well-known optimization algorithms.

2.2.3 Activation functions

Activation function is used to decide the output of nodes in hidden layers and output layers. Activation function needs to be chosen properly for different problems. The followings are some common activation functions.

Step function is a classical active function. It is defined as:

$$f(x) := \begin{cases} 0, \text{ if } x < 0 \\ 1, \text{ if } x \geq 0. \end{cases} \tag{2.3}$$

It can be used to map to binary outputs. Hence, it is usually used in classification problem. The plot of the step function is shown in Figure 2.3.

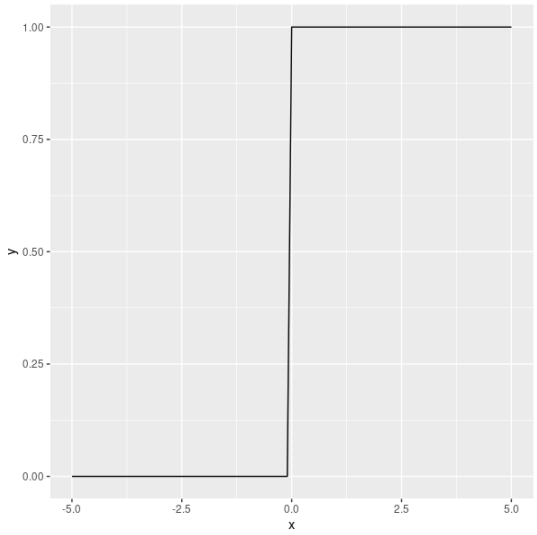


Figure 2.3 Step function.

Sigmoid function is defined as:

$$f(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}. \tag{2.4}$$

It returns the values in the range 0 to 1, which is helpful in performing computations that should be interpreted as probabilities. It is especially useful for models where we have to predict the probability as an output. Figure 2.4 shows the graph of the sigmoid function.

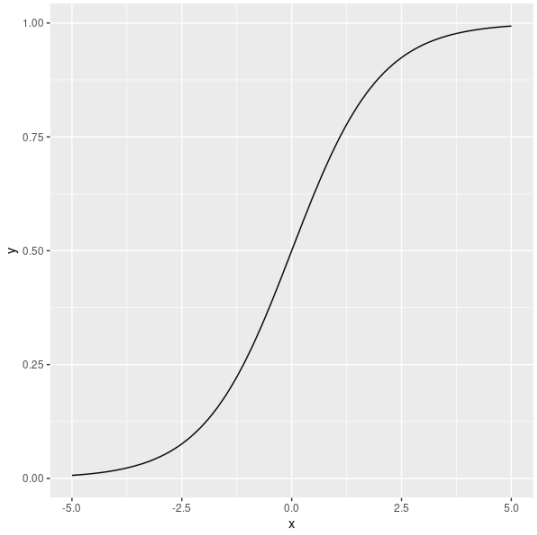


Figure 2.4 Sigmoid function.

Tanh function is like logistic sigmoid but has better properties. It is defined as:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}. \tag{2.5}$$

The range of the tanh function is from -1 to 1. When the outputs of the neural network are desired to be both positive and negative values, tanh function is a better choice compared to the sigmoid function. The graph of the tanh function is shown in Figure 2.5.

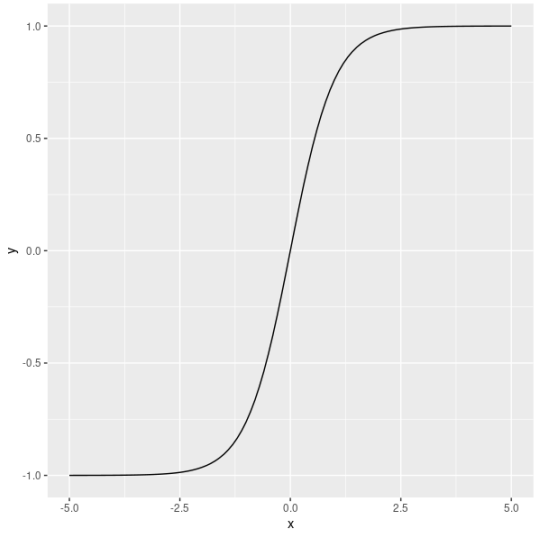


Figure 2.5 Tanh function.

Rectified Linear Unit (ReLU) activation function was first introduced to a dynamical network by Hahnloser et al. in 2000 [Hah00; Hah01]. In 2011, Glorot et al. demonstrated for the fist time that, compared to the widely used activation functions such as logistic sigmoid, ReLU has a better performance when training deeper networks [Glo11]. It is defined as:

$$f(x) = \max(0, x). \tag{2.6}$$

As shown in Figure 2.6, the ReLU is half rectified. $f(x)$ is 0 when x is negative and $f(x)$ is equal to x when x is above or equals to 0. The output range of ReLU function is zero to infinity. ReLU function is usually used in convolutional neural networks. RuLU function is, as of 2017, the most popular activation function for deep neural networks [Ram17]. There are several different forms of ReLU function, for example, leaky ReLU, parametric ReLU, etc.

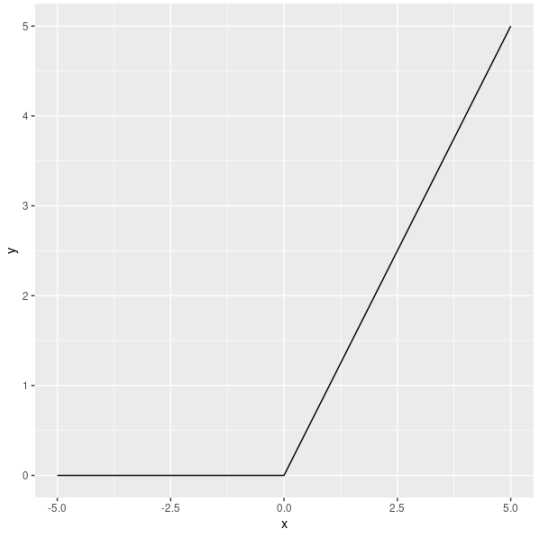


Figure 2.6 ReLU function.

2.2.4 Backpropagation algorithms

Once the structures of a neural network and the loss function are set up, we need to use training data set to train the neural network. This training process is usually called optimization. That is, we find the weights $\mathbf{w}$ of a neural network to minimize the cost function $L(\mathbf{w})$. The most commonly used algorithm is backpropagation algorithm. The backpropagation algorithm contains two main phases: forward and backward phases. In forward phase, the

inputs are fed into the neural network. After a series of computations across the layers by using the current weights, we arrive at the end predicted outputs. Also, the derivative of the loss function with respect to the output is computed. In backward phase, the gradients of loss function with respect to the weight in all layers need to be computed by using the chain rule of differential calculus. The weights of the neural network are updated by using these gradients. For example, in the steepest descent method, the weights are updated by:

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \eta \frac{\partial L(\mathbf{w}_{\text{old}})}{\partial \mathbf{w}}, \tag{2.7}$$

where $\eta \in [0,1]$ is the learning rate.

We will use a simple example to show how backpropagation algorithm works. Figure 2.7 is a simple neural network with one input layer, one hidden layer and one output layer. Each layer contains two neurons. In this network, $\mathbf{x} = (x_1, x_2)$ are the inputs; $w_1, \ldots, w_8$ and $b_1, \ldots, b_4$ are the weights and biases, respectively; h_1, h_2 are the active functions in the hidden layer; H_1, H_2 are the outputs of the hidden layer; O_1, O_2 are the activation functions in the output layer; $\mathbf{y} = (y_1, y_2)$ are the final output values. We define the activation functions h_1, h_2 and O_1, O_2 in the hidden and output layers as the sigmoid functions. The loss function is the mean square error, given by

$$L(\mathbf{w}) = \frac{1}{2}(y_{1,\text{true}} - y_1)^2 + \frac{1}{2}(y_{2,\text{true}} - y_2)^2. \tag{2.8}$$

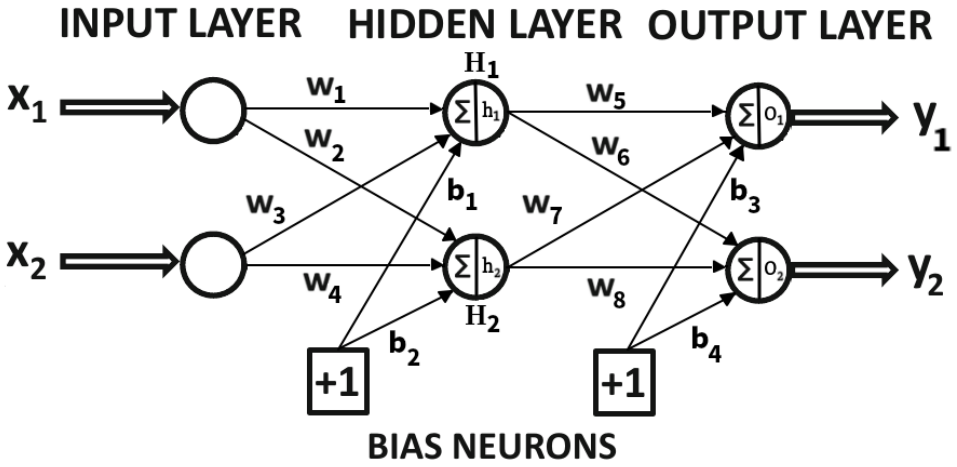


Figure 2.7 The example of a simple neural network.

In the forward phase, by using the current weights, H_1, H_2 and y_1, y_2 are computed:

$$H_1 = h_1(x_1 w_1 + x_2 w_3 + b_1) \tag{2.9}$$

$$H_2 = h_2(x_1 w_2 + x_2 w_4 + b_2) \tag{2.10}$$

$$y_1 = O_1(H_1 w_5 + H_2 w_7 + b_3) \tag{2.11}$$

$$y_2 = O_2(H_1 w_6 + H_2 w_8 + b_4). \tag{2.12}$$

In the backward phase, we need to compute the gradients of the loss function with respect to all weights in each layer. Then, we will use those gradients to update the weights. We will show how to update the weights w_1 and w_5.

The gradient of the loss function with respect to w_5 is:

$$\frac{\partial L(\mathbf{w})}{\partial w_5} = \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial \sum}\frac{\partial \sum}{\partial w_5} \quad \left(\sum = H_1 w_5 + H_2 w_7 + b_3\right), \tag{2.13}$$

where

$$\frac{\partial L}{\partial y_1} = \frac{1}{2}\cdot 2(y_{1,\text{true}} - y_1)\frac{\partial (y_{1,\text{true}} - y_1)}{\partial y_1} = y_1 - y_{1,\text{true}}$$

$$\begin{aligned}\frac{\partial y_1}{\partial \sum} = \frac{\partial O_1}{\partial \sum} &= \frac{\partial}{\partial \sum}\left(\frac{1}{1+e^{-\Sigma}}\right)\\ &= \frac{1}{1+e^{-\Sigma}}\left(1 - \frac{1}{1+e^{-\Sigma}}\right)\\ &= y_1(1-y_1)\end{aligned}$$

and

$$\frac{\partial \sum}{\partial w_5} = H_1.$$

Hence,

$$\frac{\partial L(\mathbf{w})}{\partial w_5} = (y_1 - y_{1,\text{true}})y_1(1-y_1)H_1. \tag{2.14}$$

Thus, the weight w_5 can be update as

$$w_{5,\text{new}} = w_{5,\text{old}} - \eta\frac{\partial L(\mathbf{w}_{\text{old}})}{\partial w_5}. \tag{2.15}$$

The gradient of the loss function with respect to w_1 is:

$$\frac{\partial L(\mathbf{w})}{\partial w_1} = \frac{\partial L}{\partial H_1}\frac{\partial H_1}{\partial \sum}\frac{\partial \sum}{\partial w_1} \quad \left(\sum = x_1 w_1 + x_2 w_3 + b_1\right), \tag{2.16}$$

where

$$\begin{aligned}\frac{\partial L}{\partial H_1} &= \frac{1}{2}\cdot 2(y_{1,\text{true}} - y_1)\frac{\partial (y_{1,\text{true}} - y_1)}{\partial H_1} + \frac{1}{2}\cdot 2(y_{2,\text{true}} - y_2)\frac{\partial (y_{2,\text{true}} - y_2)}{\partial H_1} \\ &= (y_1 - y_{1,\text{true}})\frac{\partial y_1}{\partial H_1} + (y_2 - y_{2,\text{true}})\frac{\partial y_2}{\partial H_1} \\ &= (y_1 - y_{1,\text{true}})y_1(1-y_1)w_5 + (y_2 - y_{2,\text{true}})y_2(1-y_2)w_6\end{aligned}$$

and

$$\frac{\partial H_1}{\partial \sum} = H_1(1-H_1)$$

and

$$\frac{\partial \sum}{\partial w_1} = x_1.$$

Hence

$$\frac{\partial L(\mathbf{w})}{\partial w_1} = [(y_1 - y_{1,\text{true}})y_1(1-y_1)w_5 + (y_2 - y_{2,\text{true}})y_2(1-y_2)w_6]H_1(1-H_1)x_1. \tag{2.17}$$

Thus, the weight w_1 can be update as

$$w_{1,\text{new}} = w_{1,\text{old}} - \eta\frac{\partial L(\mathbf{w}_{\text{old}})}{\partial w_1}. \tag{2.18}$$

The above computations can be repeated to update other weights in the neural network.

In the example above, we only consider one sample. In practice, we usually have n samples. The loss function is then changed to:

$$L(\mathbf{w}) = \sum_{i=1}^{n}\sum_{j=1}^{2}\frac{1}{2}(y_{ij,\text{true}} - y_{ij})^2. \tag{2.19}$$

If the optimization algorithms use the entire training set simultaneously to update the weights, those methods are called deterministic gradient methods. If the optimization algorithms use only a single training example at a time, those methods are called stochastic

(online) methods. If the algorithms use more than one but less than all of the training examples, those methods are traditionally called minibatch stochastic methods. Minibatch stochastic methods are commonly used in deep learning.

The learning rate is another crucial parameter in neural networks. In practice, instead of using fixed learning rate, it is necessary to gradually decrease the learning rate over time. There are many optimization algorithms used in practice, such as AdaGrad [Duc11], RMSProp [Hin12], Adam [Kin14] and SGD (with momentum).

2.3 Convolutional Neural Networks

In deep learning, a deep neural network (DNN) is an artificial neural network with multiple layers between the input and output layers [Sch15]. A convolutional neural network (CNN) is a class of deep neural networks that uses convolution operation. It has many applications in image classification and segmentation, medical image analysis, natural language processing and time series prediction, etc.

CNNs are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers [Goo16]. There are many different kind of layers contained in CNNs. A convolutional network includes an input layer, many hidden layers and an output layer. The hidden layers usually consist of a series of different types of layers. The most common layers used in hidden layers are convolutional layer, pooling layer and fully connected layer. CNNs contain at least one convolutional layer. The output layer usually consists of several neural nodes with an activation function.

2.3.1 Convolutional layer

Convolutional layer is the most important layer in CNNs. A convolutional layer is the multilayer perceptron that uses the convolution operation. In convolutional neural networks, convolutional layers are no longer fully connected to each other, but locally connected. Images have the property of local region stability, and the statistical features of one of their local regions are similarly relative to other adjacent local regions of the image. Therefore, the features of a certain local region learned by convolutional layers from an image are also suitable for other adjacent local regions of the image. The convolutional layer has many filters that can be designed to recognize certain specific features of the image, each filter slides over the feature map of the previous layer of convolution, and the parameters of filters are invariant and shared during the convolution process. This allows the convolutional neural network to be trained with very few parameters compared to the previous fully

connected neural networks that required a large number of parameters when training large scale input samples.

2.3.1.1 Convolution operation

The convolution operation is the basic operation in the convolutional layer. A convolutional layer consists of a set of filters. The parameters contained in the filters need to be learned from the training set. Usually, the size of filters is smaller than the input data.

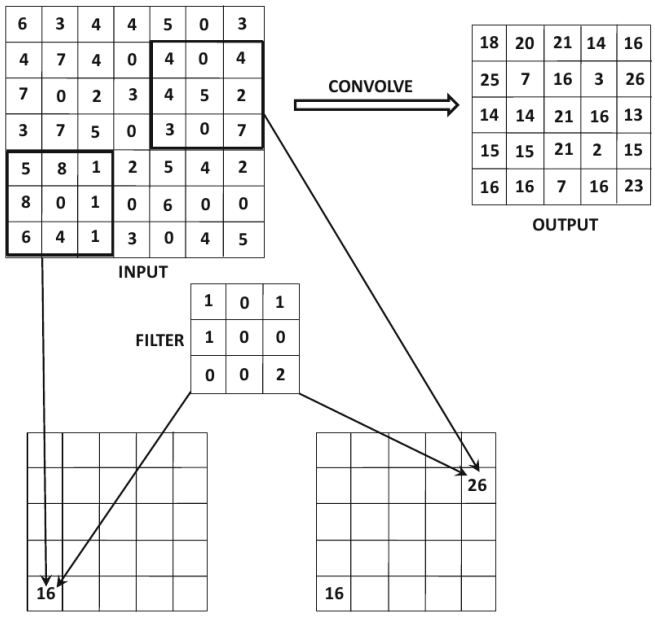


Figure 2.8 An example of convolution operation between a $7 \times 7 \times 1$ input and a $3 \times 3 \times 1$ filter with stride of 1 [Agg18].

Assuming, we have an input $\mathbf{x}$, a filter with weight $\mathbf{w}$, then the convolution operation is defined as:

$$f(\mathbf{x}) = \mathbf{x} * \mathbf{w}. \tag{2.20}$$

The convolution operation start with a filter which is a matrix of weights. The filter slides over the input data, performing an element wise multiplication with the part of the input it is currently on and then summing up the results into a single output pixel. Figure 2.8 shows an example of a convolution operation between a $7 \times 7 \times 1$ input and a $3 \times 3 \times 1$ filter with stride of 1 [Agg18].

The stride is defined as the number of location that a filter skipped when the filter

is sliding over the input data. Usually, in convolution networks, we want the output of a convolutional layer to have a low dimension when comparing with the input. So the bigger the stride is, the smaller the dimension that an output will have. If the stride is so large, we may lose some information containing in the input.

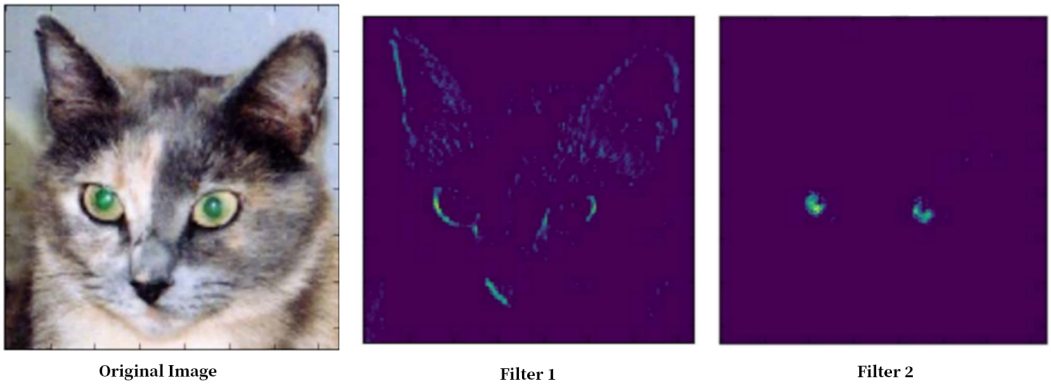


Figure 2.9 An example of applying convolution operation on a real image [Fra17].

Figure 2.9 shows an example of applying the convolution operation on a real image [Fra17]. The original image is a cat. After applying the convolution operation with filter 1, we can get the edge of the cat. This means that the filter 1 is used to grab the edge of the object. After applying the convolution operation with filter 2, we get a feature map with the eye of the cat. This means that the filter 2 is used to grab the bright green features from the original image. The convolution operation can learn spatial hierarchies of patterns. With more convolution layers, the neural network can learn much more complex and abstract visual concepts.

2.3.1.2 Padding

Usually, the output of an convolution layer has a smaller dimension compared with the input. But if we do not want to lose the dimensionality of the output, i.e., the dimension of the output stays the same with the dimension of the input, the padding method can help us with this. What the padding method will do is padding the edges with extra fake pixels. The extra fake pixels are the pixels with value 0. Figure 2.10 shows an example of padding [Agg18].

After padding, the original edge pixels becomes the pixels at the center. So it can produce an output with the same dimension as the input.

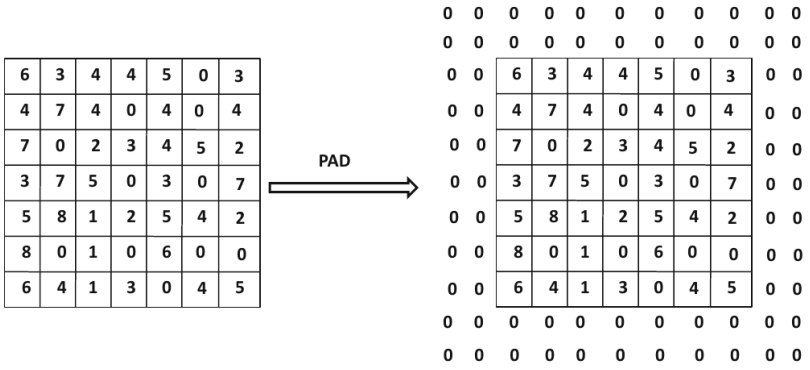


Figure 2.10 An example of padding [Agg18].

2.3.2 Pooling layer

Pooling layer is another important layer in CNNs. The pooling layer uses the pooling operation to reduce the dimension of the output. The pooling operation works on small square grid regions with dimension $P \times P$ in each layer. The output of the pooling layer will have a smaller size and the same depth as the input. There are two kinds of pooling operations. One is average pooling. Average pooling calculates the average value for each small square grid region in the input data. Another one is maximum pooling. Maximum pooling calculates the maximum value for each small square grid region in the input data. The stride in the pooling layer is the same as in the convolutional layer.

Figure 2.11 shows an example of maximum pooling with a 3× 3 filter and stride equals to 1 and 2. With the stride of pooling increases, the size of the output is becoming smaller. Pooling layer is a form of non-linear down-sampling. Pooling layer can help us to extract the important features from the original image. It is useful in image segmentation neural networks.

2.3.3 Fully connected layer

Fully connected layer is similar with the MLP. In a fully connected layer, every neuron has full connections to all activations in the previous layer. Fully connected layer can help to collect the useful classification information contained in the convolutional layer or pooling layers. In order to improve CNN network performance, ReLU activation function is usually used in the fully connected layer. In the convolutional neural network, fully connected layer commonly used at the end of the network before a classification layer.

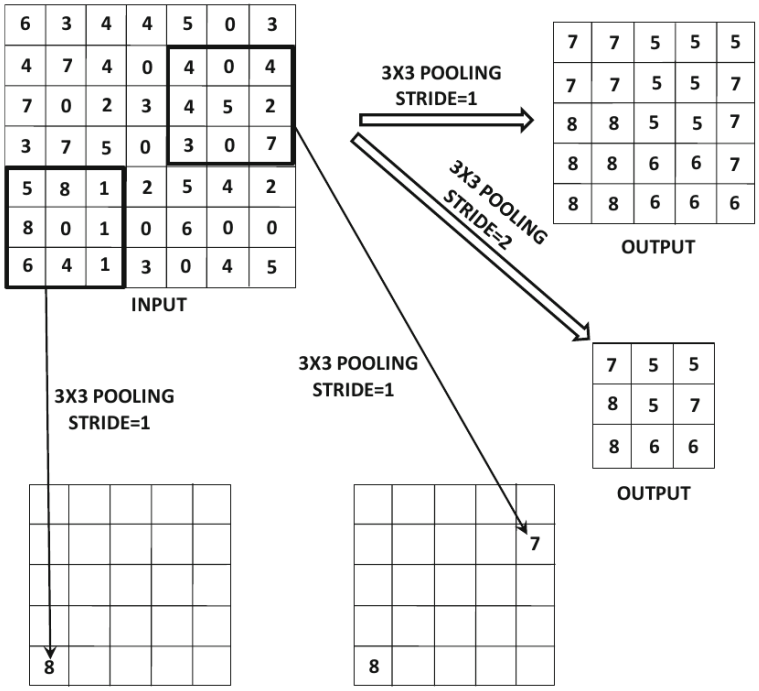


Figure 2.11 An example of maximum pooling with a 3× 3 filter, and stride equals to 1 and 2 [Agg18].

2.3.4 Dropout

Overfitting is a serious problem in convolutional neural network, especially for large scale networks. Dropout is a technique proposed by Nitish Srivastava et al. in 2014. A hidden layer with dropout means that some neurons will randomly drop out in this hidden layer. Dropping out a neuron means that a neuron will be temporarily removing it from the network, along with all its incoming and outgoing connections [Sri14].

Dropout method can be used in most types of layers, such as dense fully connected layers, convolutional layers and recurrent layers, etc. Figure 2.12 shows how dropout method works on a fully connected layer [Sri14]. In the dropout method, we can set up the dropout rate for different layers or problems. The dropout rate ranges from 0 to 1. In the simplest case, each unit is retained with a fixed probability (dropout rate) independent of other units. Usually, a good value for dropout in a hidden layer is between 0.5 and 0.8 [Sri14].

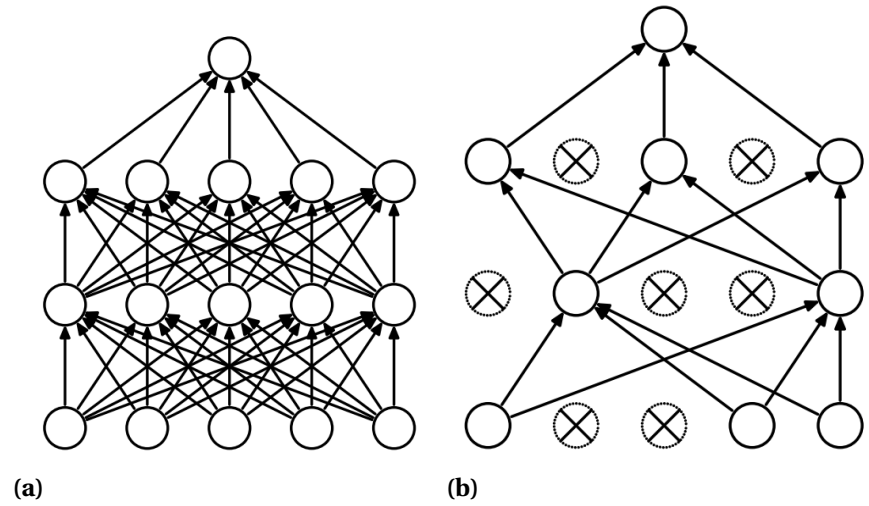


Figure 2.12 Dropout Neural Net Model:(a) is the standard neural net; (b) is the neural net after applying dropout [Sri14].

2.3.5 Loss function

2.3.5.1 Cross entropy

Cross entropy (CE) is a measure of dissimilarity between two distributions. In machine learning task, we usually use cross entropy for classification and segmentation problems. The cross entropy is defined as:

$$L_{CE} = -\sum_{i=1} p_i \log(\hat{p}_i), \tag{2.21}$$

where p_i is the true class and $\hat{p}_i$ is the probability that the prediction belong to the true class. The binary form is defined as:

$$L_{BCE} = -(p \log(\hat{p}) - (1-p)\log(1-\hat{p})), \tag{2.22}$$

where p is the true class and $\hat{p}$ is the probability that the prediction belong to the true class.

2.3.5.2 Focal loss

Focal loss adapts the standard CE to deal with extreme foreground-background class imbalance, where the loss assigned to well-classified examples is reduced [Lin17]. It is defined as:

$$L_{FL} = \left(\alpha(1-\hat{p})^{\gamma} p \log(\hat{p}) + (1-\alpha)\hat{p}^{\gamma}(1-p)\log(1-\hat{p})\right), \tag{2.23}$$

where $\alpha \in [0,1]$ is the weighting factor and $\gamma \geq 0$ is the tunable focusing parameter.

2.3.5.3 Dice loss

Dice loss was proposed by Fausto Milletari, et al. in 2016 [Mil16]. It can directly optimize the Dice coefficient which is the most commonly used segmentation evaluation metric. It is defined as:

$$L_{DC} = 1 - \frac{2TP}{2TP + FP + FN}, \tag{2.24}$$

where TP is the true positive, FP is the false positive and FN is the false negative.

2.3.6 Evaluation metrics

Evaluation metrics are used to measure the quality of the machine learning model.

2.3.6.1 Intersection over Union

Intersection over Union (IoU) is an evaluation metric used to measure the accuracy of an object detector on a particular dataset. IoU is usually used to measure the performance of any object category segmentation method [Rah16]. It is defined as

$$L_{IoU} = \frac{TP}{TP + FP + FN}. \tag{2.25}$$

2.3.6.2 Dice coefficient

Dice coefficient is a statistical metric used to gauge the similarity of two samples. Dice coefficient is widely used in medical image segmentation applications. It is defined as:

$$DC = \frac{2TP}{2TP + FP + FN}. \tag{2.26}$$

2.4 U-Net and UNet++

2.4.1 U-Net

U-Net is one of the state of art deep learning methods designed for biomedical image segmentation. It is widely used in all kinds of biomedical image segmentation problems. There are many new methods that are based on the idea of U-Net. U-Net was proposed by Olaf Ronneberger et al. in 2015. U-Net is based on a classical idea, that is, encoder-decoder [Hin06]. This idea was firstly used to compress images and reduce the noise.

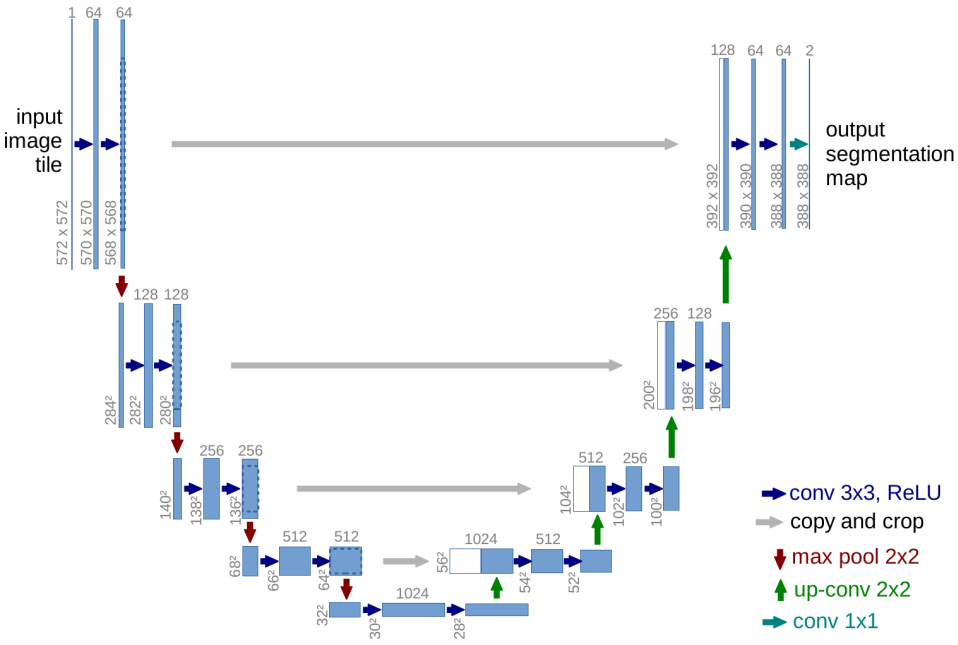


Figure 2.13 The structure of U-Net [Ron15].

Figure 2.13 shows the structure of U-Net [Ron15]. U-Net has a symmetric structure. The left part is the contracting part (encoder), which is used to extract the context information of the object from the original image. The contracting part consist of ten 3×3 convolutional layers with ReLU as the activation function and four 2×2 maximum pooling layers with stride 2 for downsampling. The right part is the expanding part (decoder), which is used to localize the position of the object. The expanding part contains seven 3×3 convolutional layers with ReLU as the activation function and five up-convolutional layers in which half the number of feature channels concatenates with the correspondingly cropped feature map from the contracting part. The last layer is a 1×1 convolution layer with stride 1 and unpadding. The last two channels represent the background class and object class, respectively. U-Net totally has 23 convolution layers and 4 maximum pooling layers. U-Net is a well designed deep learning architecture.

2.4.2 UNet++

UNet++ is a new powerful architecture for biomedical image segmentation. UNet++ is based on the structure of U-Net. The main idea behind UNet++ is to bridge the semantic gap between the feature maps of the encoder and decoder prior to fusion [Zho18]. Compared with U-Net, UNet++ added two sub-networks and re-designed skip pathways. Also, UNet++ uses the deeply supervised method to help updating the sub-networks.

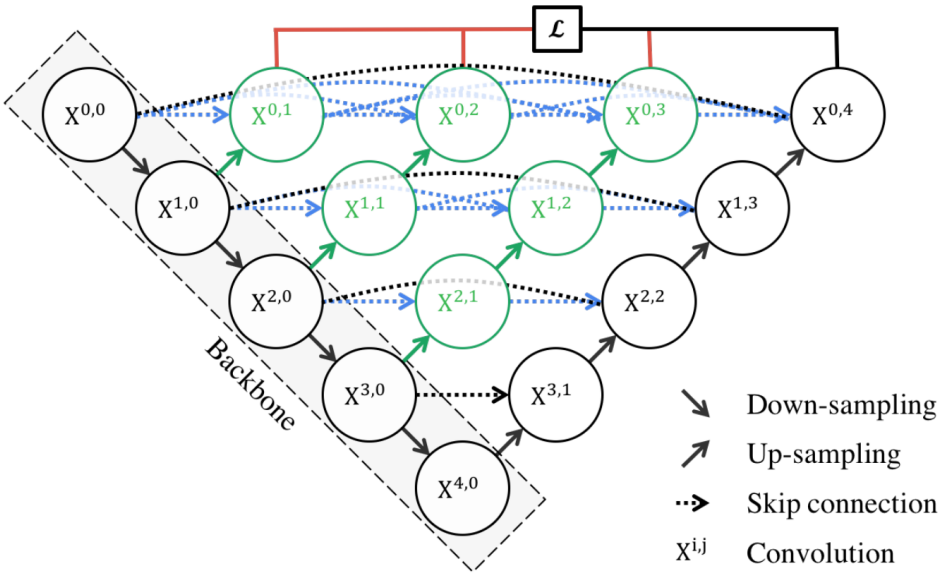


Figure 2.14 The structure of UNet++ [Zho18].

The structure of UNet++ is shown in Figure 2.14 [Zho18]. The green circles are convolutional blocks of the sub-networks. The blue dash lines are the re-designed skip connections. The red lines indicate the usage of deep supervision. All feature maps extracted by $X^{1,0}, X^{2,0}, X^{3,0}$ and $X^{4,0}$ blocks are very important. Those features contain not only shallow and low-level feature maps, but also fine-grained feature maps. So by adding the sub-networks and skip connections, we can fully use those extract features maps. The skip connections can effectively generate the segmentation masks with fine details and recover the fine-grained details of the target objects even with complex background. The role of deep supervision here is to allow the sub-network trainable. By monitoring the results of sub-networks, all the convolutional blocks in UNet++ can be trained. After training, we can compare the results of each sub-network. There are 3 sub-networks here: the network with blocks $X^{0,0}, X^{1,0}, X^{0,1}$, the network with blocks $X^{0,0}, X^{1,0}, X^{2,0}, X^{1,1}, X^{0,2}$ and the network with blocks $X^{0,0}, X^{1,0}, X^{2,0}, X^{3,0}, X^{2,1}, X^{1,2}, X^{0,3}$. If a small sub-network has the similar performance compared with the whole networks. We can do the pruning on UNet++. We can only use the small sub-network to do the prediction.

The backbone architecture refers to the way in which the backbone interconnects the networks attaching to it and how it manages the way in which packets from one network move through the backbone to other networks. The backbone of UNet++ is not fixed. There are many backbones we can choose from, such as VGG networks [Sim14], ResNet networks [He16], Inception V3 [Sze16] and DenseNet networks [Hua17], etc. Those models all have

trained weights on Imagenet dataset [Den09]. We can choose the different backbone for different segmentation applications.

2.5 Evolutionary Genetics

Phylogenetic trees are diagrams that represent hypotheses about the evolutionary relationships among organisms. Phylogenetic tree is built based on the information we've collected about the set of species, things like the DNA(RNA) sequences of their genes and their physical features.

For virus, there are two different types of populations: exponential growth virus population and constant growth virus population. In exponential growth population, the growth rate of the population stays the same regardless of population size, making the population grows faster as it gets larger in size. In constant growth population, this population grows by a constant amount for each unit of time, making the population grows linearly. Figure 2.15 shows the idealized caricatures of virus phylogenies that distinguish between virus population with (A) exponential growth; (B) constant size [Vol13]. In Figure 2.15 (A), the virus grows exponentially and the phylogenetic tree has long terminal branches (the long lines to the tip of the tree). In Figure 2.15 (B), the virus population is constant and the phylogenetic tree has short terminal branches, but long internal branches. Finding the exponential growth virus is meaningful in viral phylodynamics. However, it is very difficult to find the exponential growth samples in a very big phylogenetic tree. Here, we want to use deep learning method to find out the exponential growth samples contained in the viral sequence data.

Assuming mutation occurs along the phylogenetic tree at a constant rate. We will build a pairwise difference matrix for the viral sequences. Each entry in the matrix denotes the distance between the two corresponding sequences, i.e., how many different genes between two corresponding species.

We will give an example showing the differences in the pairwise difference matrices between the constant growing population and the exponential growing population. Figure 2.16 shows the two phylogenetic trees with mutation marked on it, each "X" denotes a mutation along the tree. For both groups, there are 8 sequences of virus sampled. The $x_1, \ldots, x_8$ are the exponential growth samples and the $y_1, \ldots, y_8$ are the constant growth samples. Each branch point represents the most recent common ancestor of all the groups descended from that branch point. In order to find the number of different genes between two species, we need to find their most recent common ancestor first. For example, one of the red points in Figure 2.16 is the most recent common ancestor of x_1 and x_2. We start from

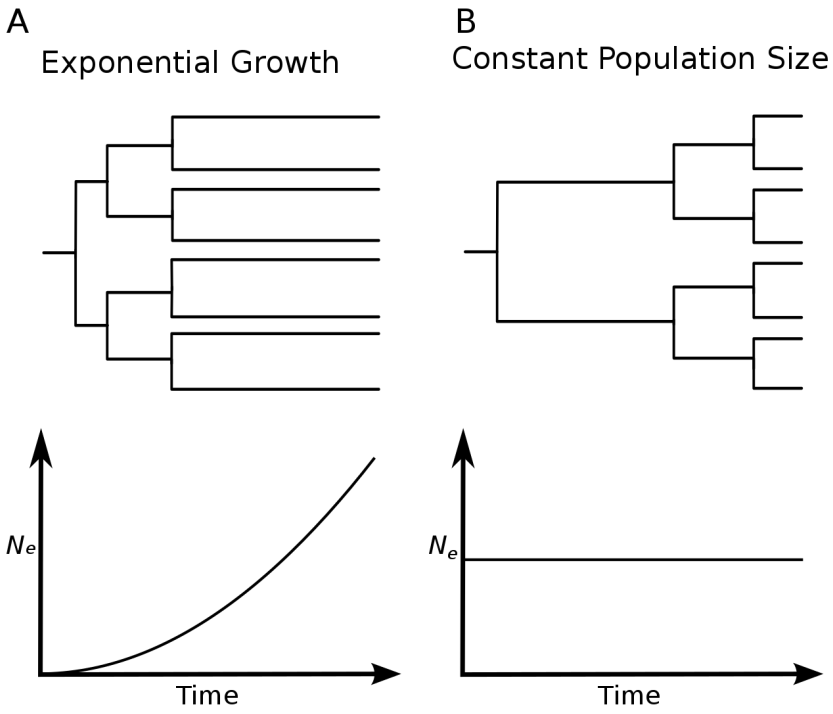


Figure 2.15 Idealized phylogenetic trees of the (A) exponential growth virus population and (B) constant growth virus population and the corresponding population size [Vol13].

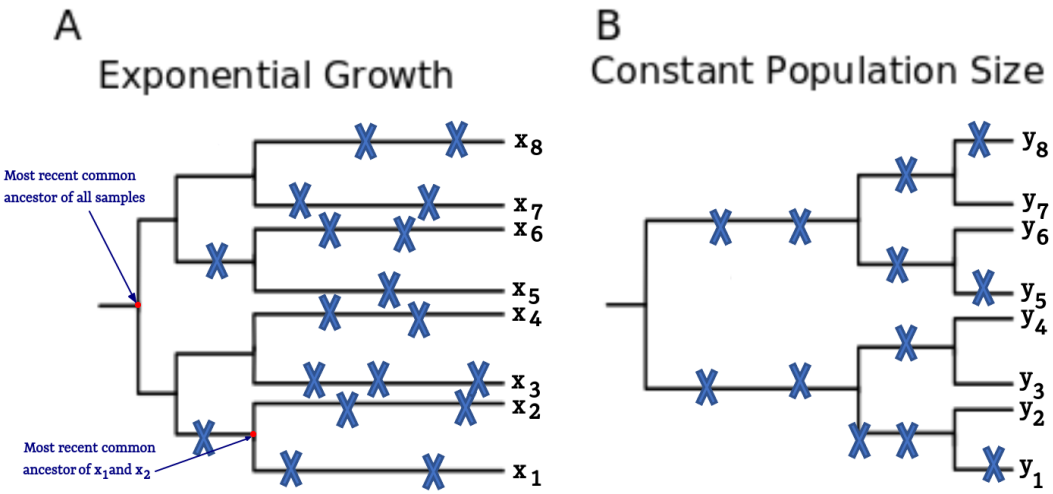


Figure 2.16 The phylogenetic trees with mutations of (A) exponential growth virus population and (B) constant growth virus population, respectively.

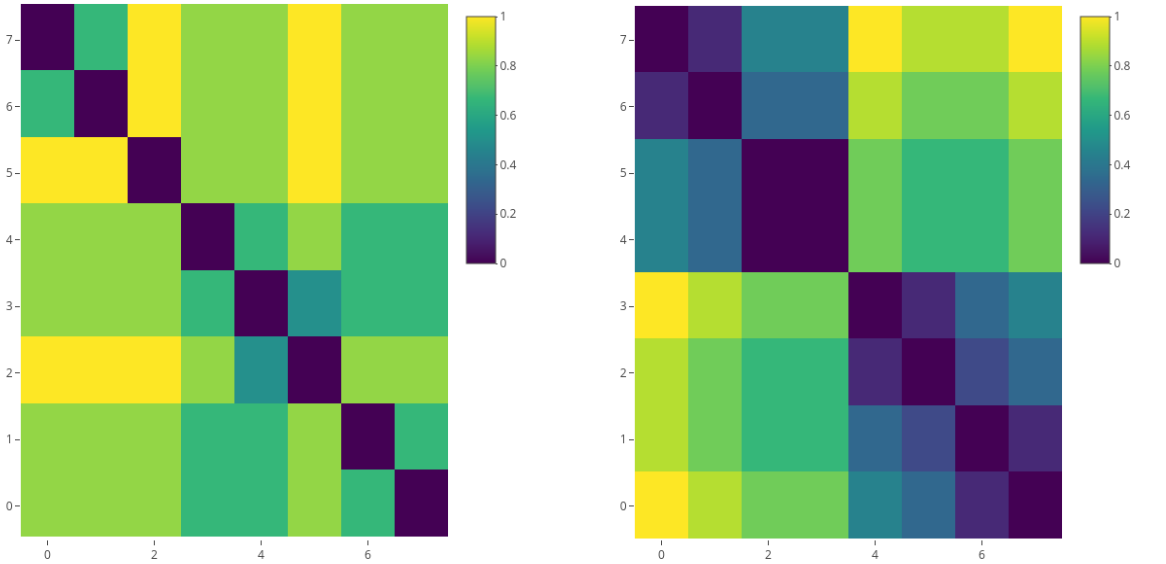
x_1 go back to the most recent common ancestor, and then go to x_2 from the most recent common ancestor. During this process, we count how many mutations(i.e., X) showing in the path. The number of mutations in the path is the number of different genes between two species. So, there is 4 different genes between x_1 and x_2.

After counting all the difference genes between species, we can build two 8×8 matrices. The pairwise difference matrices for them are (A is the matrix for group A, B is the matrix for group B):

$$A = \begin{array}{c} \begin{matrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \end{matrix} \\ \begin{pmatrix} 0 & 4 & 6 & 5 & 5 & 6 & 5 & 5 \\ 4 & 0 & 6 & 5 & 5 & 6 & 5 & 5 \\ 6 & 6 & 0 & 5 & 5 & 6 & 5 & 5 \\ 5 & 5 & 5 & 0 & 4 & 5 & 4 & 4 \\ 5 & 5 & 5 & 4 & 0 & 3 & 4 & 4 \\ 6 & 6 & 6 & 5 & 3 & 0 & 5 & 5 \\ 5 & 5 & 5 & 4 & 4 & 5 & 0 & 4 \\ 5 & 5 & 5 & 4 & 4 & 5 & 4 & 0 \end{pmatrix} \end{array} \begin{matrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{matrix} \qquad B = \begin{array}{c} \begin{matrix} y_1 & y_2 & y_3 & y_4 & y_5 & y_6 & y_7 & y_8 \end{matrix} \\ \begin{pmatrix} 0 & 1 & 4 & 4 & 9 & 8 & 8 & 9 \\ 1 & 0 & 3 & 3 & 8 & 7 & 7 & 8 \\ 4 & 3 & 0 & 0 & 7 & 6 & 6 & 7 \\ 4 & 3 & 0 & 0 & 7 & 6 & 6 & 7 \\ 9 & 8 & 7 & 7 & 0 & 1 & 3 & 4 \\ 8 & 7 & 6 & 6 & 1 & 0 & 2 & 3 \\ 8 & 7 & 6 & 6 & 3 & 2 & 0 & 1 \\ 9 & 8 & 7 & 7 & 4 & 3 & 1 & 0 \end{pmatrix} \end{array} \begin{matrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{matrix}$$

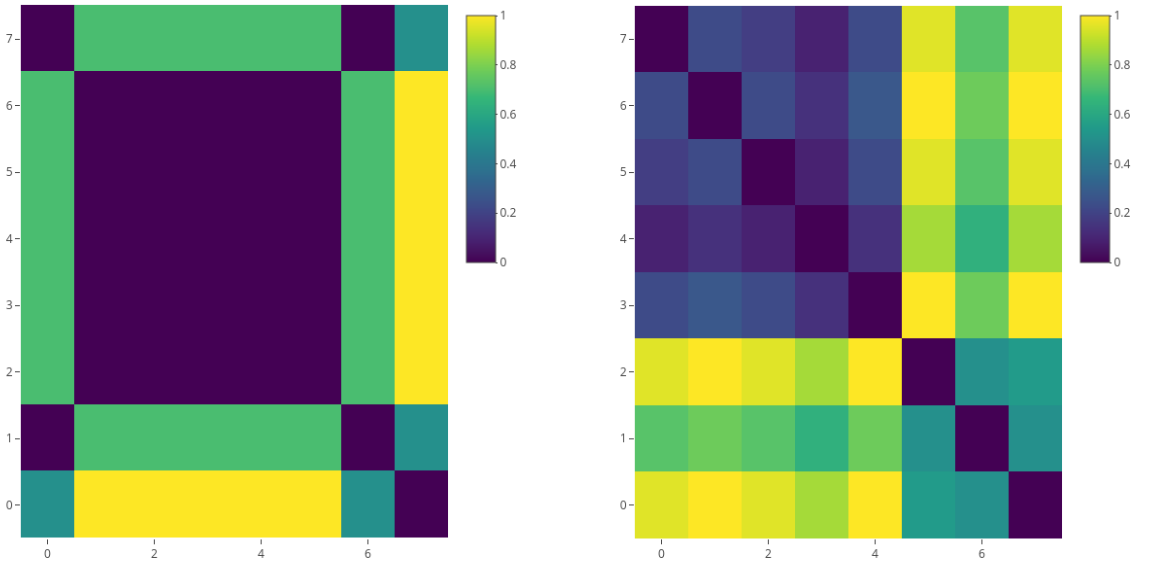
Except for diagonal elements, the values in matrix B range from 0 to 9, but for matrix A, the range is from 3 to 6. The values of entries in matrix A are much more closer to each other. We can clearly see the difference between two matrices. To make it more clear, we will normalize these two matrices by dividing each element in the matrix with the maximal value, and plot the heat map for the normalized matrices A and B. Figure 2.17 clearly shows the difference in the pairwise matrices. In Figure 2.17 (a), the colors are light in all locations except on the diagonal. However, in Figure 2.17 (b), there are many darker blocks around the diagonal, especially in the upper left and lower right corners. Hence, the demographic history can lead to differences in the heat map of pairwise difference matrices.

What if there is an exponential growth group containing in a set of viral genetic sequences? Figure 2.18 shows the 5 exponential growth virus sequences contained in the 8 sampled virus sequences. Compared with the maximal number of mutations occurred between the constant growth virus, the maximal number of mutations occurred between the exponential growth virus is relatively small. So, after we normalized the pairwise difference matrix, the values represented the exponential growth virus are closer to 0. Hence, in the heat map, the exponential growth virus will have more dark blocks around the diagonal. We can clearly recognize the exponential growth virus sequences in Figure 2.18. The virus



(a) **(b)**

Figure 2.17 The heat maps for the normalized matrices A and B, respectively.



(a) **(b)**

Figure 2.18 The heat maps contain exponential block.

sequences from 1 to 5 in Figure 2.18 (a) and the virus sequences from 0 to 4 in Figure 2.18 (b) are the exponential growth virus sequences.

Being able to detect exponential growth is of paramount importance. We can use this to predict an epidemic, or check a virus adaptation to the drug pressure, i.e., drug resistance, or check the immune escape within a person.

Based on the Wright-Fisher model [Hof17] and applying the Poisson distribution to generate mutations between two generations, we generate the simulated data set for genealogies for viruses. The Wright-Fisher model is the simplest population genetic model. Consider a gene with two alleles, A or B. In diploid populations consisting of N individuals there are 2N copies of each gene. An individual can have two copies of the same allele or two different alleles. We denote the frequency of one allele p and the frequency of the other q. In Wright-Fisher model, we assume that generations do not overlap; each individual from the offspring generation now picks a parent at random from the previous generation and each offspring inherits the genetic information of the parent. The Wright-Fisher model is a discrete-time Markov chain that describes the evolution of the count of one of these alleles over time. Let $X_t \in \{0,1,\ldots,2N\}$ be the count of the A allele in a population with N diploid individuals at generation t. Each generation, a collection of alleles are sampled, with replacement, from the current population at generation t to form a new population at generation $t+1$. This process describes the binomial sampling of alleles in each generation. The probability transition matrix can be written as:

$$P_{ij} = \binom{2N}{j} \left(\frac{i}{2N}\right)^j \left(1-\frac{i}{2N}\right)^{2N-j}, \tag{2.27}$$

where i is an allele count at generation $t-1$, j is the allele count at generation t. So we can obtain an offspring generation from a given parent generation by using the Wright-Fisher model.

Using the Wright-Fisher model, we generated 366000 pairwise difference matrices in total. Each pairwise difference matrices represents a sample of 128 viruses (from the simulated genealogy), associated with 128 corresponding labels (denoting whether the viruses were from a population undergoing exponential population expansion, '0' means 'constant growth virus', '1' means 'exponential growth virus'). Thus the size of heat maps and masks for our data set is 128×128. The size of the exponential blocks contained in the heat map ranges from 5 to 125, and 3000 samples for each size of the exponential blocks. Also, we have 3000 samples which do not contain the exponential blocks. The exponential blocks lay randomly in the diagonal of the heat map. Each heat map contains only one exponential block. To save memory in the training process, we plot the heat map as black

and white. Some examples of the heat map and mask are shown in Figure 2.19. The mask shows the location of the exponential block.

Our goal here is, given a heat map, using the image segmentation method to find the exponential block contained in the heat map and provide a probability for the exponential block class.

2.6 Experiments and Results

The entire simulated data set contains 366000 samples, among them 3000 samples do not contain the exponential blocks. From the 363000 samples which contains exponential block, we will choose 10 samples from each size (range from 5 to 125) of the exponential blocks samples as the validation set. So the validation set totally contains 1210 samples and has all the different size of exponential blocks inside. For the test set, we choose 30 samples from each size of the exponential blocks samples. So we totally have 3630 samples in the test set. The test set also includes all the different size of exponential blocks samples. Notice, there is no overlap between the validation set and the test set. The validation and the test set will be fixed from now on. We will select the training set from the rest of 358160 samples. We define the large exponential block sample as the sample with exponential block size equal or bigger than 20, and the small exponential block sample as the sample with exponential block size smaller than 20.

2.6.1 UNet++ for large exponential block samples

The training set is so large, if we choose all the rest of 358160 samples as the training set. Another reason that we will not choose all the rest samples as training set is we may not have sample from each size of the exponential blocks in the real sequencing data. Also, some sequencing data have thousands virus samples. The size of the exponential block will be ranged from a very small number to thousand. It is unwisely to choose samples from each size of the exponential blocks. So how can we only use the samples from several sizes of the exponential blocks and still can guarantee the accuracy of our results. In order to select the appropriate training set, we will generate 8 different training sets. Here, we define the common difference of the training set. The common difference of the training set is defined as the difference between the size of each exponential blocks in the training set. This means that if the common difference of the training set is 10, then this training set will contain the samples which have the exponential blocks with size 5, 15, 25, 35, 45, 55, 65, 75, 85, 95, 105, 115, 125.

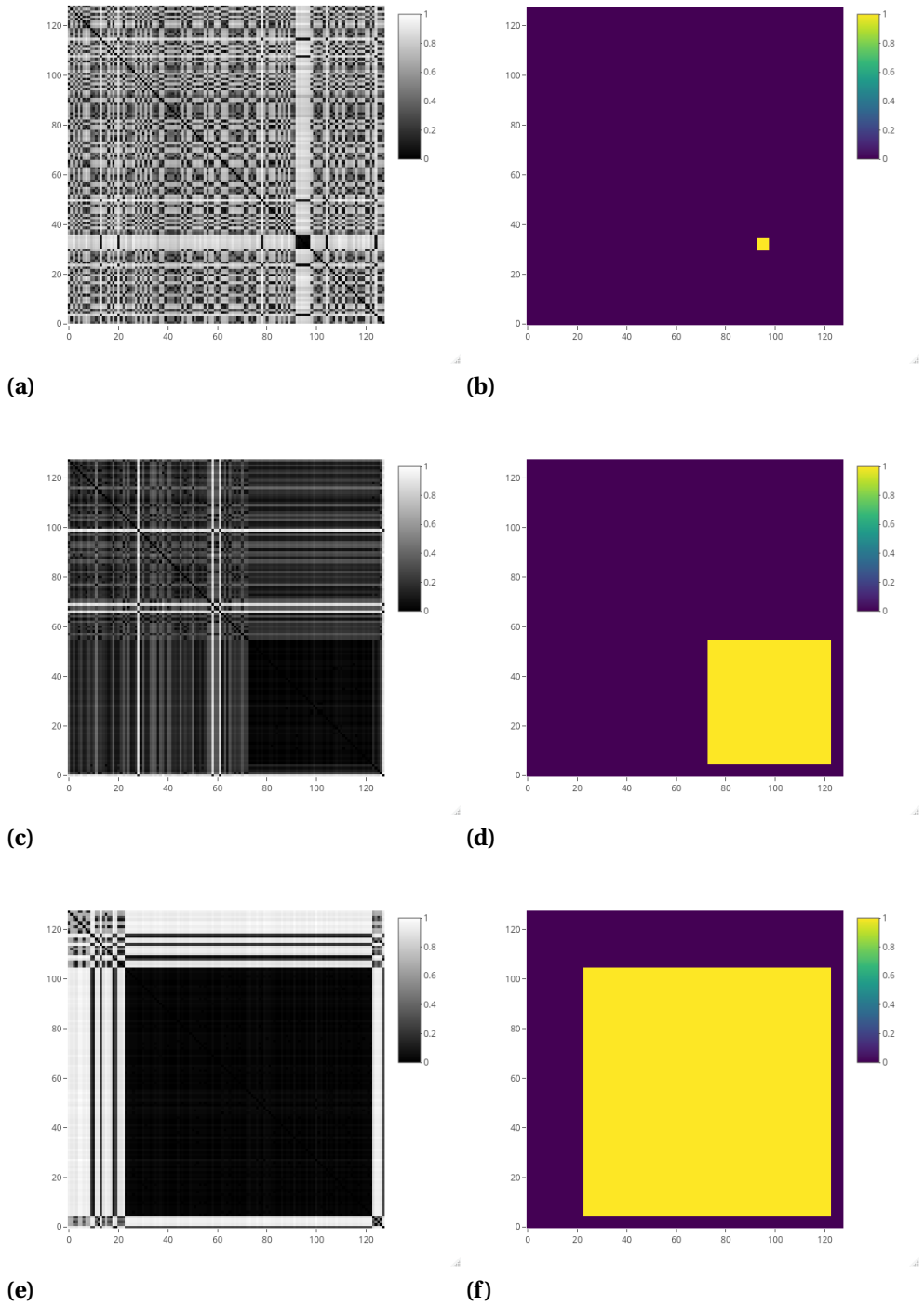


Figure 2.19 Some examples of heat maps in the simulated data set: graph (a) and (b) is the heat map and mask map which contains a exponential black with size 5; (c) and (d) is the heat map and mask map which contains a exponential black with size 50; (e) and (f) is the heat map and mask map which contains a exponential black with size 100.

The common differences of the 8 training sets are 1, 2, 3, 5, 10, 15, 20 and 25 , respectively. For each training set, we have 10000 training samples in total. This means that there are $\lceil \frac{10000}{\#\text{different size}} \rceil$ samples of each size of the exponential blocks in each training set.

We apply the UNet++ algorithm to the 8 training sets (the sets of heat maps). In the UNet++, we choose Inception V3 as our backbone, the weights based on the Imagenet as the encoder weights and the Adam method as the optimizer. We use the combination of binary cross entropy and dice coefficient as the loss function:

$$L = \frac{1}{2} L_{\text{BCE}} - \text{DC}. \tag{2.28}$$

The learning rate for Adam is 0.001. We train each model for 20 epochs and the batch size sets as 32. Here, we use IoU method to evaluate the performance of our models.

Figure 2.20 shows the performance of UNet++ with different training sets. For the large exponential block samples, even with different training sets, the UNet++ has nearly the same performance. Except for the UNet++ based on the training set with common difference 25, the mean IoU accuracy for the large exponential block test samples are all greater than 95%. This means that, for the large exponential block samples, we do not need to use the training set which contains samples from each size of the exponential blocks.

However, we noticed that the performance of UNet++ for the small exponential block test samples dropped dramatically when the training set with common difference reaches 20. And the mean IoU accuracy of large exponential block test samples dropped significantly when the common difference of the training set reaches 25. This implies that the training set with common difference 20 is a good choice for UNet++ for large exponential block samples. But the mean IoU accuracy of the small exponential block samples are very low compared with other samples in the UNet++ based on training set with common difference 20. For some samples with small sizes of the exponential blocks, the IoU accuracies are 0. There are no predictions for those samples. Also, some predictions for those samples are completely wrong. In our problem, it is unacceptable for having no prediction or completely wrong prediction. This will cause clinicians or biologists to make wrong conclusions about the virus proliferation.

To improve the performance of UNet++ for small exponential block samples, the first idea that was tried is to increase the training samples. The training set we used here is the training set with common difference 20. So for small exponential block samples, we only have the samples with exponential block size 5. We do not have samples with exponential block size from 6 to 19 in our training samples. This maybe the reason why UNet++ have bad performance for samples with small exponential block. Another reason is a single

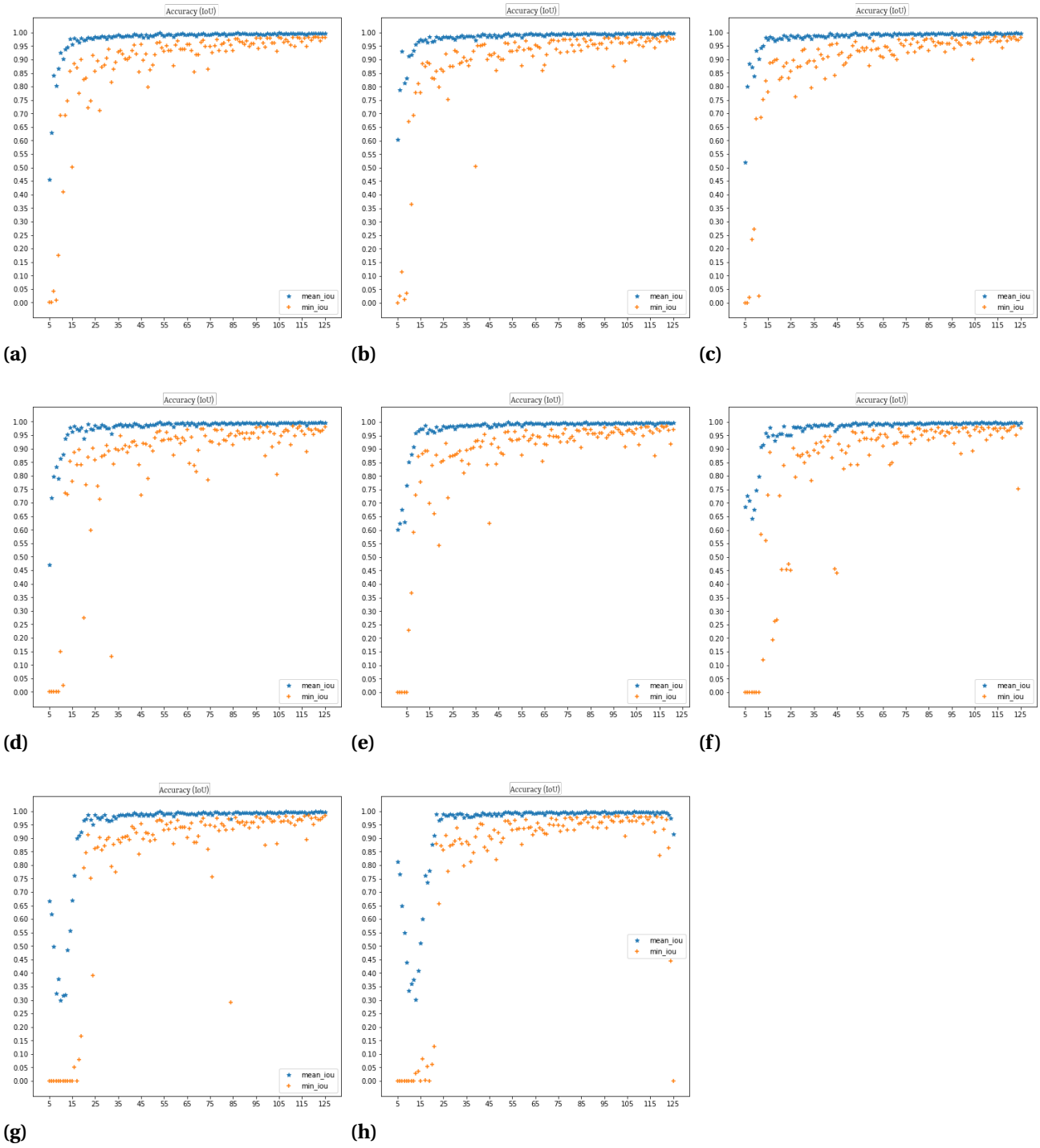


Figure 2.20 The IoU accuracy of the UNet++ of different training sets with common difference 1, 2, 3, 5, 10, 15, 20, 25, respectively.

convolutional neural network can not perform well on both small and big objects at the same time.

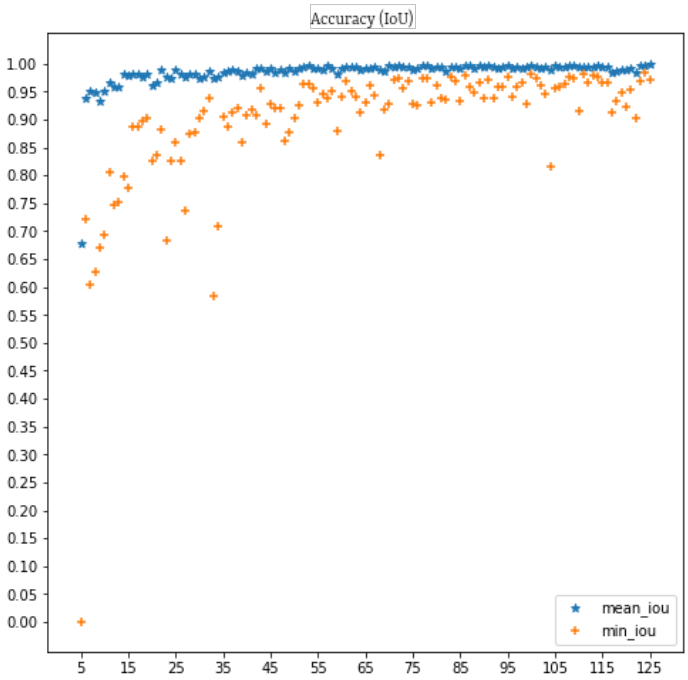
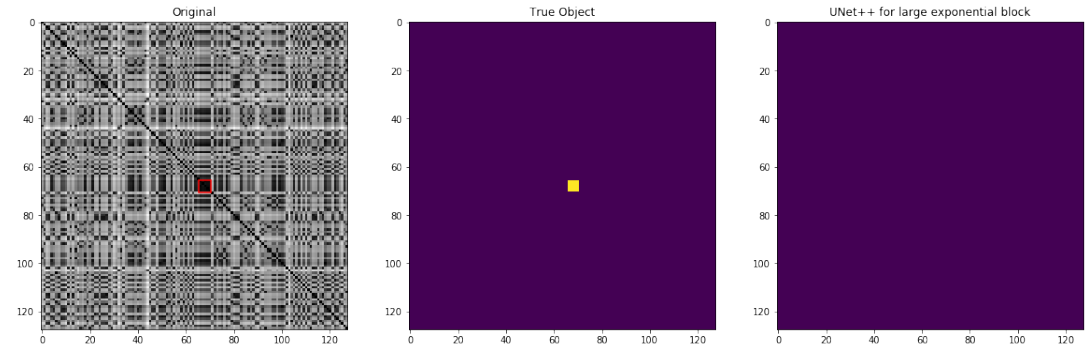


Figure 2.21 The IoU accuracy of the UNet++ for large exponential block samples.

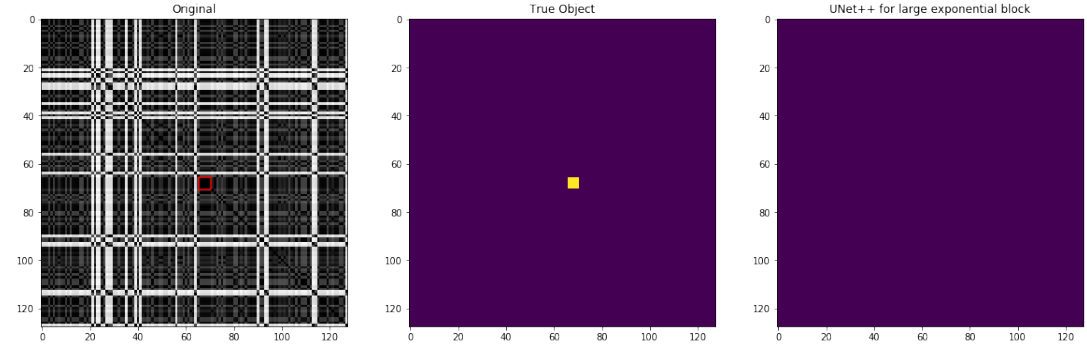
To over come this problem, we add samples which contain small size exponential blocks (size from 6 to 19) into our training set. For each size of the exponential blocks, we add 1000 samples into our training set. Hence, we totally have 23996 samples in our new training set. Also, we found that the SGD optimizer has a better performance than Adam when we used the new training set. Thus, we chose SGD as the optimizer for the new training set. The hyper parameters for SGD optimizer were set as follows: learning rate is 0.01, the decay factor is 10^{-6} and the momentum is 0.9. All other hyper parameters of the UNet++ remain the same as before. We called this UNet++ as the UNet++ for large exponential block samples.

The result of UNet++ for large exponential block samples is shown in Figure 2.21. Compared with UNet++ based on the training set with common difference 20, this one significantly improved the accuracy of small exponential block samples. Except for the samples with size 5 exponential block, the mean IoU accuracy of all other samples are around 95%. However, as shown in Figure 2.21, even if we add more small exponential block sample to the training set, we still have 0 accuracy for some samples with size 5 exponential

block. The average accuracy for all test samples with size 5 exponential block is 67.7%.



(a)



(b)

Figure 2.22 The worst cases for identifying the size 5 exponential blocks in the test samples (Red rectangle are the true exponential block area, the yellow areas are the predicted results).

Figure 2.22 shows the worst cases for identifying the size 5 exponential blocks by using the UNet++ for large exponential block samples. As discussed before, we have zero tolerance for the non-predictive situations. We need to find another way to solve the problem of non-prediction.

2.6.2 UNet++ for small exponential block samples

A simple way to solve the non-predictive issue is to build another UNet++, which only focuses on samples with small exponential blocks. The training set that we used here contains all samples (except the samples in validation and test sets) with exponential block size from 5 to 10. For each size of exponential block, we have 2960 samples. In total, we have 17760 samples.

In the UNet++ for small exponential block samples, we choose Inception V3 as the

backbone, the weights based on the Imagenet data set as the encoder weights, the SGD method as the optimizer and equation (2.28) as the loss function. We set the batch size as 32 and train the model for 20 epochs.

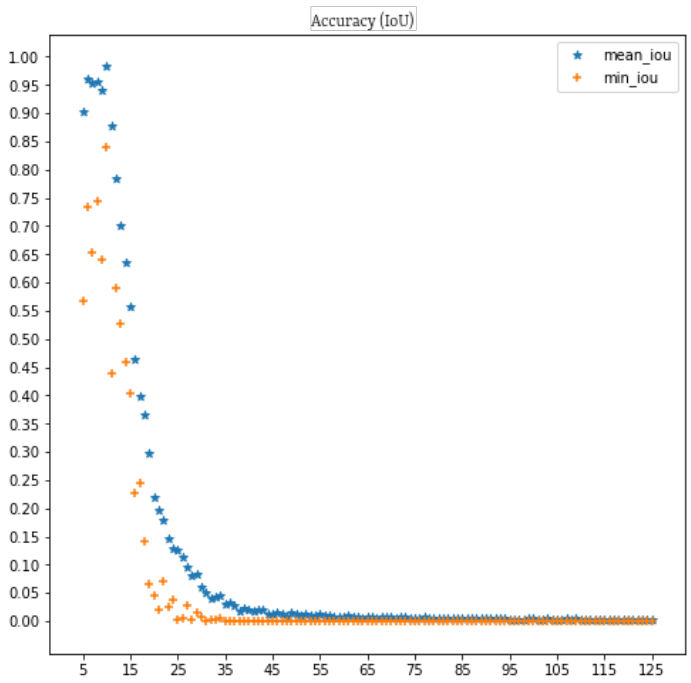


Figure 2.23 The performance of the UNet++ for small exponential block samples.

Figure 2.23 shows the performance of the UNet++ for small exponential block samples on the test set. The mean IoU accuracy of samples with exponential blocks size from 5 to 10, are all higher than 90%. Also, for all the small exponential block samples, we do not have non-predictive results, especially on samples with size 5 exponential block. Since we only have samples with exponential block size from 5 to 10, the accuracy of test samples drop dramatically with the increasing in sizes of exponential blocks.

In order to make a good prediction for both small and large exponential block samples, we need to combine those two UNet++.

2.6.3 Ensemble of UNet++

Currently, we already have two deep learning models: one of them is UNet++ for large exponential block samples and the other is UNet++ for small exponential block samples. UNet++ for large exponential block samples has good performance for most of the samples,

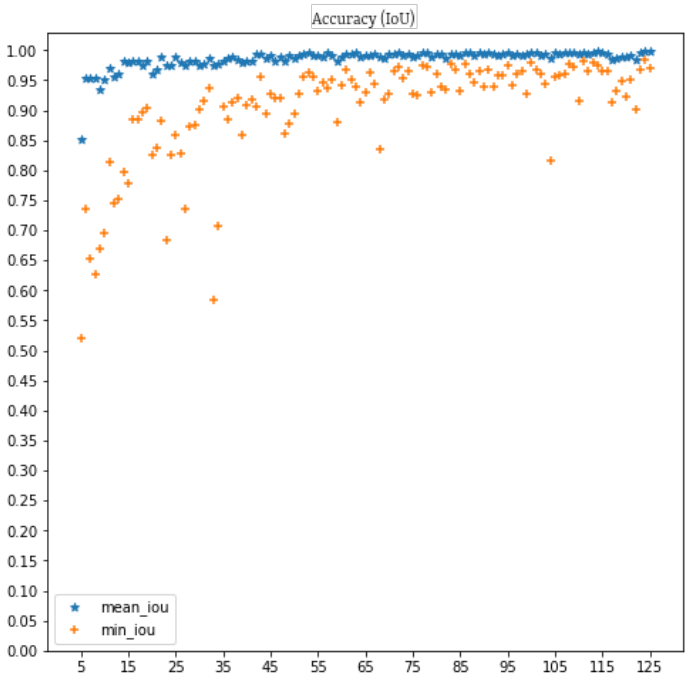


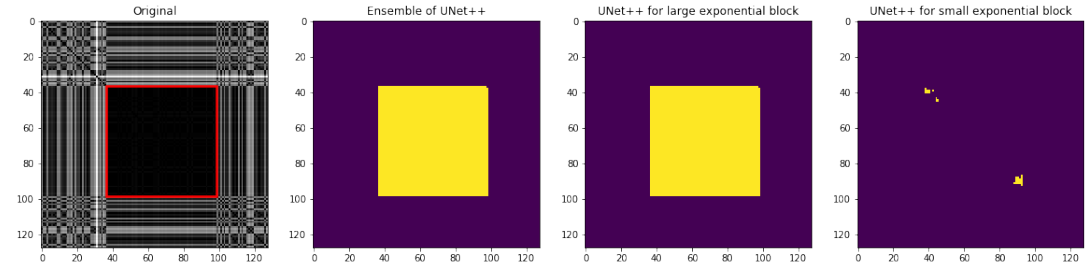
Figure 2.24 The performance of the ensemble of UNet++.

except for samples with size 5 exponential block. UNet++ for small exponential block samples has good performance for samples with exponential block range from 5 to 10. In order to get the best performance, we use the idea from ensemble method to combine two UNet++ networks. Instead of using the results from a single UNet++, we use the addition of results from both UNet++ networks. Let N_L denote the prediction of UNet++ for large exponential block samples, and N_S denote the prediction of the UNet++ for small exponential block samples. The final prediction of the ensemble of UNet++, denote by $N_{ensemble}$, is

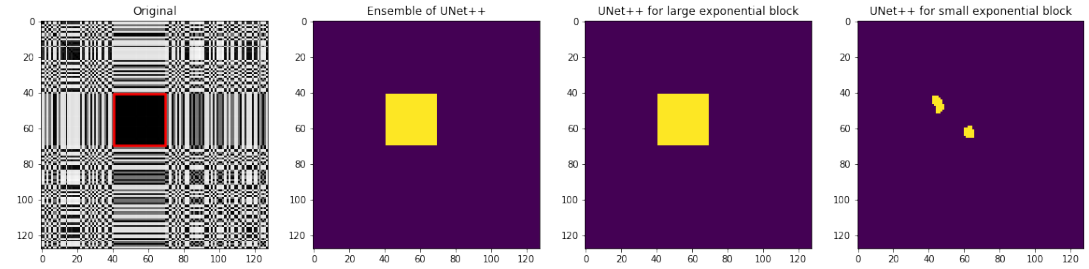
$$N_{ensemble} = S(N_L + N_S - 0.5), \tag{2.29}$$

where $S(\cdot) : \mathbb{R} \to \mathbb{R}$ is the step function. The performance of the ensemble of UNet++ is shown in Figure 2.24. Except for the samples with size 5 and 9 exponential blocks, the mean IoU accuracy for all other samples are higher than 95%. The minimal IoU accuracy of all test samples are higher than 50%. Compared with the best performance of the UNet++ for large exponential block samples, the mean IoU accuracy of samples with size 5 exponential block increased from 67.71% to 85.05% and the minimal IoU accuracy increased from 0.00% to 52.08%. This means that we do not have the non-predictive situation anymore.

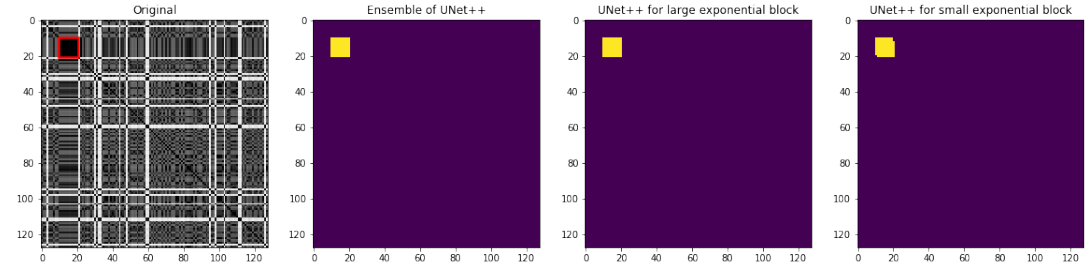
Figure 2.25 shows the predictions from the ensemble UNet++ and both single UNet++ networks. From Figure 2.25 (a) and (b), we observe that the UNet++ for large exponen-



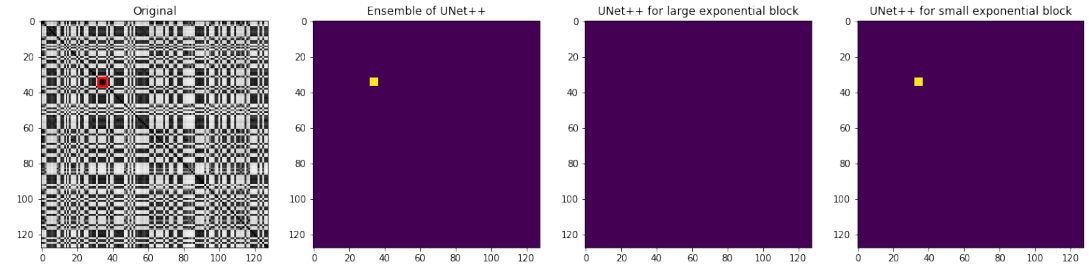
(a)



(b)



(c)



(d)

Figure 2.25 The predicted results for both UNet++ networks: (a) Sample with size 72 exponential block; (b) Sample with size 29 exponential block; (c) Sample with size 10 exponential block; (d) Sample with size 5 exponential block (Red rectangle are the true exponential block area, the yellow areas are the predicted results).

tial block samples has perfect predictions for large exponential block samples, the IoU accuracies are 99.97% and 100.00%, respectively. But, the UNet++ for small exponential block samples only got partial of the true blocks predicted. The IoU accuracies are 0.81% and 10.34%, respectively. Figure 2.25 (c) shows the sample with size 10 exponential block, both UNet++ networks have perfect performance. The accuracies are 100.00% and 97.52%, respectively. Figure 2.25 (d) shows the sample with size 5 exponential block, the UNet++ for large exponential block samples did not have any prediction(0% accuracy) and the UNet++ for small exponential block samples has the perfect prediction(100% accuracy). Figure 2.25 shows that the ensemble UNet++ has strong universal capabilities and the abilities to adapt to different size of exponential blocks. In Figure 2.25, the IoU accuracy of the ensemble UNet++ are 99.97%, 100%, 100% and 100% respectively.

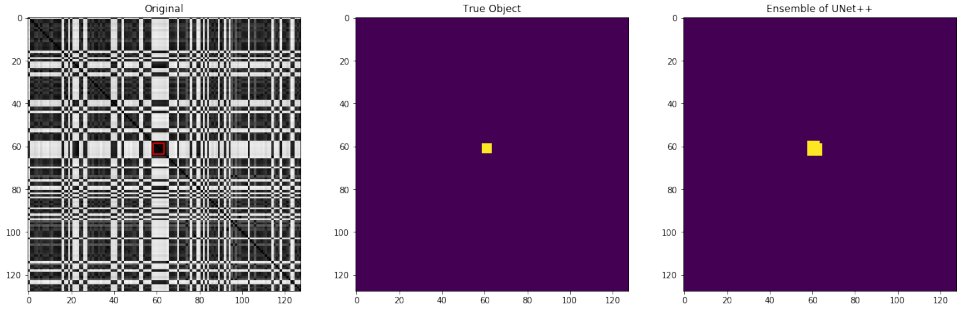
For all samples in the test set, we only have 3 cases with accuracy lower than 60%. Figure 2.26 shows the prediction of those 3 cases. The (a) and (b) in Figure 2.26 are samples with size 5 exponential block. Even though, the prediction area in Figure 2.26 (a) is much bigger than the true area, the prediction area contains the true area. There are two prediction areas contained in Figure 2.26 (b). But the prediction area also contains the true area. Figure 2.26 (c) is the sample with size 33 exponential block. We can see that the prediction area contains only part of the true area. This situation is the one we do not want. Because this may cause clinicians or biologists to leave out some virus samples with exponential growth.

2.6.4 Classifier

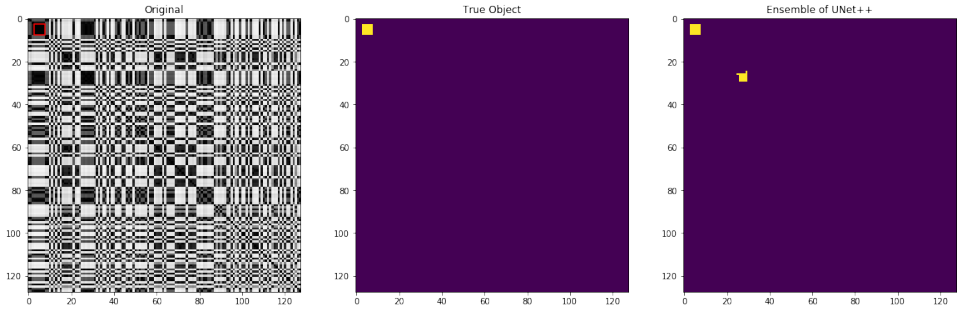
After we finished with all the prediction task, we want to make sure that all blocks we predict are exponential blocks. So, we want a classifier which can give the probability that a sequencing sample contains an exponential block. We use the idea of transfer learning to build our softmax classifier.

Since the UNet++ for large exponential block samples has a good performance for almost all the samples, we take out the backbone of UNet++ for large exponential block samples with the updated weights. Once we have a new sample, we can use this backbone to extract feature maps. Then we flatten these feature maps into a vector. This vector is a representation of the new sample. Two dense layers and dropout layers are added after this. After each dense layer, there is a dropout layer. We set the drop rate as 0.5 in both dropout layers. The last layer in our classifier is the softmax layer. The structure of the classifier are shown in Figure 2.27.

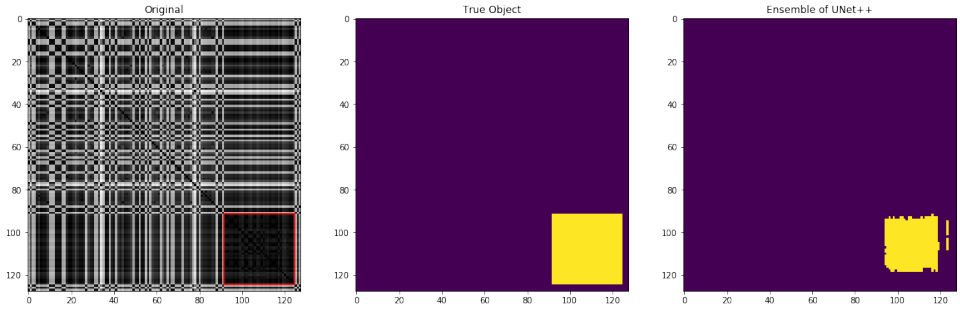
The training set for the classifier is the training set of the UNet++ for large exponential block samples with 2000 more samples that do not contain the exponential block. We divide



(a)



(b)



(c)

Figure 2.26 The samples with worst prediction of ensemble UNet++: (a) Sample with size 5 exponential block; (b) Sample with size 5 exponential block(a) Sample with size 33 exponential block.

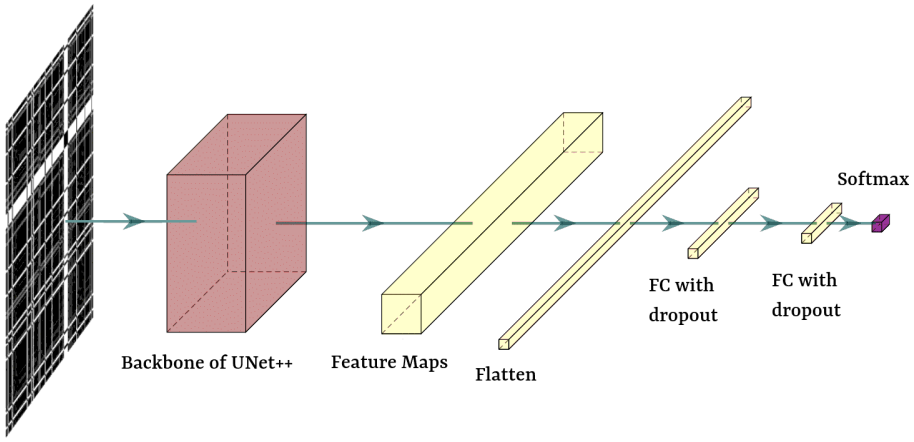


Figure 2.27 The structure of the classifier.

the rest of 1000 samples without exponential block into two groups: 400 of them will be added to the validation set, 600 of them will be added to the test set. Here, we choose the Adam method as the optimizer and the categorical cross entropy as the loss function. The learning rate for Adam is 0.001. We train the model for 20 epochs and the batch size sets as 16.

After we applied the classifier to the test set, the confusion matrix of the results are shown in Figure 2.28. We have 4230 samples in the test set, among them 3630 are the examples with exponential block, 600 are the examples without. The accuracy for examples without exponential block is 54.5%, which is not good. But the accuracy for examples with exponential block is 99.8%, which is very high. The main task here is to find the exponential blocks contained in the samples. We do not want to miss a single sample with exponential block. Hence, we can accept the classifier, even though it performs not as good on the samples without exponential block.

2.7 Conclusion

Table 2.1 shows the segmentation accuracy with different UNet++ models. The ensemble of UNet++ has the highest segmentation accuracy 98.55%.

Figure 2.29 shows the final results of our ensemble UNet++ with classifier. The ensemble UNet++ with classifier has strong universal capabilities and the abilities to adapt to the samples with different sizes of exponential blocks. We can also provide the precise

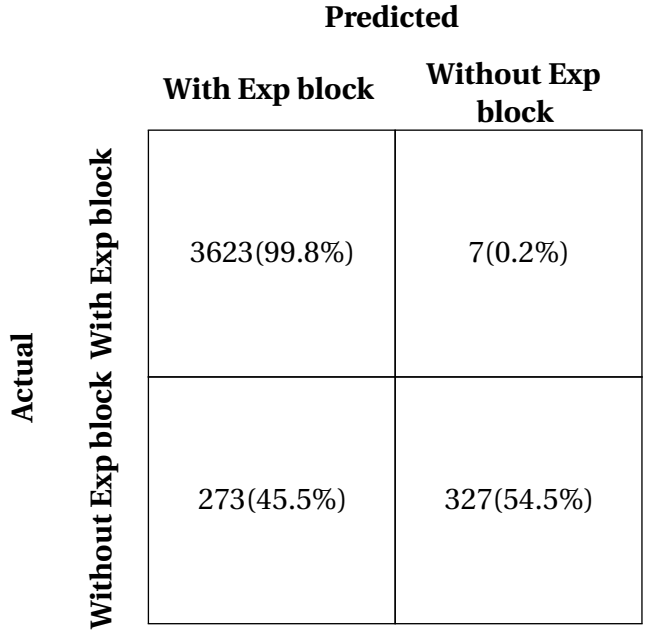


Figure 2.28 Confusion matrix for the classifier.

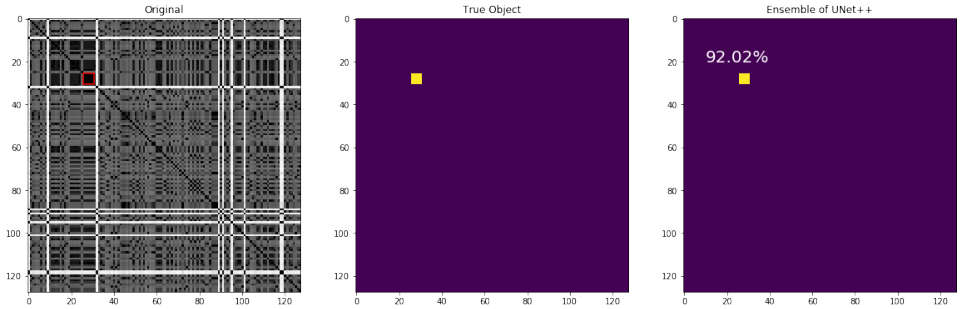
prediction for the exponential block contained in the sequencing samples. Based on the result of the classifier, if the detected exponential block with a probability higher than 50% , we will treat this detected exponential block as the true exponential block.

Table 2.1 Segmentation results for UNet++ for large exponential block samples, UNet++ for small exponential block samples and Ensemble of UNet++.

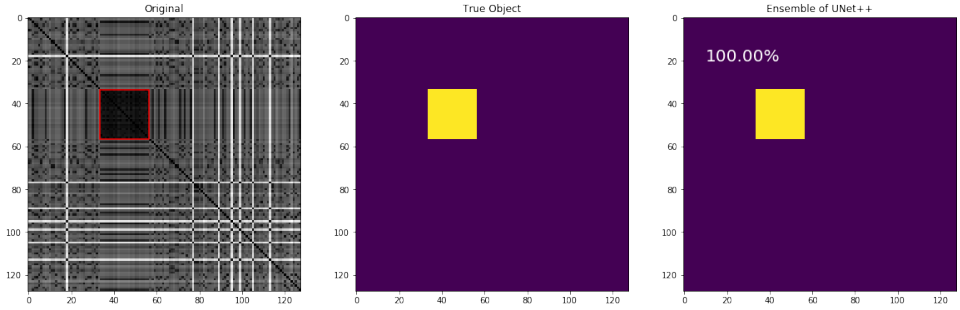
Architecture	Accuracy(IoU: %)
UNet++ for large exponential block samples	98.43
UNet++ for small exponential block samples	10.81
Ensemble of UNet++	**98.55**

2.8 Contributions

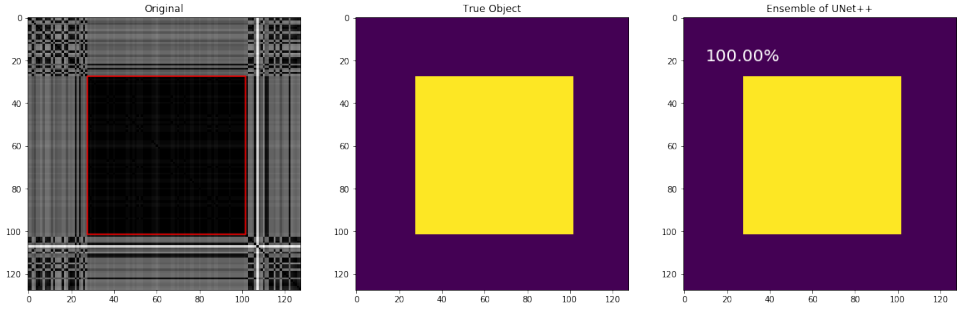
We are the first one who used deep learning method to find the exponential growth sub-groups in the virus sequencing data. We illustrate that the population changes would lead to differences in the pairwise difference matrices. We transfer the pairwise difference matrices



(a)



(b)



(c)

Figure 2.29 The final results of ensemble UNet++.

of sequencing data into heat maps. By using the idea of ensemble method, we build two UNet++ networks, one is designed for the large exponential block samples, the other is for the small exponential block samples. After we combined two UNet++, we can give precise prediction for the samples with exponential block range from 5 to 125. In the end, we added a classifier to our model. This provides the probability that a sample contains an exponential block.

2.9 Future Work

Currently, all the sequencing data are simulated data. The next step is to apply our proposed model to the real sequencing data. Also, the IoU accuracy of some samples with small exponential block are still low. There are some current state of art image segmentation methods that one can try to see if the results can be improved such as DeepLab V3 [Che17b], hierarchical multi-scale attention for semantic segmentation [Tao20], etc.

In our model, the classifier is being trained separately. One can try to find a way to add the classifier into the ensemble UNet++ networks in the future. Once this is done, we can train the segmentation networks and the classifier simultaneously. This can simplify the model and may help to increase the accuracy.

Another future consideration is to build a website so that clinicians or biologists can upload their real sequencing data to the website. The computations on the real sequencing data will be done on the remote server and the results will be displayed in some user friendly interface for the biologists or clinicians to do the analysis on their sequencing data.

BIBLIOGRAPHY

[Ada04] Adams, B. M. et al. “Dynamic multidrug therapies for HIV: Optimal and STI control approaches”. *Mathematical Biosciences & Engineering* **1**.2 (2004), p. 223.

[Ada05] Adams, B. M. et al. “HIV dynamics: modeling, data analysis, and optimal treatment protocols”. *Journal of Computational and Applied Mathematics* **184**.1 (2005), pp. 10–49.

[Agg18] Aggarwal, C. C. et al. *Neural networks and deep learning*. Springer, 2018.

[AHDDSG03] Anti-HIV Drugs (DAD) Study Group, D. C. on Adverse Events of. “Combination antiretroviral therapy and the risk of myocardial infarction”. *New England Journal of Medicine* **349**.21 (2003), pp. 1993–2003.

[Ant08] Antos, A. et al. “Fitted Q-iteration in continuous action-space MDPs”. *Advances in neural information processing systems*. 2008, pp. 9–16.

[Att17] Attarian, A. & Tran, H. *An optimal control approach to structured treatment interruptions for hiv patients: a personalized medicine perspective*. Tech. rep. North Carolina State University. Center for Research in Scientific Computation, 2017.

[Att12] Attarian, A. R. et al. “Patient Specific Subset Selection, Estimation and Validation of an HIV-1 Model with Censored Observations under an Optimal Treatment Schedule.” (2012).

[Bel57] Bellman, R. “A Markovian decision process”. *Journal of mathematics and mechanics* (1957), pp. 679–684.

[Ben75] Bentley, J. L. “Multidimensional binary search trees used for associative searching”. *Communications of the ACM* **18**.9 (1975), pp. 509–517.

[Bre96] Breiman, L. “Bagging predictors”. *Machine learning* **24**.2 (1996), pp. 123–140.

[Bre01] Breiman, L. “Random forests”. *Machine learning* **45**.1 (2001), pp. 5–32.

[Bre84] Breiman, L. et al. *Classification and regression trees*. CRC press, 1984.

[Cal02] Callaway, D. S. & Perelson, A. S. “HIV-1 infection and low steady state viral loads”. *Bulletin of mathematical biology* **64**.1 (2002), pp. 29–64.

[Che17a] Chen, L.-C. et al. “Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs”. *IEEE trans-*

actions on pattern analysis and machine intelligence **40**.4 (2017), pp. 834–848.

[Che17b] Chen, L.-C. et al. "Rethinking atrous convolution for semantic image segmentation". *arXiv preprint arXiv:1706.05587* (2017).

[Che16] Chen, T. & Guestrin, C. "Xgboost: A scalable tree boosting system". *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016, pp. 785–794.

[Den09] Deng, J. et al. "Imagenet: A large-scale hierarchical image database". *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[Dix90] Dixit, A. K., Sherrerd, J. J., et al. *Optimization in economic theory*. Oxford University Press on Demand, 1990.

[Duc11] Duchi, J. et al. "Adaptive subgradient methods for online learning and stochastic optimization." *Journal of machine learning research* **12**.7 (2011).

[Ern05] Ernst, D. et al. "Tree-based batch mode reinforcement learning". *Journal of Machine Learning Research* **6**.Apr (2005), pp. 503–556.

[Ern06] Ernst, D. et al. "Clinical data based optimal STI strategies for HIV: a reinforcement learning approach". *Proceedings of the 45th IEEE Conference on Decision and Control*. IEEE. 2006, pp. 667–672.

[Fra17] Francois, C. *Deep learning with Python*. 2017.

[Fre01] Freedberg, K. A. et al. "The cost effectiveness of combination antiretroviral therapy for HIV disease". *New England Journal of Medicine* **344**.11 (2001), pp. 824–831.

[Geu06] Geurts, P. et al. "Extremely randomized trees". *Machine learning* **63**.1 (2006), pp. 3–42.

[Glo11] Glorot, X. et al. "Deep sparse rectifier neural networks". *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 315–323.

[Goo16] Goodfellow, I. et al. *Deep learning*. Vol. 1. MIT press Cambridge, 2016.

[Gor99] Gordon, G. J. *Approximate solutions to Markov decision processes*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 1999.

[Gul02] Gulick, R. M. "Structured treatment interruption in patients infected with HIV". *Drugs* **62**.2 (2002), pp. 245–253.

[Hah01] Hahnloser, R. H. & Seung, H. S. “Permitted and forbidden sets in symmetric threshold-linear networks”. *Advances in neural information processing systems*. 2001, pp. 217–223.

[Hah00] Hahnloser, R. H. et al. “Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit”. *Nature* **405**.6789 (2000), pp. 947–951.

[He16] He, K. et al. “Deep residual learning for image recognition”. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[He17] He, K. et al. “Mask r-cnn”. *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2961–2969.

[Hin12] Hinton, G. et al. “Neural networks for machine learning lecture 6a overview of mini-batch gradient descent”. *Cited on* **14**.8 (2012).

[Hin06] Hinton, G. E. & Salakhutdinov, R. R. “Reducing the dimensionality of data with neural networks”. *science* **313**.5786 (2006), pp. 504–507.

[Ho95] Ho, T. K. “Random decision forests”. *Proceedings of 3rd international conference on document analysis and recognition*. Vol. 1. IEEE. 1995, pp. 278–282.

[Ho98] Ho, T. K. “The random subspace method for constructing decision forests”. *IEEE transactions on pattern analysis and machine intelligence* **20**.8 (1998), pp. 832–844.

[Hof17] Hofrichter, J. et al. *Information geometry and population genetics*. Springer, 2017.

[Hua17] Huang, G. et al. “Densely connected convolutional networks”. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.

[Jes14] Jessen, H. et al. “How a single patient influenced HIV research—15-year follow-up”. *The New England journal of medicine* **370**.7 (2014), p. 682.

[Kae96] Kaelbling, L. P. et al. “Reinforcement learning: A survey”. *Journal of artificial intelligence research* **4** (1996), pp. 237–285.

[Kal07] Kalyanakrishnan, S. & Stone, P. “Batch reinforcement learning in a complex domain”. *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*. 2007, pp. 1–8.

[Kin14] Kingma, D. P. & Ba, J. “Adam: A method for stochastic optimization”. *arXiv preprint arXiv:1412.6980* (2014).

[Lin17] Lin, T.-Y. et al. “Focal loss for dense object detection”. *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2980–2988.

[Lis99] Lisziewicz, J. et al. “Control of HIV despite the discontinuation of antiretroviral therapy”. *New England Journal of Medicine* **340**.21 (1999), pp. 1683–1683.

[Lon15] Long, J. et al. “Fully convolutional networks for semantic segmentation”. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 3431–3440.

[Lor00] Lori, F et al. “Structured treatment interruptions to control HIV-1 infection”. *The Lancet* **355**.9200 (2000), pp. 287–288.

[Mal11] Malof, J. M. & Gaweda, A. E. “Optimizing drug therapy with reinforcement learning: The case of anemia management”. *The 2011 International Joint Conference on Neural Networks*. IEEE. 2011, pp. 2088–2092.

[Mil16] Milletari, F. et al. “V-net: Fully convolutional neural networks for volumetric medical image segmentation”. *2016 fourth international conference on 3D vision (3DV)*. IEEE. 2016, pp. 565–571.

[Orm02] Ormoneit, D. & Sen, Ś. “Kernel-based reinforcement learning”. *Machine learning* **49**.2-3 (2002), pp. 161–178.

[Per96] Perelson, A. S. et al. “HIV-1 dynamics in vivo: virion clearance rate, infected cell life-span, and viral generation time”. *Science* **271**.5255 (1996), pp. 1582–1586.

[Put14] Puterman, M. L. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[Qui86] Quinlan, J. R. “Induction of decision trees”. *Machine learning* **1**.1 (1986), pp. 81–106.

[Qui93] Quinlan, J. R. “C4. 5: Programming for machine learning”. *Morgan Kauffmann* **38** (1993), p. 48.

[Rah16] Rahman, M. A. & Wang, Y. “Optimizing intersection-over-union in deep neural networks for image segmentation”. *International symposium on visual computing*. Springer. 2016, pp. 234–244.

[Ram17] Ramachandran, P. et al. “Searching for activation functions”. *arXiv preprint arXiv:1710.05941* (2017).

[Red16] Redmon, J. et al. “You only look once: Unified, real-time object detection”. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[Rie05] Riedmiller, M. “Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method”. *European Conference on Machine Learning*. Springer. 2005, pp. 317–328.

[Ron15] Ronneberger, O. et al. “U-net: Convolutional networks for biomedical image segmentation”. *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.

[Ros00] Rosenberg, E. S. et al. “Immune control of HIV-1 after early treatment of acute infection”. *Nature* **407**.6803 (2000), pp. 523–526.

[Rui00] Ruiz, L. et al. “Structured treatment interruption in chronically HIV-1 infected patients after long-term viral suppression”. *Aids* **14**.4 (2000), pp. 397–403.

[Sch15] Schmidhuber, J. “Deep learning in neural networks: An overview”. *Neural networks* **61** (2015), pp. 85–117.

[Sim14] Simonyan, K. & Zisserman, A. “Very deep convolutional networks for large-scale image recognition”. *arXiv preprint arXiv:1409.1556* (2014).

[Sri14] Srivastava, N. et al. “Dropout: a simple way to prevent neural networks from overfitting”. *The journal of machine learning research* **15**.1 (2014), pp. 1929–1958.

[Sut18] Sutton, R. S. & Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

[Sze16] Szegedy, C. et al. “Rethinking the inception architecture for computer vision”. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2818–2826.

[Tao20] Tao, A. et al. “Hierarchical Multi-Scale Attention for Semantic Segmentation”. *arXiv preprint arXiv:2005.10821* (2020).

[Tea07] Teague, A. “HIV: Now a manageable chronic disease”. *Pharmacy Times* **73**.3 (2007), p. 37.

[Vol13] Volz, E. M. et al. “Viral phylodynamics”. *PLoS Comput Biol* **9**.3 (2013), e1002947.

[Wat89] Watkins, C. J. C. H. “Learning from delayed rewards” (1989).

[Wei97] Wein, L. M. et al. “Dynamic multidrug therapies for HIV: a control theoretic approach”. *Journal of Theoretical Biology* **185**.1 (1997), pp. 15–29.

[Wei93] Weiss, R. A. “How does HIV cause AIDS?” *Science* **260**.5112 (1993), pp. 1273–1279.

[Wie12] Wiering, M. & Van Otterlo, M. *Reinforcement learning*. Vol. 12. Springer, 2012.

[Wil93] Williams, R. J. & Baird, L. C. *Tight performance bounds on greedy policies based on imperfect value functions*. Tech. rep. Citeseer, 1993.

[Wit02] Witten, I. H. & Frank, E. "Data mining: practical machine learning tools and techniques with Java implementations". *Acm Sigmod Record* **31**.1 (2002), pp. 76–77.

[Zho18] Zhou, Z. et al. "Unet++: A nested u-net architecture for medical image segmentation". *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support*. Springer, 2018, pp. 3–11.