上海交通大学学位论文

时序差分算法的隐式偏好与连续建模

姓　名：詹越
学　号：**519070910038**
导　师：罗涛
学　院：数学科学学院
学科/专业名称：数学与应用数学（吴文俊班）
申请学位层次：学士

2023 年 **05** 月

A Dissertation Submitted to
Shanghai Jiao Tong University for Bachelor Degree

TEMPORAL DIFFERENCE ALGORITHM'S IMPLICIT BIAS AND CONTINUOUS MODELING

Author: Yue Zhan
Supervisor: Tao Luo

School of Mathematical Sciences
Shanghai Jiao Tong University
Shanghai, P.R.China
June 28th, 2023

上海交通大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全知晓本声明的法律后果由本人承担。

学位论文作者签名：詹越
日期：2023年 6 月 10 日

上海交通大学
学位论文使用授权书

本人同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。

本学位论文属于：

☑公开论文

□ 内部论文，保密 □1 年/□2 年/□3 年，过保密期后适用本授权书。

□ 秘密论文，保密 ___ 年（不超过 10 年），过保密期后适用本授权书。

□ 机密论文，保密 ___ 年（不超过 20 年），过保密期后适用本授权书。

（请在以上方框内选择打“√”）

学位论文作者签名：詹越　　指导教师签名：

日期：2023年 6 月 10 日　　日期：2023年 6月 10 日

摘 要

Q-学习算法为强化学习无模型算法中最为常用的一种算法, 根据梯度定义的不同又可以分为时序差分算法 (temporal difference algorithm, TD) 和残差梯度算法 (residual gradient algorithm, RG)。之前的一些研究实验结果表明，在一些常见环境下，时序差分算法往往能更快地学习到一个更好的策略，并且具有鲁棒性，而用残差梯度算法训练得出的解虽然最后的 Bellman 残差比较小，但是不能学习到一个好的策略。

在本文中，我们探索了时序差分算法的偏好与隐式正则化。我们通过一系列实验观察，首先发现在网格环境，所有单步奖励相同的设定下，在时序差分算法学到好的策略时，其各状态的价值函数大小与它们离最终状态的距离（以下简称距离）密切相关；随后引入一个与距离相关的观测量，用来量化策略的好坏；接着我们观察并比较了不同的状态嵌入（独热嵌入与坐标嵌入）下的时序差分算法与残差梯度算法学习到的策略与观测量的变化，实验验证了在不同状态嵌入下该观测量仍可以反映策略的好坏。

除了实验观测，我们在一个较为简单的环境下分析了时序差分算法的收敛性质，来说明在单步奖励为相同常数的情况下，价值函数与距离相关，从而在理论上说明我们发现的观测量确实可以反映策略的好坏。用梯度流建模计算出各价值函数的收敛点，并用数值实验验证了模型的准确性。

关键词：强化学习，Q-学习，时序差分算法，梯度流

ABSTRACT

Q-learning algorithm is the most commonly used model-free algorithm in reinforcement learning. Q-learning algorithms can be further divided into temporal difference algorithm (TD) and residual gradient algorithm (RG). Previous research experimental results show that in some common environments, TD tends to learn a better policy faster and the algorithm is robust, while training with RG yields a solution that does not learn a good policy although the Bellman residual is smaller.

In our work, we explore the implicit regularization and convergence property of TD. Through a series of experimental observations, we first find that in a grid world environment and all one-step rewards are the same, when TD learns a good policy, the value of each state is closely related to their distance from the final state (distance for short). Subsequently, a distance-dependent observation is introduced to quantify the goodness of the policy. We then observe and compare the variation of the learned policy and the observation for TD and RG under different state embeddings (one-hot embedding and coordinate embedding) and verify that the equidistant similarity can also reflect the goodness of policy in both embeddings.

Except experimental observation, we analyze the convergence properties of TD in a simple environment setting, to explain that in the setting where all rewards are the same, the values of states are related to their distance, subsequently prove that the equidistant similarity can partly reflect the goodness of policy. We then use gradient flow modeling to calculate the convergence point of each value function, and verify the accuracy of the model with numerical experiments.

Key words: Reinforcement learning, Q-learning, Temporal difference algorithm, Gradient flow

CONTENTS

1 Chapter One Intriduction

1.1 Background

Reinforcement learning (RL) is a machine learning approach that emphasizes on how to act based on the environment in order to maximize the expected benefits. In recent years, reinforcement learning has achieved huge success in a variety of domains, such as GO[1], computer games[2], robotics[3], and other control problems[4]. Artificial Neural networks, composed by connections between biological neurons as weights between nodes, have been applied successfully to speech recognition, image analysis and adaptive control, in order to construct software agents (in computer and video games) or autonomous robots[5].

The term reinforcement learning (RL) first began to appear in the engineering literatures in the 1960s. Among these literatures, Marvin Minsky had the most impact with the term reinforcement learning, which was introduced in his 1961 paper "Steps Toward Artificial Intelligence"[6]. After the 1961 paper was published, the term reinforcement learning began to be widely used.

Although reinforcement learning was first introduced by Minsky in his 1961 paper, this trial-and-error learning had begun to develop before that. It is now believed that the sources of reinforcement learning are closely related to two fields: namely, animal learning in psychology and the optimization theory of optimal control. Animal learning in psychology was also one of the fields of early AI algorithm research. Part of the direction of early AI research was an effort to mimic the processes of human brain and animal learning to build models and algorithms. One of the most important approaches was trial-and-error learning, and Minsky's PhD thesis was considered one of these directions. In the 1950s and 1960s, people including Minsky, Clark, Farley, and others worked in this field. However, much of the subsequent research in reinforcement learning turned out to be supervised learning algorithms, which were actually a system of pattern recognition and perceptual learning, although the concepts of reward and punishment were also present. The most representative of this type of approach is the system MENACE (Matchbox Educable Naughts and Crosses Engine) proposed by Donald Michie to solve Tic-Tac-Toe games[7]. Since then, the development of reinforcement learning has not been prominent.

Another development in animal trial-and-error learning in psychology that is also very noteworthy is temporal difference learning. It derives part of its concepts from animal psychology, drawing on the stimuli brought to animals by food or pain. This piece is not prominent, but is very important and is one of the foundational pieces of work for the classic reinforcement learning model Q-learning.

Optimal control is another area of research that is closely related to reinforcement learning. In the 1950s, optimal control was used to describe the problem of designing controllers to minimize a measure of the behavior of dynamic systems over time. One approach to solving this problem was developed in the mid-1950s by Richard Bellman and others by extending the 19th century theory of Hamilton and Jacobi. This approach uses the concept of a dynamical system state and value function or "best return function" to define a functional equation, now commonly referred to as the Bellman equation. A class of methods for solving optimal control problems by solving this equation is known as dynamic programming (Bellman, 1957a)[8]. Bellman (1957b)[9] also introduced a discrete stochastic version of the optimal control problem known as Markov decision process (MDP), and Ron Howard (1960)[10] devised an iterative approach to the MDP strategy. All of these are essential elements of modern reinforcement learning theory and algorithms.

In 1989, Chris Watkins proposed the model of Q-learning, which is a combination of temporal differencing and optimal control. It also unified the previous paths developed by different reinforcement learning. But this model could not be effectively solved for too many states, etc. Finally, in 2013, DeepMind proposed a reinforcement learning model based on deep learning, namely deep Q-network, and published an improved version in 2015. At this point, reinforcement learning entered the public eye. Nowadays, RL has a number of mature and well-known algorithms, such as Monte Carlo (MC), Q-learning, SARSA, deep Q-network (DQN), Deep Deterministic Policy Gradient (DDPG), Asynchronous Advantage Actor-Critic Algorithm (A3C) and so on.

Markov decision process (MDP) is a basic model of reinforcement learning[11]. It includes set of state spaces, set of actions, transfer probabilities and rewards after one step. A lot of algorithms have been developed to solve the MDP model. They can be further divided into two categories: model-based algorithms and model-free algorithms.

Deep learning (DL), proposed by Hinton et al.[12] in 2006, is a new field of Machine learning (ML). DL is the process of learning the intrinsic laws and levels of representation of sample data, and the information obtained from these learning processes can be of great help in the interpretation of data such as text, images, and sounds. Its ultimate goal is to enable machines to have the same analytical learning ability as humans, capable of recognizing data such as text, images, and sounds. DL is a complex machine learning algorithm that has achieved results in language and image recognition that far surpass previous related techniques. It has achieved many results in search technology, data mining, machine learning, machine translation, natural language processing, multimedia learning, speech, recommendation and personalization techniques, and other related fields. Deep learning enables machines to mimic human activities such as seeing, hearing, and thinking, solving many complex pattern recognition challenges and enabling significant advances in AI-related technologies.

The differences between deep learning models and traditional shallow learning models are:

- The deep learning model structure contains more layers, and the number of layers containing hidden nodes is usually more than 5, and sometimes even contains up to 10 or more layers of hidden nodes.
- The importance of feature learning for deep models is clearly emphasized, i.e., by layer-by-layer feature extraction, the features of data samples in the original space are transformed to a new feature space to represent the initial data, which makes the classification or prediction problem easier to implement. Compared with manually designed feature extraction methods, the data features obtained using deep model learning are more representative of the rich intrinsic information of big data.

Deep learning is a new research direction in the field of machine learning, which has made breakthroughs in recent years in several types of applications such as speech recognition and computer vision. The motivation lies in building models to simulate the neural connectivity structure of the human brain, and in processing these signals of images, sounds and texts, layering the data features through multiple transformation stages to give an interpretation of the data.

Deep reinforcement learning (DRL) is the integration of reinforcement learning and artificial neural networks. Benefiting from powerful computation, big data, new algorithms, mature architectures and software packages, we are delighted to see the rise of deep reinforcement of learning[13]. In deep reinforcement learning, the value function of each state in environment is represented as output of deep neural networks. Subsequently, related deep learning techniques can be applied in the training process. There are a number of techniques that can be used in DRL, such as deep Q-network (DQN)[14], asynchronous advantage actor-critic (A3C)[15], trust region policy optimization (TRPO)[16] and proximal policy optimization (PPO)[17].

Q-learning is a model-free reinforcement learning method that learns action values by using a table called Q-table. However, Q-learning has some limitations when the state and action spaces are large, continuous, or high-dimensional. It is hard to store and update a huge Q-table, and it is also hard to generalize to unseen states.

Deep Q network (DQN) is a deep reinforcement learning algorithm that combines Q-learning with deep neural networks. It is the most widely used model-free algorithm in DRL[18]. DQN solves these problems by using a deep neural network to approximate the Q-table. The network takes the state as input and outputs the action values for each possible action. The network is trained by using a variant of Q-learning update rule, which minimizes the difference between the target Q-value and the predicted Q-value. The target Q-value is computed by using the reward and the maximum Q-value for the next state, which is obtained by using another network called the target network. The target network is periodically updated by copying the weights from the main network. This helps to stabilize the learning process and avoid divergence.

DQN also uses some other techniques to improve its performance, such as experience replay and ε-greedy exploration. Experience replay is a method of storing and sampling previous transitions in a buffer, which breaks the temporal correlations and reduces the variance of the updates. ε-greedy exploration is a method of balancing exploration and exploitation by choosing a random action with a small probability ε, and choosing the greedy action with probability 1-ε.

DQN was first proposed by Mnih et al.[14], who applied it to play various Atari games at human or superhuman level. Since then, DQN has been extended

and improved by many researchers, such as Double DQN, Dueling DQN, Prioritized Experience Replay DQN, Rainbow DQN, etc.

There are two types of loss function in DQN model, Bellman residual loss (Bootstrapping loss) and Monte Carlo loss. There exists a max operator in Bellman residual loss in Q-learning, so this loss can be called Bellman optimal residual Loss. According to different gradient definitions of Bellman residual loss, Q-learning algorithms can be further divided into two types: temporal difference algorithms (TD) and residual gradient algorithms (RG)[19].

Temporal difference (TD) learning is a class of model-free methods that learn by bootstrapping from the current estimate of the value function. It samples from the environment and performs updates based on current estimates. In recent years, TD is a popular algorithm used in many fields, such as:

- TD-Gammon: A program that learned to play the game of backgammon at the level of expert human players by using TD(λ) algorithm.
- AlphaGo: A program that learned to play the game of Go and defeated the world champion by using a combination of deep neural networks and TD learning.[1]
- Atari games: A program that learned to play various Atari games at human or superhuman level by using Q-learning, a TD algorithm that learns action values.[14]
- Robot navigation: A program that learned to navigate a robot in a maze by using SARSA, a TD algorithm that learns on-policy action values.[3]

TD methods adjust predictions to match later, more accurate, predictions about the future before the final outcome is known. This is a form of bootstrapping, as illustrated with the following example: Suppose you wish to predict the weather for Saturday, and you have some model that predicts Saturday's weather, given the weather of each day in the week. In the standard case, you would wait until Saturday and then adjust all your models. However, when it is, for example, Friday, you should have a pretty good idea of what the weather would be on Saturday – and thus be able to change, say, Saturday's model before Saturday arrives. The simple form of TD is called TD(0). More general TD methods, such as TD(λ) and n-step TD, are introduced later and put into massive applications.

The advantage of temporal difference algorithm over dynamic programming (DP) algorithm is that it does not need to know the model of the environment and can learn directly from experience. The advantage over the Monte Carlo (MC) algorithm is that instead of waiting for the end of an episode to update the value function, it can be updated at each time step, thus improving the learning efficiency. The disadvantage of temporal difference algorithm is that it may produce large biases because it uses estimates to update the estimates and may be influenced by the initial values; also, it requires suitable step parameters and exploration strategies to ensure convergence and stability.

There are several ways to improve TD methods. First, designing more effective exploration strategies, such as ε-greedy, softmax, UCB, etc., to balance the exploration and exploitation trade-offs. Second, using function approximators, such as neural networks, linear regression, etc., to deal with high-dimensional or continuous state spaces and action spaces. Third, using multi-step or backtracking methods, such as n-step TD, TD(λ), etc., to regulate the tradeoff between single-step and Monte Carlo updates. Fourth, using multiple strategies or objectives, such as Q-learning, Expected Sarsa, etc., to achieve off-policy learning or variance reduction. In our work, we mainly use TD in deep Q network.

Although TD algorithm is powerful and useful in a variety of fields, theoretical analysis of TD's convergence property or implicit regularization is rarely mentioned. So we naturally hope to discover why TD is so strong and robust in many different kinds of RL problems.

In our work, we explore the preference and convergence properties of temporal difference algorithm under different environmental settings, and investigate the properties of the value function when a good policy is learned. Through a series of experimental observations, we firstly notice that in the grid world when temporal difference algorithm learns a good policy, the size of its respective state value function size is closely related to their distance from the final state (distance for short). Specifically, values of states with the same distance is nearly the same, and values of states with different distances decreases with increasing distance. Subsequently, we introduce a distance-dependent observation (called equidistant similarity) to quantify the goodness of the policy. To confirm the criterion of this observation, the performance of TD with different state embeddings (one-hot embedding and coordinate embedding) is observed and compared. The equidistant

similarity reveals the goodness of policy in both embeddings, implying that it is an effective indication of the goodness of policy.

Except from experiment observations, we analyze the convergence properties of temporal difference algorithm in a simple environment setting, using continuous gradient descent modeling to calculate the convergence point of each value function, and verify the accuracy of the model with numerical experiments.

1.2 Related works

There have been lots of comparisons between TD and RG. Although TD's gradient is not the true gradient of loss, it was introduced by Sutton in 1988[20], earlier than RG. RG was first introduced by Baird in 1995[19] to serve as an alternative method for solving Bellman residual minimization problem. However, comparisons between TD and RG were not so abundant at that time. In 2003, Schoknet and Merke conclude that TD(0) converges faster than RG under linear function approximation and one-hot embedding settings[21]. On the one hand, it's hard to extend their work on Q-learning settings because their proof relies on the iterative structure of linear network. On the other hand, in the presence of linear function approximation, the TD(0) algorithm may diverge when transitions are arbitrarily sampled. Overall, it seems that the use of linear function approximation can affect the convergence behavior of these algorithms and may require alternative approaches such as RG to achieve convergence. In 2008, Li uses stochastic analysis to prove that in worst cases, TD(0) tends to make smaller prediction errors, while RG tends to make smaller temporal differences[22]. This is also based on linear approximation and hard to extend to Q-learning settings. Scherrer compares the loss of TD and RG and the solution of TD and RG in MDP with given policy. He does not state the relationship between the loss and the goodness of policy[23]. Zhang et al.[24] combine RG and Deep Deterministic Policy Gradient (DDPG) to construct Bi-Res-DDPG method. It tends to modify RG in training RL models. Some previous research experimental results show that in some common environments, TD tends to learn a better policy faster and the model is robust, while RG yields a solution that does not learn a good policy although the final Bellman residual is smaller[25].

There are a few papers about finite-time analysis of TD learning. Bhandari

et al.[26] give a simple and explicit finite-time analysis of TD learning when using linear function approximation. They make use of standard techniques for analyzing stochastic gradient descent algorithms and gives some key insights. They also discuss how the main results can be extended to the study of a variant of Q-learning applied to optimal stopping problems. In 2022, Sun et al.[27] establish the finite-time analysis for the adaptive TD with multi-layer ReLU network approximation whose samples are generated from a Markov decision process. They prove that adaptive TD algorithms guarantee convergence when the neural network is sufficiently wide, and adaptive TD with neural network approximation converges to a projected optimal action-value function. They also prove that adaptive TD can converge faster than the classical one with multiple-layer ReLU network. But the loss in their work does not have the max operator. Lee[28] analyzes linear stochastic iterative algorithms and TD learning from a control theoretic perspective and gives the finite-time analysis of TD learning.

The idea of using stochastic gradient descent (SGD) to construct model has a long history. An overview of large amounts of SGD's researches can be seen, for example in Bottou[29]. There are some widely used methods by modifying SGD, such as AdaGrad[30], RMSProp[31], Adam[32]. Stochastic differential equations (SDE) is the continuous version of SGD. SDE is an ideal model but can guide us to predict the results of the discrete model. It can be used in optimal non-linear filtering[33], stochastic control, optimal stopping and other fields like Mathematical Finance[34]. Li et al.[35] create a stochastic modified equation (SME) as an approximation of the SGD and prove the order of the approximation. By some solvable SME they can compute the dynamics of the corrosponding SGD. An et al.[36] also propose SMEs for modeling the asynchronous stochastic gradient descent (ASGD) algorithms and they show the convergence of ASGD to the SME in the continuous time limit. As an application, they propose an optimal mini-batching strategy for ASGD via solving the optimal control problem of the associated SME.

1.3 The significance of this article

The contributions of our work can be concluded as follows:

- In the setting of a constant reward function, we observe a quantity, i.e., the *equidistant similarity*, that can better indicate the goodness of policy than

Bellman residual error.

- We conduct a series of experiments to verify that the equidistant similarity is effective in different embeddings.
- We construct a continuous model of TD by using ODE in an ideal case and try to give a proof that the policy is closely related to the equidistant similarity.

2 Chapter Two Preliminaries

2.1 Deep learning

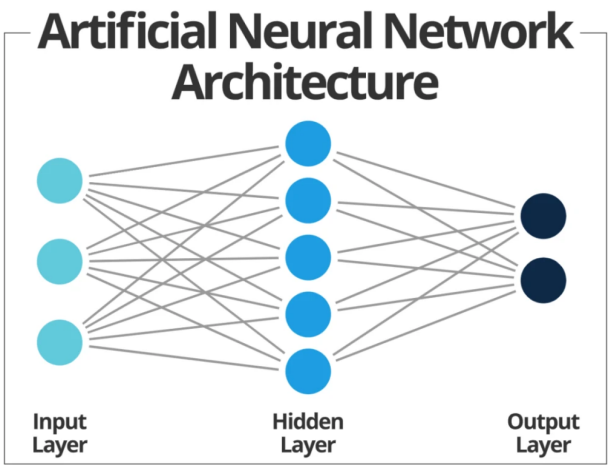


Figure 2-1 A sketch of neural network[37]

Deep learning is a class of the broad family of machine learning, which is based on artificial neural networks. Traditional machine learning methods, such as linear regression, logistic regression, support vector machine (SVM), only have one input layer and one output layer. In other words, you input a feature, then the model outputs a result, which is compared with the training data. In deep learning, the model has many hidden layers other than input layer and output layer. One hidden layer is composed by many units called neurons. In a full connected layer, each neuron in one layer is connected with all neurons in the next layer, we call all the connections as "weights", usually denoted by W. There is also an activation layer to serve as the nonlinear function in approximation. For example, some common activation layers are ReLU, sigmoid, tanh and so on. The activation layer is denoted as $\sigma(\cdot)$.

We denote the input data as (x, y), and assume that there are L layers of full connected layers and L activation layers, each of which is between two full connected layers. Then the output can be represented as:

$$f(x) = \sigma_L(W_L(\sigma_{L-1}(W_{L-1}(\cdots(W_1(x)+b_1)+\cdots)))+b_L)$$

The process above is called *forward propogation*. We have to use the loss function $L(W;x) = \frac{1}{n}\sum_{i=1}^{n}[f(x_i) - y_i]^2$ (n denotes the number of training data) to solve all partial derivatives of each weight. Then we can use gradient descent method to update all weights in the network. This process is called *back propogation*.

These two processes are mutually reinforcing. When the training begins, we first conduct forward propogation to get the predict value, then we compute the loss function and update the weights by back propogation. After one cycle of training, we get a new set of weights. Then we can start another cycle. In this process, the loss function will descend continually and reach a minimum or local minimum, i.e., the weights (or parameters $\theta = \mathrm{vec}(\{w_l, b_l\}_{l=1}^{L})$) will reach an optimal or local optimal.

There are also some tricks to modify the training of neural network. Dropout[38] is a regularization trick to prevent neural networks from overfitting by simply removing some of non-output neurons randomly from the original network. Batch normalization[39] conducts normalization on each mini-batch (a small batch of total training data), to accelerate the training by reducing the effect that change of one layer's weights will change all latter distributions of layers' weights.

2.2 Elements of reinforcement learning

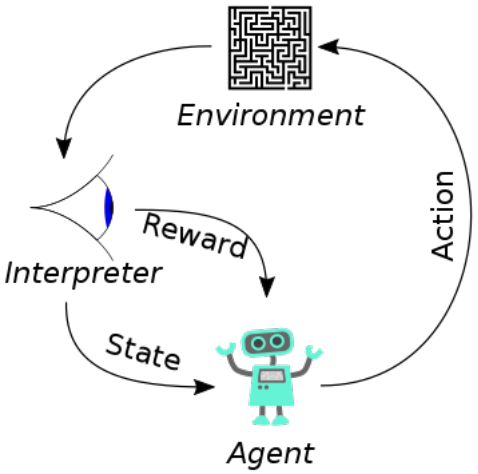


Figure 2-2 Typical framing of a RL scenario[40]

In general, RL includes two components, agent and environment. An agent takes actions in an environment, which is interpreted into a reward and a representation of the state, which are fed back into the agent. Beyond the agent and the environment, we can identify four main subelements of a RL system: a policy, a reward signal, a value function, and, optionally, a model of the environment[41].

A *policy* gives a definition of the agent's behavior at a given time. In a deterministic environment, we can write a policy as a mapping $\pi : \mathcal{S} \to \mathcal{A}$, where $\mathcal{S}$ represents the state space in the environment and $\mathcal{A}$ represents the action space. When in a specific state, by the policy mapping the agent can know what action should be taken. The policy may be a function or a lookup table, while in general cases it may be a stochastic distribution for each action. The probability that $a_t = a$ if $s_t = s$ is denoted as $\pi(a|s)$.

A *reward signal* defines the characters of events for the agent. When the agent takes an action, the environment returns a number called reward. The reward can be positive or negative, which respectively means good or bad events for the agent. The goal of agent is to maximize the total reward during the long process of training. Rewards are defining features in RL problems, deciding the policy learned at last. In general, rewards may also be stochastic functions of states in the environment.

A *value function* indicates what is good in the long run. Roughly speaking, the value of a state is a total reward that the agent can obtain in the total training. It takes the states that are likely to follow the present state into account. We have to bear in mind that rewards are primary and intrinsic in the environment and values are secondary and being estimated. There are no values without rewards. In fact, methods of estimating values are the most concerned part in almost all RL problems.

The last element in RL is a *model* of the environment. With a model, one can predict the behavior of the environment. Methods for solving RL problems with a model are called model-based methods. The opposites are model-free methods, which are explicitly trial-and-error learners.

2.3 Markov decision process (MDP)

MDP is a basic and classical framework in RL problems. It involves the delayed rewards so the trade-off between immediate rewards and delayed rewards should be considered.

A MDP is a 4-tuple (S, A, P_a, R_a), where:

- S is a set of states called the *state space*.
- A is a set of actions called the *action space*.
- $P(s', r|s, a) = \mathbb{P}(s_{t+1} = s', R_t = r|s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' amd receive reward r at time $t+1$.
- $R(\cdot|s, a)$ is the distribution of immediate reward received after transitioning from s, due to action a. For simplicity, we denote $r(s, a) = \mathbb{E}[R_t|s_t = s, a_t = a]$ as the expectation of immediate reward.

During the training process, the agent plays the role of making decision and it interacts with the environment continually. At time t, the agent choose an action $a_t$to take, the environment will change the state from s_t to s_{t+1} responding to this action and give the agent a reward R_t. This is a basic step of training. In the end, the MDP and agent together form a trajectory: $s_0, a_0, R_0, s_1, a_1, R_1, s_2, a_2, R_2 \ldots$.

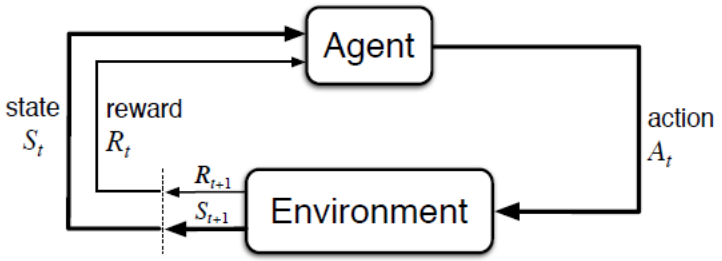


Figure 2-3 The agent−environment interaction in MDP[41]

At each step of training, the agent should take action according to a *policy*. A policy in general is a distribution that predicts the next state given current state and action. We denote $\pi(s'|a, s)$ as the probability that the next state is s' given current state s and action a. $\pi(\cdot|a, s)$ is the distribution given a and s.

MDP is an idealized form of RL problems for which theoretical analysis can be done. Next we introduce some key concepts that can be defined under this form.

The goal of RL is to maximize the "total" reward, i.e. the cumulative reward in the whole trajectory. For example, when the environment is a labyrinth, the agent should learn how to walk to the terminal. If the agent takes a step and does not arrive the terminal, it will receive a negative penalty. In the opposite, it will receive a positive reward. Remember that there are blind alleys in the labyrinth. Although some states look very close to the terminal, it might lead the agent to a blind alley. As a result, the agent will receive more negative penalties when go back from the wrong way.

A natural way to define the total reward is to sum all the rewards in the trajectory. From this idea we can define the *return* in the simplest case:

$$G_t := R_{t+1} + R_{t+2} + R_{t+3} + \cdots .$$

where t is the starting time. This definition makes sense when the trajectory is finite or there is a terminal time T such that G_t is a finite sum. In this case we call the process from the beginning of one training to the terminal as an *episode*. However, in many other cases, the interaction between agent and environment can't be devided into episodes. The sum of rewards may go on without limit. This will happen usually in continuing tasks, such as an application to a robot with a long life span. Subsequently, the return could easily be infinite. To solve this problem, we need to introduce a parameter called *discounting*. Literally, it means that once the agent takes a step, the reward of this step will multiply a discount rate γ. Hence we can give a refining definition of return:

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

where $\gamma \in [0,1]$ is called a *discounting rate*.

We restate that the goal of RL problems is to find a policy that maximize the return at each state. To achieve this goal, the concept of *value function* is indispensable. The value function of a state s under a policy π, denoted by $V_\pi(s)$,

is the expected return of state s following policy π:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s].$$

where $\mathbb{E}$ means the expectation with respect to the distribution $\pi(a|s)$.

Similarly, we can define an *action value function* $Q_\pi(s,a)$ as the expected return starting from state s, taking action a, following policy π:

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a].$$

There is an explicit relation between value function and action value function by the law of iterative expectation:

$$V_\pi(s) = \mathbb{E}[Q_\pi(s,A)|A \sim \pi(\cdot|s)]. \tag{2.1}$$

A useful property of return is that it can be written as a combination of immediate reward and the return starting from next time step:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \cdots) \\ &= R_{t+1} + G_{t+1}. \end{aligned} \tag{2.2}$$

By using this property we can rewrite the expression of value function $V_\pi(s)$:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[G_t|s_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{k+t+1}|s_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{k+t+2}|s_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|s_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma V_\pi(s')]. \end{aligned} \tag{2.3}$$

Equation (2.3) is the well-known Bellman equation for v_π. It reveals the relation between the value of current state and values of its successive states. That is to say, the value of a state is decided by the distributions of transition and reward and values of its successive states.

The value function of a state quantifies the importance of this state for solving RL problems. In finite MDP, we can find a policy π^* such that $V_{\pi^*}(s) \geq V_\pi(s), \forall s \in \mathcal{S}$. We call this π^* as an optimal policy, denoted as $\pi^* \geq \pi, \forall \pi$. Although there may be more than one optimal policy, the value function for all states in optimal policies are the same by our definition, called the optimal value function, denoted as V_*. It is naturally defined: $V_*(s) = \max_\pi V_\pi(s)$. Optimal action-value function can be defined similarly: $Q_*(s,a) = \max_\pi Q_\pi(s,a)$ It can be verified that $V_*(s) = \max_a Q_*(s,a)$.

The presence of $Q(s,a)$ makes the agent choose policy easilier. The agent only has to choose the action that maximizes $Q(s,a)$ and does not have to reach for one more step. Our following work uses deep neural network to compute $Q(s,a)$ rather than $V(s)$ directly.

2.4 Q-learning algorithm

Q-learning was firstly introduced by Watkins[18]. At present, Q-learning is the most popular model-free algorithm in RL. It is defined by

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[R_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)], \tag{2.4}$$

where α is the *learning rate* and γ is the *discounting rate*. We express this iteration by a *Bellman operator* T:

$$TQ(s,a) = r(s,a) + \gamma \mathbb{E}[\max_{a'} Q(s',a') | s' \sim P(\cdot|s,a)]. \tag{2.5}$$

We notice that $TQ_* = Q_*$, i.e. Q_* is a fixed point of operator T. It can be verified that the Bellman operator T is γ-contractive w.r.t the sup norm over $\mathcal{S} \times \mathcal{A}$. Subsequently, for arbitrary Q_0, sequence $\{Q_k\}_{k\geq 0}$ will converge to the optimal action-value function at a linear rate.

In deep Q-learning network (DQN), we approximate the action-value function $Q(s,a)$ by a deep neural network. Given a training data set $D_N = \{s_i, a_i, s_i', r_i\}_{i=1}^N$, and use θ to parameterize $Q(s,a)$. The empirical Bellman residual loss function

is defined as:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(Q_\theta(s_i,a_i) - r_i - \gamma \max_{a'} Q_\theta(s_i',a'))^2. \tag{2.6}$$

The gradient of loss function w.r.t. θ is calculated:

$$\nabla\mathcal{L}_{true} = \frac{2}{N}\sum_{i=1}^{N}(Q_\theta(s_i,a_i) - r_i - \gamma \max_{a'} Q_\theta(s_i',a'))(\nabla Q_\theta(s_i,a_i) - \gamma\nabla \max_{a'} Q_\theta(s_i',a_i)).$$

Methods that use this true gradient are called residual gradient (RG) method. There is another gradient defined as:

$$\nabla\mathcal{L}_{semi} = \frac{2}{N}\sum_{i=1}^{N}(Q_\theta(s_i,a_i) - r_i - \gamma \max_{a'} Q_\theta(s_i',a'))(\nabla Q_\theta(s_i,a_i)).$$

This semi-gradient is used for temporal difference (TD) method.

2.5 Gradient flow and gradient descent

In the beginning there is *time*. Time moves, and creates dynamics on space.

In continuous time, time flows continuously at a constant unit rate: $t \in \mathbb{R}^+$. Whereas in discrete time, time ticks at a discrete time step: $k \in \mathbb{Z}^+ = \{0,1,2,\cdots\}$. We can also view discrete time as a discretization of continuous time at some time step $\delta > 0$: $t = \delta k$ and we typically think of the time step δ to be small (infinitesimal, so low-order approximations suffice).

Let $\mathcal{X}$ be space. We assume $\mathcal{X} = \mathbb{R}^d$ for simplicity, but more generally $\mathcal{X}$ is a complete manifold. A *dynamics* on $\mathcal{X}$ is specified via a differential equation:

$$\dot{X}_t = F(X_t), \tag{2.7}$$

where $F : \mathcal{X} \to \mathbb{R}^d$ is a vector field that assigns the velocity vector $F(x)$ to each point x in space, which specifies where the point should go next. But concretely, if at time t we are at point X_t and know our velocity is $\dot{X}_t$, where should we go next? The problem is there is no "next" in continuous time. Instead, the instantaneous

velocity vector only tells us where we will approximately be in the near future:

$$X_{t+\delta} = X_t + \delta \dot{X}_t + O(\delta^2). \tag{2.8}$$

So if at time t the velocity is $F(X_t)$, then our best guess for where we should be at time $t+\delta$ is $X_{t+\delta} = X_t + \delta F(X_t)$.

So if we define the sequence $\{x_k := X_{\delta k}\}$, then we obtain an algorithm in discrete time that is defined iteratively via the update rule:

$$x_{k+1} = x_k + \delta F(x_k). \tag{2.9}$$

Thus, we can view algorithms in discrete time as a discretization of dynamics in continuous time, or dynamics as the continuous-time limit ($\delta \to 0$) of algorithms. Equivalently, dynamics in continuous time are templates that we can implement as algorithms in discrete time (which is what we ultimately care about) via some systematic discretization methods that maintain some provable guarantees.

The prototypical example we have in mind is the *gradient flow* dynamics in continuous time:

$$\dot{X}_t = -\nabla f(X_t), \tag{2.10}$$

and the corresponding *gradient descent* algorithm in discrete time:

$$x_{k+1} = x_k - \varepsilon \nabla f(x_k). \tag{2.11}$$

Gradient flow is a greedy method in continuous time that flows following the (locally-optimal) steepest descent direction. In particular, gradient flow is a descent method, which means it always decreases the objective function:

$$\frac{d}{dt} f(X_t) = \langle \nabla f(X_t), \dot{X}_t \rangle = -||\nabla f(X_t)||^2 \leq 0. \tag{2.12}$$

This is a natural property for optimization: Since our goal is to minimize f , it makes sense to decrease f as much as possible at each time. Thus, gradient flow is a simple and intuitive method for convex optimization in continuous time. Gradient descent is also a descent method, but (in contrast to continuous time) we

need an additional assumption that the objective function f be L-smooth (i.e., ∇f is L-Lipschitz, or $|\nabla^2 f(x)| \leq L$) and the step size ε be less than $\frac{2}{L}$. Moreover, the optimal step size choice is $\varepsilon = \frac{1}{L}$.

When it comes to ML or RL, the objective we want to minimize is $L(\theta)$. So in experiment, we update the parameters θ by using the gradient descent of this form:

$$\theta_{k+1} = \theta_k - \varepsilon \nabla_\theta L(\theta).$$

And we can construct a continuous model by gradient flow:

$$\dot{\theta}_t = -\nabla_\theta L(\theta). \tag{2.13}$$

2.6 Summary

In this section we first introduce the basic knowledge of DL and RL. Then we introduce the elements of MDP, an ideal model in RL. Next we introduce the well-known Q-learning algorithm, particularly temporal difference algorithm. Finally we explain the gradient flow and gradient descent and how it can be applied into RL. In the following several sections we will introduce our work.

3 Chapter Three Experiment Settings

We conduct our experiments mainly on the grid world of different size, a deterministic environment with finite number of states and actions. The action space is {up, down, left and right}. There is only one terminal state. If the agent reaches the terminal, it will receive a positive reward. Otherwise, each step has a small penalty. If the agent hits the boundary, it will stay at its current state.

In the grid world, discounting rate $\gamma = 0.9$, the terminal reward for the goal state is -0.05, the one-step penalty is -0.05. To control the randomness of training, we use a fixed data set under the off-policy scheme. The data set is generated by taking all the actions in all states other than the terminal state. When applying the SGD, we randomly choose a small batch of the full data set to finish one epoch of training. The deep neural network we use is simple, one linear layer or two linear layers or two linear layers with ReLU, which is convenient for analysis. Other parameters like learning rate, training epochs are kept the same in one trial, where we train the data set with both TD and RG algorithms. The optimizer of model is set to be 'sgd' as default. The discount of learning rate is set to be 1 as default, i.e. the learning rate η does not decrease and stays as a constant.

We use mainly two state embeddings: one-hot embedding and coordinate embedding. One-hot embedding is a square matrix Φ where each state $s_i$is represented as a column, denoted as $\phi(s_i)$. Each $\phi(s_i)$ is a vector where the i-th component takes $\sqrt{1-(|\mathcal{S}|-1)\varepsilon^2}$ and other components take ε. This embedding makes sure that $||\phi(s_i)|| = 1$ for all $i \in \mathcal{S}$ and $\phi(i)^\top \phi(j) =$ constant, for all $i \neq j$, which is more convenient for us to analysis. In our experiments, the ε is usually set as 0.05. The coordinate embedding defines each state vector as its corrosponding position coordinate. For example, in a 3×3 two-dim grid world, each state is a two-dim vector and the state embedding matrix has nine rows and two columns.

s_0	s_1	s_2
s_3	s_4	s_5
s_6	s_7	s_8

Table 3-1 An example of a 3×3 grid world, where s_8 is the terminal state. We set the index of states ranging from 0 to $n-1$ if $|\mathcal{S}| = n$. If the agent takes up or right in s_2, it will stay in s_2.

4 Chapter Four Observations of TD's Convergence Properties

We hope to explore the convergence properties of TD, so we conduct a series of experiments to record the final policies and state values of TD method. In this section we mainly use one-hot embedding to represent the states, while in the last subsection we use coordinate embedding to ocserve the performance of TD in different settings.

In several environments from simple to complex, we use one linear layer or two linear layers of neural network to approximate the values. We use fixed initial state values when using neural network with one linear layer, while we use random initialization of parameters in networks when using two neural network with two linear layers. The results are as follows.

4.1 Observations of experiments in 1-dim environments

Firstly, we set the environment to be a 1×5 grid world and let s_2 to be the terminal state . The environment looks like the following:

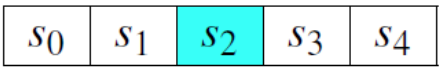


Figure 4-1 A 1×5 grid world with terminal state s_2

To have a reliable reference, we both use TD and RG methods to train the network.

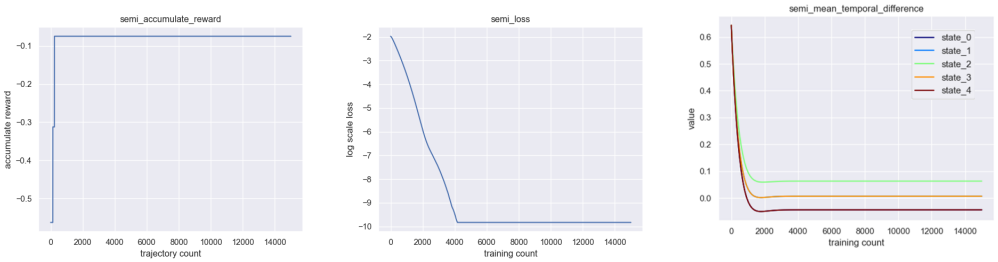
We first use one layer of linear networks with constant value initialization. We set values of all states as 1. And the initialization of parameters θ is set by solving the system of linear equations:

$$\Phi \cdot \theta(a) = Q(\cdot, a). \tag{4.1}$$

For simplicity, all $Q(\cdot, a)$ are set as $[1,1,1,1,1]^\top$. We set the training epoch as 15000, the learning rate as 0.01. The results are as follows:

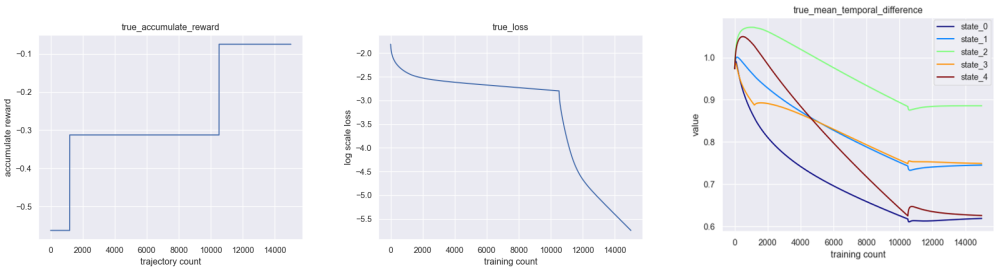
(a) TD's policy (b) RG's policy

Figure 4-2 Both TD and RG's policies are good.



(a) TD's accumulate reward (b) TD's loss (log scale) (c) TD's state values

Figure 4-3 Results of TD method. The accumulate reward means the mean value of total rewards of trajectories from all non-terminal states given a current policy. When the accumulate reward goes up, it means the policy turns better. The loss is the logarithm with a base of 10. The state values plot the values of all states.



(a) RG's accumulate reward (b) RG's loss (log scale) (c) RG's state values

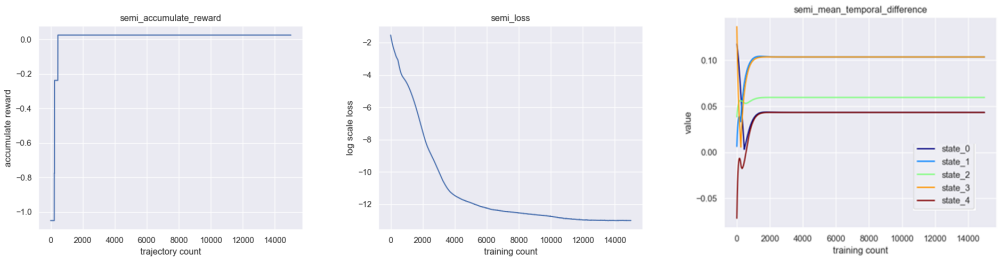
Figure 4-4 Results of RG method. The accumulate reward goes up to the same as TD's after around 10000 epochs, and the loss begins to descend quickly at that time.

From the results we see that although both TD and RG's policies are good, RG's learning is much slower than TD and the loss of RG is larger than TD.

Then we use two layers of linear networks with random initialization. For the stability in training, we set the means and standrad deviations of initialization of two layers' parameters as 0 and 0.1. We set the training epoch as 15000, learning rate as 0.01. The results are follows:

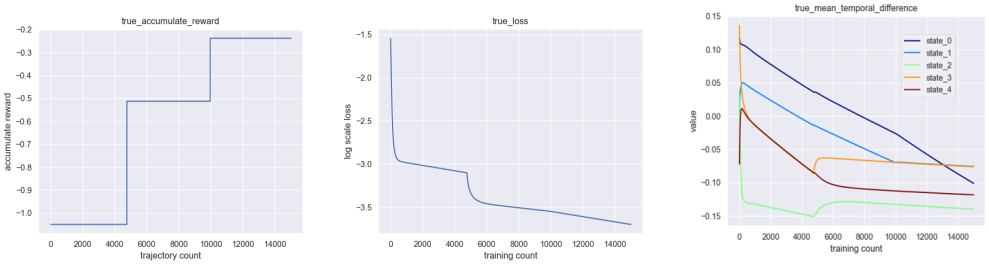
(a) TD's policy (b) RG's policy

Figure 4-5 TD learns a better policy than RG.



(a) TD's accumulate reward (b) TD's loss (log scale) (c) TD's state values

Figure 4-6 TD still learns faster and converge to the optimal policy quickly.



(a) RG's accumulate reward (b) RG's loss (log scale) (c) RG's state values

Figure 4-7 RG learn a bad policy and its learning is slower than TD. After 15000 epochs RG has not converge yet.

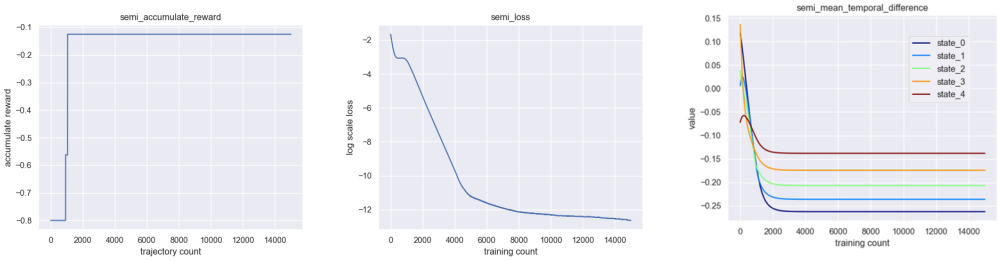
From the results we can see that TD learns a better policy than RG, the training process of RG is obviously much slower than TD, and RG's loss is larger than TD's. And TD's total accumulate reward is larger than RG's.

We also take repeat experiments 10 times. Sometimes RG also learns a good policy as well as TD, but the training process of RG is slower and the loss function of RG descends slowly. In the end of training, RG's loss is usually larger than that of TD.

From the above results, we actually have a basic understanding of the performances of TD and RG. To obtain more observations, we change the terminal state from 2 to 4, i.e. the rightmost cell in the grid world. And we repeat the experiment above to see how the results will change for the two algorithms in this setting. The results are as follows:

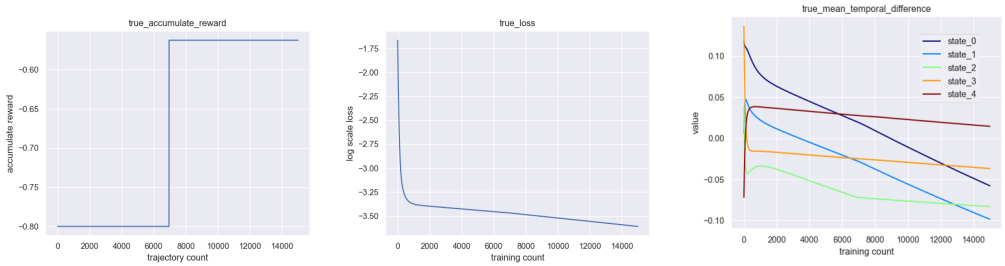
(a) TD's policy (b) RG's policy

Figure 4-8 TD learns a better policy than RG.



(a) TD's accumulate reward (b) TD's loss (log scale) (c) TD's state values

Figure 4-9 Results of TD method. We notice that other than the terminal s_4, the values of non-terminal states in the end decrease as their distance from the terminal state goes up. This criterion is also shown in previous experiment, where the terminal state is s_2.



(a) RG's accumulate reward (b) RG's loss (log scale) (c) RG's state values

Figure 4-10 Results of RG method. RG's policy is not good and its order of values does not satisfy the criterion we mentioned above, because s_0's value is larger than s_1's.

TD still performs better than RG. However, we discover some criterions about the values of states in TD's training. We make some definitions in the grid world first.

Definition 4.1. *Distance* of state s (denote as $d(s)$) is the least number of steps this state needs to walk to the terminal.

For example, in the 1×5 grid world and terminal s_2, s_0's distance $d(s_0) = 2$, because it needs to take two right steps to reach the terminal. The terminal state's distance is regarded as 0. Then we have the following criterion: when TD learns a good policy, states that have the same distance also have the same value, and states with longer distance have lower values (other than the terminal state).

This can be seen in the above experiments. When the terminal is s_2, values of s_1 and s_3 are very close. The same is as values of s_0 and s_4. When the terminal is s_4, values of other nonterminal states decrease as the distance grows. Notice that the initialization of parameters is random, this criterion is independent of the way of initialization. To confirm our observations, we train the model using TD in the 1×7 grid world, with terminal state s_4.

s_0	s_1	s_2	s_3	s_4	s_5	s_6

Table 4-1 A 1×7 grid world with terminal s_4

The values of each state are as follows:

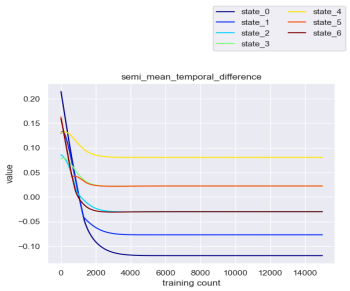
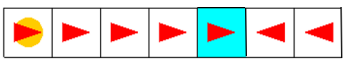
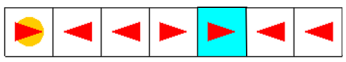


Figure 4-11 TD's values of each states in 1×7 grid world. We can see that $d(s_3) = d(s_5) = 1$, so s_3 and s_5's values are very close and their values are the largest other than the terminal state's. Both s_2 and s_6's distances are 2, so their values are smaller than the value of states with $d = 1$. s_0's distance is the largest, 4, so its value is the smallest.

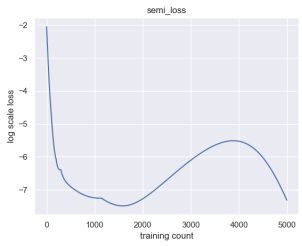
Besides the criterion, we also find an interesting phenomenon: a smaller loss (Bellman residual error) does not indicate a better policy. We draw the loss change and policies of TD and RG after 5000 epochs of training. The results are as follows:



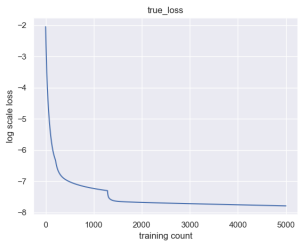
(a) TD's policy



(b) RG's policy



(c) TD's loss (log scale)



(d) RG's loss (log scale)

Figure 4-12 A smaller loss does not imply a better policy

This is corresponding to the previous research of the two algorithms[25]. So we need another observation to measure the goodness of the policy. The criterion we find gives us an idea about the observation. In fact, we can design an observation to quantify the closeness of values of the same distance.

4.2 Equidistant similarity: a measure of the goodness of policy

Assume that $\mathcal{S}$ is the state space, distances of all states value from 0 to D. Each distance i has k_i different states. The set of states corrosponding to the distance i is $S_i = \{s_i^j\}_{j=1}^{k_i}$. The set of state values corrosponding to the distance i is $V_i = \{V(s_i^j)\}_{j=1}^{k_i}$. Inspired by the idea in the previous subsection, now we can define *equidistant similarity* as follows:

Definition 4.2. *Equidistant similarity* (denoted as κ) is defined as the sum of variances of values in the same distance:

$$\begin{aligned}\kappa &:= \sum_{i=1}^{D} \mathrm{var}(V_i) \\ &= \sum_{i=1}^{D} \frac{1}{k_i} \sum_{j=1}^{k_i} (V(s_i^j) - \frac{1}{k_i} \sum_{l=1}^{k_i} V(s_i^l))^2\end{aligned} \tag{4.2}$$

This observation describes the closeness of different state values in the same distance. Next we will conduct the experiments in 2-dim grid world and see the change of κ and the relationship among loss, κ and policy.

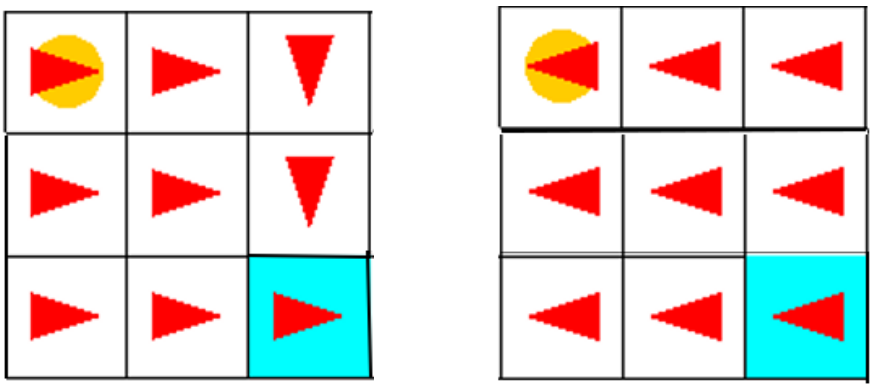
4.3 Observations of experiments in 2-dim environments

To testify the effectiveness of this observation, we still need more experiments in more complex environment settings. So we observe κ in 2-dim grid world. First we perform the experiments in 3×3 grid world with terminal state 8. The following is the environment:

s_0	s_1	s_2
s_3	s_4	s_5
s_6	s_7	s_8

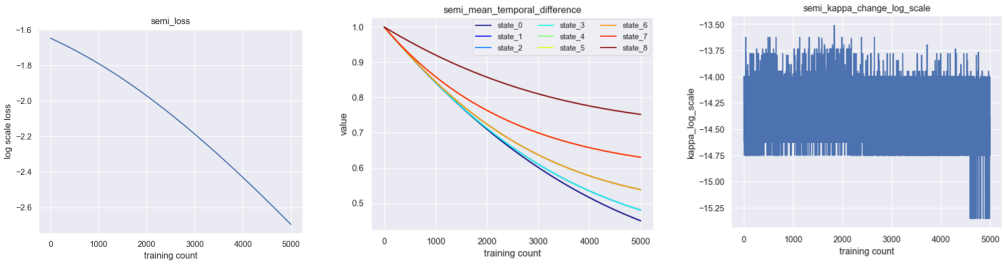
Table 4-2 3×3 **grid world with terminal state** s_8

Similarly, we first use one layer of linear network to train the model. And we plot the change of κ as the training count grows. We set training epochs as 5000, learning rate as 0.01. The initial values of all states are set 1. The results are as follows:

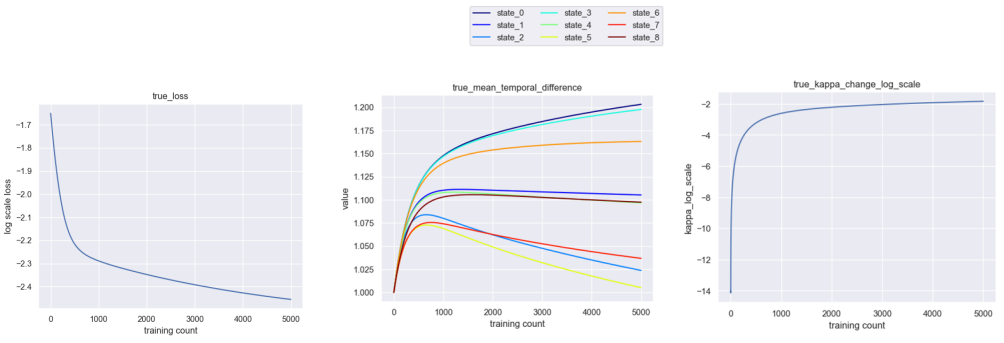


(a) TD's policy (b) RG's policy

Figure 4-13 TD learns a better policy than RG



(a) TD's loss (log scale) (b) TD's state values (c) κ_{TD} (log scale)

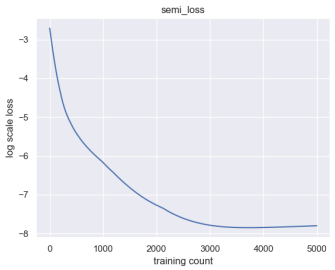


(d) RG's loss (log scale) (e) RG's state values (f) κ_{RG} (log scale)

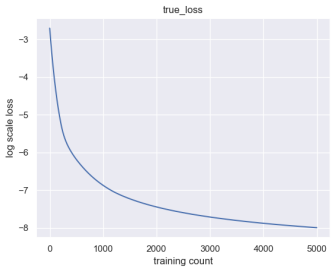
Figure 4-14 TD and RG's loss are close, but their κ vary greatly, which partly implies that TD's policy is better than RG's.

Then We use two layers of linear networks to train the model. To accelerate the training, we set the standrad deviations of initialization of two layers' param-

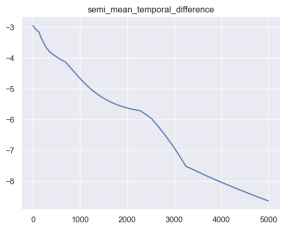
eters as 0.1 and 1. The size of network is set as 10×9 and 4×10. The results are as follows:



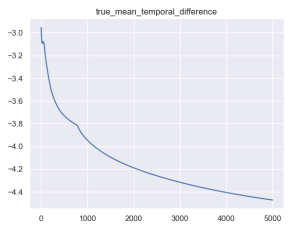
(a) TD's loss (log scale)



(b) RG's loss (log scale)



(c) κ_{TD} (log scale)



(d) κ_{RG} (log scale)

Figure 4-15 Loss and κ of TD and RG in 3×3 grid world

In this trial, TD's policy is better than RG's, which strengthens our experimental criterion and testifies the effectiveness of κ. Although TD's loss is close to RG's loss, we can compare their policies by their equidistant similarity. In repeat experiments, when TD and RG both learn the good policy, their loss and κ are both close.

4.4 Summary

In this section we first observe the training results of TD and RG. Then we find the criterion that when TD learns a better policy, the state values of the same distance are close and decrease as the distance grows. From this criterion we define an equidistant similarity κ, aiming to better describe the goodness of policy. Next we train the model in 3×3 grid world and observe the change of κ. The

results show that the criterion in 1-dim grid world also exists in 2-dim grid world and κ can somehow reflect the goodness of policy. Finally we train the model in coordinate embedding and show that training in one-hot embedding is more accurate and faster than in coordinate embedding, and κ can show its generalization in different embeddings.

In many other repeat experiments, there are cases where both loss and κ of TD and RG are close, but TD's policy is better than RG. This means that equidistant similarity is not a perfect indicator of the goodness of policy. The problem is that we do not consider the order relation of state values in different distance. We may modify this κ later.

5 Chapter Five TD's Performance In Different State Embeddings

5.1 TD's performance in coordinate embeddings

State embedding is an important topic in RL. One-hot embedding is effective when the state space is small. However, some RL problems may have large number of states or even uncountable number of states. For example, in the mountain car problem, there is a car placed stochastically at the bottom of a sinusoidal valley. There are two actions the car can take: accelerations in left or right. The goal of the problem is to strategically accelerate the car to reach the goal state on top of the right hill. The state representation of the car is a 2-dim vector: one component is the position of the car and the other is the velocity of the car. Both components are continuous so the state space is uncountable. In this case, we have to use state embedding in a lower dimension. Many other problems of large scale also have this requirements. Otherwise, the cost of computation can be much higher if using one-hot embedding. This is why we have to consider an embedding from large state space to a low dimensional representation.

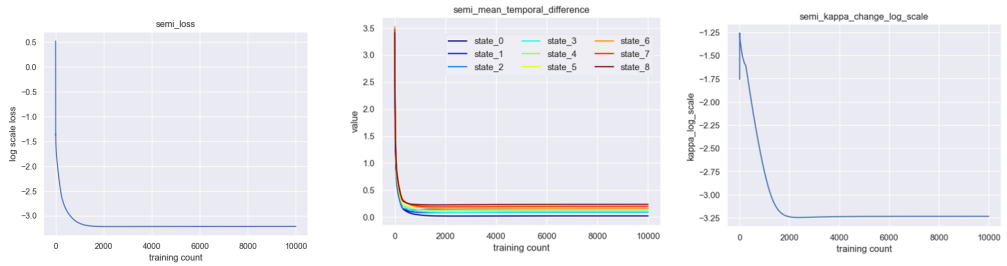
In this subsection we use coordinate embedding to represent the states in 2-dim grid worlds. For example, the representation of states in 3×3 grid world is as following:

(1,1) s_0	(2,1) s_1	(3,1) s_2
(1,2) s_3	(2,2) s_4	(3,2) s_5
(1,3) s_6	(2,3) s_7	(3,3) s_8

Table 5-1 State representation of coordinate embedding

First we still perform experiments in 3×3 grid world and watch the difference of TD's performance between one-hot embedding and coordinate embedding. Notice that the dimension of state representation changes from 9×9 to 9×2, which is much lower. To stabilize the training, we use two layers of lin-

ear networks with bias (without bias, the training in coordinate embedding can be certainly bad). The middle size of the network is set as 100. The training epoch is set as 10000. The result is as follows:

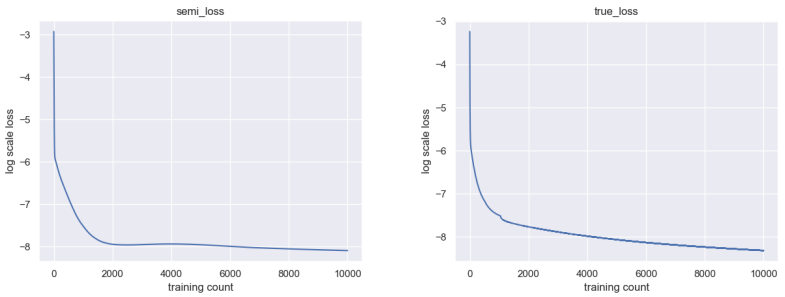


(a) TD's loss (log scale) (b) TD's state values (c) κ_{TD} (log scale)

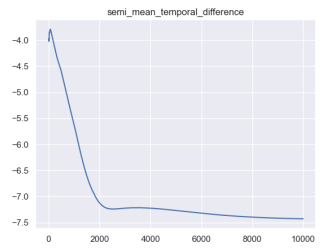
Figure 5-1 TD's performance in coordinate embedding

From the result, we can see that the criterion we have observed in one-hot embedding is still applicable to coordinate embedding. However, the loss and κ in coordinate embedding are both larger than those in one-hot embedding, which means that one-hot embedding performs better when the problem's scale is small.

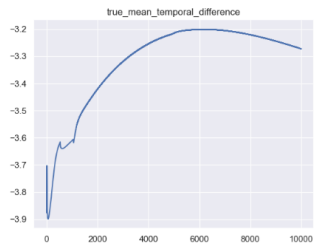
Besides, we hope to check that κ is an effective indicator of good policy not only in one-hot embedding but also in coordinate embedding. So we compare the loss and κ of TD and RG in 5×5 grid world with coordinate embedding. The result is as following:



(a) TD's loss (log scale) (b) RG's loss (log scale)



(c) κ_{TD} (log scale)



(d) κ_{RG} (log scale)

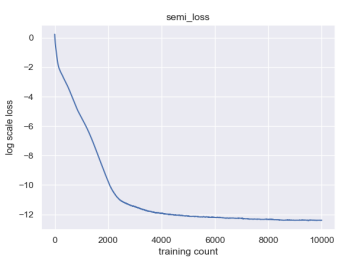
Figure 5-2 Performance of κ as an indicator of good policy

From the figures above we can see that although TD's loss and RG's loss have little difference, κ_{TD} is obviously smaller than κ_{RG}. The fact is that TD's policy is better than RG's, so κ is effective in different state embeddings.

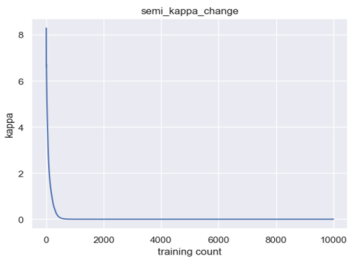
5.2 Trials in more complex environments

We hope to discover the difference between one-hot embedding and coordinate embedding, so we conduct the experiments in some more complex environments. We only use TD to compare the two embeddings. And we use two layers of linear networks with bias. The sizes of the two layers of linear networks is 100×25 and 4×100.

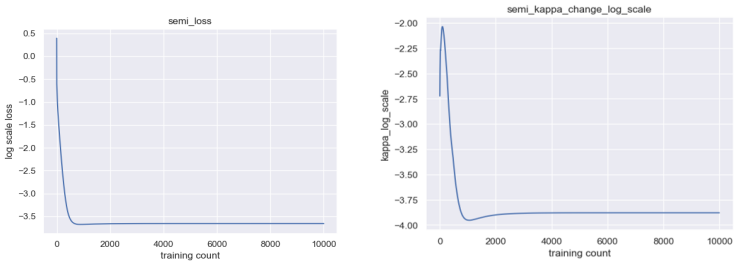
We first conduct the experiments in 5×5 grid world. We use TD to train the network in both one-hot embedding and coordinate embedding. We set training epoch as 10000, learning rate as 0.01. The results are as follows:



(a) One-hot's loss (log scale)



(b) $\kappa_{\text{one-hot}}$ (log scale)



(c) Coordinate's loss (log scale) (d) $\kappa_{\text{coordinate}}$ (log scale)

Figure 5-3 Both policies in two embeddings are good, but loss and κ in one-hot embedding are both smaller than in coordinate embedding. This implies that training in one-hot embedding is more accurate.

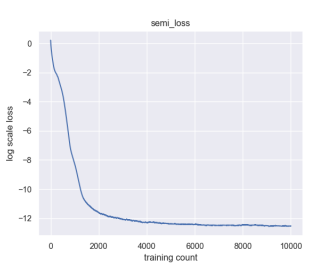
In the meanwhile, we record the time of training in both embeddings. The time of one-hot embedding is 227s, while the time of coordinate embedding is 116s. This means that in terms of the computation cost, coordinate embedding is better than one-hot embedding in large scale problems.

We also want to see that if coordinate embedding is better than one-hot embedding when the environment is more complex, so we dig a hole in the middle of 5×5 grid world. The environment is as follows:

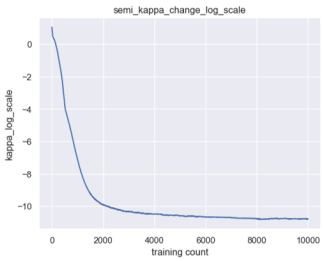
s_1	s_2	s_3	s_4	s_5
s_6	s_7	s_8	s_9	s_{10}
s_{11}	s_{12}		s_{13}	s_{14}
s_{15}	s_{16}	s_{17}	s_{18}	s_{19}
s_{20}	s_{21}	s_{22}	s_{23}	s_{24}

Table 5-2 A 5×5 grid world with a hole in the middle.

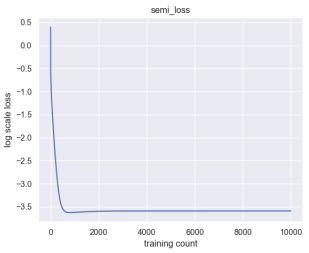
We still use TD and two layers of linear networks to train the model. The training epoch is 10000, learning rate is 0.01. The result is as following:



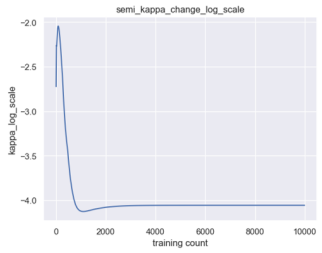
(a) One-hot's loss (log scale)



(b) $\kappa_{\text{one-hot}}$ (log scale)



(c) Coordinate's loss (log scale)



(d) $\kappa_{\text{coord.}}$ (log scale)

Figure 5-4 Policy in one-hot embedding is better than that in coordinate embedding. Loss and κ in one-hot embedding are both smaller than in coordinate embedding. This implies that training in one-hot embedding is more accurate.

Then we add a ReLU layer between the two linear layers and extend the middle size of network to 200, set learning rate as 0.05. In this setting training in coordinate embedding can be good. But its loss and κ are still larger than in one-hot embedding.

This implies that one-hot embedding does have advantage over coordinate embedding in complex environments in small scale problems. However, in large scale problems, we still need low dimensional embedding to lower the cost of computation.

5.3 Summary

In this section we try to train the model in complex environments in different embeddings. The result seems to show that one-hot embedding has great advantage over lower dimensional embeddings when solving small scale problems. The

loss and κ in one-hot embedding are much smaller than those in coordinate embedding, which means that training in one-hot embedding is more accurate. But when it comes much more states or even uncountable states, we still need to find a low dimensional embedding to save the cost of computation and speed up the training.

6 Chapter Six Continuous Modeling of TD

6.1 A convergence analysis of TD in linear case

In this section we try to write a continuous gradient flow of TD method, and compute the converge point of all $Q(s,a)$ in a simple environment. First we list the assumptions.

The basic setting of the environment is as follows. Give state set as $\mathcal{S} = \{s_0, s_1, s_2\}$, action set as $\mathcal{A} = \{a_1, a_2\}$. The terminal state is s_2. All rewards are a constant r. The data tuple is $\{s_0, a_1, s_0, r\}, \{s_0, a_2, s_1, r\}, \{s_2, a_1, s_1, r\}, \{s_2, a_2, s_2, r\}$. We train the model using one layer of linear network in one-hot embedding. So all $Q(s,a)$ can be represented as $Q(s,a) = \phi(s)^\top \theta(a)$. $\phi(s_i)^\top \phi(s_i) = 1$ and $\phi(s_i)^\top \phi(s_j) = \alpha < 1$. We assign the value $Q_0(s_0,a_1) = Q_0(s_0,a_2) = Q_0(s_2,a_1) = Q_0(s_2,a_2) = C > \frac{r}{1-\gamma}$ and the terminal value $Q(s_1,a_1) = Q(s_1,a_2)$ fixed as C.

s_0	s_1	s_2

Table 6-1 The simple MDP environment

Under the assumptions above, the loss function at first step is

$$\begin{aligned} L_0 &= \sum_{s,a,s',r} (Q_0(s,a) - r - \gamma \max_{a'} Q_0(s',a'))^2 \\ &= (\phi(s_0)^\top \theta(a_1) - r - \gamma \phi(s_0)^\top \theta(a_1))^2 + (\phi(s_0)^\top \theta(a_2) - r - \gamma C)^2 \\ &\quad + (\phi(s_2)^\top \theta(a_1) - r - \gamma C)^2 + (\phi(s_2)^\top \theta(a_2) - r - \gamma \phi(s_2)^\top \theta(a_2))^2 \\ &= 4((1-\gamma)C - r)^2. \end{aligned} \tag{6.1}$$

The gradient for $\theta(a_1), \theta(a_2)$ are

$$\begin{aligned} \nabla_{\theta(a_1)} L_0 &= 2\phi(s_0)(\phi(s_0)^\top \theta(a_1) - r - \gamma \phi(s_0)^\top \theta(a_1)) + 2\phi(s_2)(\phi(s_2)^\top \theta(a_1) - r - \gamma C) \\ &= 2(\phi(s_0) + \phi(s_2))((1-\gamma)C - r), \\ \nabla_{\theta(a_2)} L_0 &= 2\phi(s_0)(\phi(s_0)^\top \theta(a_2) - r - \gamma C) + 2\phi(s_2)(\phi(s_2)^\top \theta(a_2) - r - \gamma \phi(s_2)^\top \theta(a_2)) \\ &= 2(\phi(s_0) + \phi(s_2))((1-\gamma)C - r). \end{aligned} \tag{6.2}$$

The θ update is

$$\begin{aligned}\theta(a_1) &= \theta(a_1) - \eta\nabla_{\theta(a_1)}L_0,\\ \theta(a_2) &= \theta(a_2) - \eta\nabla_{\theta(a_2)}L_0.\end{aligned} \tag{6.3}$$

We notice that $\frac{dQ(s_i,a_i)}{d\theta(a_i)} = \phi(s_i)$, and $\frac{dQ(s_i),a_i}{dt} = \frac{dQ(s_i,a_i)}{d\theta(a_i)}\frac{d\theta(a_i)}{dt}$. The action value function update is

$$\begin{aligned}Q_1(s_0,a_1) &= Q_0(s_0,a_1) - \eta\phi(s_0)^\top\nabla_{\theta(a_1)}L_0 = C - 2\eta(1+\alpha)((1-\gamma)C-r),\\ Q_1(s_0,a_2) &= Q_0(s_0,a_2) - \eta\phi(s_0)^\top\nabla_{\theta(a_2)}L_0 = C - 2\eta(1+\alpha)((1-\gamma)C-r),\\ Q_1(s_2,a_1) &= Q_0(s_2,a_1) - \eta\phi(s_2)^\top\nabla_{\theta(a_1)}L_0 = C - 2\eta(1+\alpha)((1-\gamma)C-r),\\ Q_1(s_2,a_2) &= Q_0(s_2,a_2) - \eta\phi(s_2)^\top\nabla_{\theta(a_2)}L_0 = C - 2\eta(1+\alpha)((1-\gamma)C-r).\end{aligned} \tag{6.4}$$

Define

$$C_1 = 2\eta(1+\alpha)((1-\gamma)C-r) > 0. \tag{6.5}$$

Now we have $Q_1(s_1,a_1) = Q_1(s_1,a_2) > Q_1(s_0,a_1) = Q_1(s_0,a_2) = Q_1(s_2,a_1) = Q_1(s_2,a_2)$. The current policy is $\pi(s_0) = a_1, \pi(s_1) = a_1, \pi(s_2) = a_1$ (W.L.O.G). In the second step, the new loss function is

$$\begin{aligned}L_1 &= \sum_{s,a,s',r}(Q_1(s,a) - r - \gamma\max_{a'}Q_1(s',a'))^2\\ &= (Q_1(s_0,a_1) - r - \gamma Q_1(s_0,a_1))^2 + (Q_1(s_0,a_2) - r - \gamma C)^2\\ &\quad + (Q_1(s_2,a_1) - r - \gamma C)^2 + (Q_1(s_2,a_2) - r - \gamma Q_1(s_2,a_1))^2\\ &= 2((1-\gamma)(C-C_1) - r)^2 + 2(C - C_1 - r - \gamma C)^2.\end{aligned} \tag{6.6}$$

The gradient is computed as

$$\begin{aligned}\nabla_{\theta(a_1)}L_0 &= 2\phi(s_0)(\phi(s_0)^\top\theta(a_1) - r - \gamma\phi(s_0)^\top\theta(a_1)) + 2\phi(s_2)(\phi(s_2)^\top\theta(a_1) - r - \gamma C)\\ &= 2\phi(s_0)((1-\gamma)(C-C_1) - r) + 2\phi(s_2)(C - C_1 - r - \gamma C),\\ \nabla_{\theta(a_2)}L_0 &= 2\phi(s_0)(\phi(s_0)^\top\theta(a_2) - r - \gamma C) + 2\phi(s_2)(\phi(s_2)^\top\theta(a_2) - r - \gamma\phi(s_2)^\top\theta(a_2))\\ &= 2\phi(s_0)(C - C_1 - r - \gamma C) + 2\phi(s_2)((1-\gamma)(C-C_1) - r).\end{aligned} \tag{6.7}$$

The action value function update is

$$\begin{aligned}
Q_2(s_0,a_1) &= Q_1(s_0,a_1) - \eta\phi(s_0)^\top \nabla_{\theta(a_1)} L_1 \\
&= C - C_1 - 2\eta((1-\gamma)(C-C_1)-r) - 2\eta\alpha(C-C_1-r-\gamma C), \\
Q_2(s_0,a_2) &= Q_1(s_0,a_2) - \eta\phi(s_0)^\top \nabla_{\theta(a_2)} L_1 \\
&= C - C_1 - 2\eta\alpha((1-\gamma)(C-C_1)-r) - 2\eta(C-C_1-r-\gamma C), \\
Q_2(s_2,a_1) &= Q_1(s_2,a_1) - \eta\phi(s_2)^\top \nabla_{\theta(a_1)} L_1 \\
&= C - C_1 - 2\eta\alpha((1-\gamma)(C-C_1)-r) - 2\eta(C-C_1-r-\gamma C), \\
Q_2(s_2,a_2) &= Q_1(s_2,a_2) - \eta\phi(s_2)^\top \nabla_{\theta(a_2)} L_1 \\
&= C - C_1 - 2\eta((1-\gamma)(C-C_1)-r) - 2\eta\alpha(C-C_1-r-\gamma C).
\end{aligned} \tag{6.8}$$

Define

$$\begin{aligned}
C_2^1 &= 2\eta((1-\gamma)(C-C_1)-r), \\
C_2^2 &= 2\eta(C-C_1-r-\gamma C).
\end{aligned} \tag{6.9}$$

Then $Q_2(s_0,a_1) = Q_2(s_2,a_2) = C - C_1 - C_2^1 - \alpha C_2^2, Q_2(s_0,a_2) = Q_2(s_2,a_1) = C - C_1 - \alpha C_2^1 - C_2^2$. $C_2^1 - C_2^2 = 2\eta\gamma C_1 > 0$. Subsequently, we have $Q_2(s_1,a_1) = Q_2(s_1,a_2) > Q_2(s_0,a_2) = Q_2(s_2,a_1) > Q_2(s_0,a_1) = Q_2(s_2,a_2)$. The current policy is $\pi(s_0) = a_2, \pi(s_1) = a_1, \pi(s_2) = a_1$. We can claim that for all $n \geq 2$, the policy is fixed as above, namely, the max operator in $\nabla_\theta L$ can be removed.

Proposition 6.1. In the setting of one layer of linear network and constant reward function, if all $Q(s_i,a_i)$ are initially set as $C > \frac{r}{1-\gamma}$ and the terminal value $Q(s_1,a_i)$ are fixed, after two steps of gradient descent (for all $n \geq 2$), the following holds:

$$Q_n(s_1,a_1) = Q_n(s_1,a_2) > Q_n(s_0,a_2) = Q_n(s_2,a_1) > Q_n(s_0,a_1) = Q_n(s_2,a_2). \tag{6.10}$$

We use induction to finish the proof.

Proof. For $n = 2$, (6.10) holds.

Assume for $n = k$, (6.10) holds. Then

$$\begin{aligned}
L_k = &(Q_k(s_0,a_1) - r - \gamma Q_k(s_0,a_2))^2 + (Q_k(s_0,a_2) - r - \gamma C)^2 \\
&+ (Q_k(s_2,a_1) - r - \gamma C)^2 + (Q_1(s_2,a_2) - r - \gamma Q_k(s_2,a_1))^2.
\end{aligned} \tag{6.11}$$

The gradient for $\theta(a_1), \theta(a_2)$ are

$$\begin{aligned}\nabla_{\theta(a_1)} L_k &= 2\phi(s_0)(Q_k(s_0,a_1) - r - \gamma Q_k(s_0,a_2)) + 2\phi(s_2)(Q_k(s_2,a_1) - r - \gamma C),\\ \nabla_{\theta(a_2)} L_K &= 2\phi(s_1)(Q_k(s_0,a_2) - r - \gamma C) + 2\phi(s_2)(Q_k(s_2,a_2) - r - \gamma Q_k(s_2,a_1)).\end{aligned} \tag{6.12}$$

Then the action value functions of next step are:

$$\begin{aligned}Q_{k+1}(s_0,a_1) &= Q_k(s_0,a_1) - \eta\phi(s_0)^\top \nabla_{\theta(a_1)} L_k\\ &= (1-2\eta)Q_k(s_0,a_1) + 2\eta(r+\gamma Q_k(s_0,a_2)) - 2\eta\alpha(Q_k(s_2,a_1) - r - \gamma C),\\ Q_{k+1}(s_0,a_2) &= Q_k(s_0,a_2) - \eta\phi(s_0)^\top \nabla_{\theta(a_2)} L_k\\ &= (1-2\eta)Q_k(s_0,a_2) + 2\eta(r+\gamma C) - 2\eta\alpha(Q_k(s_2,a_2) - r - \gamma Q_k(s_2,a_1)),\\ Q_{k+1}(s_2,a_1) &= Q_k(s_2,a_1) - \eta\phi(s_2)^\top \nabla_{\theta(a_1)} L_k\\ &= (1-2\eta)Q_k(s_2,a_1) + 2\eta(r+\gamma C) - 2\eta\alpha(Q_k(s_0,a_1) - r - \gamma Q_k(s_0,a_2)),\\ Q_{k+1}(s_2,a_2) &= Q_k(s_2,a_2) - \eta\phi(s_2)^\top \nabla_{\theta(a_2)} L_k\\ &= (1-2\eta)Q_k(s_2,a_2) + 2\eta(r+\gamma Q_k(s_2,a_1)) - 2\eta\alpha(Q_k(s_0,a_2) - r - \gamma C).\end{aligned} \tag{6.13}$$

By inductive assumption, we compare each term of Q_{k+1} and have

$$Q_{k+1}(s_0,a_2) = Q_{k+1}(s_2,a_1), Q_{k+1}(s_0,a_1) = Q_{k+1}(s_2,a_2).$$

Also by inductive assumption,

$$\begin{aligned}&Q_{k+1}(s_0,a_2) - Q_{k+1}(s_0,a_1)\\ &= (1-2\eta)(Q_k(s_0,a_2) - Q_k(s_0,a_1)) + 2\eta\gamma(1-\alpha)(C - Q_k(s_0,a_2))\\ &\quad + 2\eta\alpha(Q_k(s_0,a_2) - Q_k(s_0,a_1))\\ &> 0.\end{aligned} \tag{6.14}$$

Thus the induction assumption holds. □

From the proposition above, we find that the order relations of all $Q(s,a)$ are

deterministic after 2 steps. So we can write a continuous form of gradient descent:

$$\begin{aligned}
\frac{d\theta(a_1)}{dt} &= -\nabla_{\theta(a_1)}L(\theta),\\
\frac{d\theta(a_2)}{dt} &= -\nabla_{\theta(a_2)}L(\theta),\\
\frac{dQ(s_0,a_1)}{dt} &= -\phi(s_0)^\top\nabla_{\theta(a_1)}L(\theta)\\
&= -2(Q(s_0,a_1)-r-\gamma Q(s_0,a_2))-2\alpha(Q(s_2,a_1)-r-\gamma C),\\
\frac{dQ(s_0,a_2)}{dt} &= -\phi(s_0)^\top\nabla_{\theta(a_2)}L(\theta)\\
&= -2(Q(s_0,a_2)-r-\gamma C)-2\alpha(Q(s_2,a_2)-r-\gamma Q(s_2,a_1)),\\
\frac{dQ(s_2,a_1)}{dt} &= -\phi(s_2)^\top\nabla_{\theta(a_1)}L(\theta)\\
&= -2(Q(s_2,a_1)-r-\gamma C)-2\alpha(Q(s_0,a_1)-r-\gamma Q(s_0,a_2)),\\
\frac{dQ(s_2,a_2)}{dt} &= -\phi(s_2)^\top\nabla_{\theta(a_2)}L(\theta)\\
&= -2(Q(s_2,a_2)-r-\gamma Q(s_2,a_1))-2\alpha(Q(s_0,a_2)-r-\gamma C).
\end{aligned} \tag{6.15}$$

We can write this gradient flow as an ODE:

$$\frac{dQ}{dt} = MQ+N,$$

where $Q=[Q(s_0,a_1),Q(s_0,a_2),Q(s_2,a_1),Q(s_2,a_2)]^\top$,

$$M=\begin{pmatrix} -2 & 2\gamma & -2\alpha & 0\\ 0 & -2 & 2\alpha\gamma & -2\alpha\\ -2\alpha & 2\alpha\gamma & -2 & 0\\ 0 & -2\alpha & 2\gamma & -2 \end{pmatrix},$$

$N=[2r+2\alpha r+2\alpha\gamma C, 2r+2\gamma C+2\alpha r, 2r+2\gamma C+2\alpha r, 2r+2\alpha r+2\alpha\gamma C]^\top$.

We set $Q_0=[C-C_1-C_2^1-\alpha C_2^2, C-C_1-\alpha C_2^1-C_2^2, C-C_1-\alpha C_2^1-C_2^2, C-C_1-C_2^1-\alpha C_2^2]^\top$. Then we can solve this ODE system by using fundamental solution matrix.

Theorem 6.2. For a system of homogeneous linear ordinary differential equations

$$\frac{dy}{dx} = Ay, A \in \mathbb{R}^{n\times n},$$

the fundamental solution matrix of this system is $e^{Ax} := \sum_{k=0}^{\infty} \frac{(Ax)^k}{k!}$.

For a system of inhomogeneous linear ordinary differential equations

$$\frac{dy}{dx} = Ay + b, A \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^{n\times 1},$$

denote its fundamental solution matrix as $\Phi(x)$, we can use the method of variation of parameters to solve the general solution:

$$y(x) = \Phi(x)(c + \int_{x_0}^{x} \Phi^{-1}(s)bds), \tag{6.16}$$

where c is the constant related to the initial value $y(x_0)$. Given an initial value $y(x_0) = y_0$, we can compute the particular solution.

We can use the eigenvalues and eigenvectors to solve the fundamental solution matrix of $\frac{dQ}{dt} = MQ + N$. Computed by matlab, the eigenvectors of M

$$D = \begin{pmatrix} 1 & -1 & 1 & -1 \\ \frac{\gamma\alpha+(\alpha(\alpha\gamma^2-4\gamma+4\alpha))^{\frac{1}{2}}}{2(\gamma-\alpha)} & \frac{\gamma\alpha-(\alpha(\alpha\gamma^2+4\gamma+4\alpha))^{\frac{1}{2}}}{2(\gamma+\alpha)} & \frac{\gamma\alpha-(\alpha(\alpha\gamma^2-4\gamma+4\alpha))^{\frac{1}{2}}}{2(\gamma-\alpha)} & \frac{\gamma\alpha+(\alpha(\alpha\gamma^2+4\gamma+4\alpha))^{\frac{1}{2}}}{2(\gamma+\alpha)} \\ \frac{\gamma\alpha+(\alpha(\alpha\gamma^2-4\gamma+4\alpha))^{\frac{1}{2}}}{2(\gamma-\alpha)} & \frac{-\gamma\alpha+(\alpha(\alpha\gamma^2+4\gamma+4\alpha))^{\frac{1}{2}}}{2(\gamma+\alpha)} & \frac{\gamma\alpha-(\alpha(\alpha\gamma^2-4\gamma+4\alpha))^{\frac{1}{2}}}{2(\gamma-\alpha)} & \frac{-\gamma\alpha-(\alpha(\alpha\gamma^2+4\gamma+4\alpha))^{\frac{1}{2}}}{2(\gamma+\alpha)} \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

are linear independent, so the fundamental solution matrix can be computed by

$$\Phi(t) = [e^{\lambda_1 t}d_1, e^{\lambda_2 t}d_2, e^{\lambda_3 t}d_3, e^{\lambda_4 t}d_4], \tag{6.17}$$

where $\{\lambda_i\}_{i=1}^4$ are the eigenvalues of M and $\{d_i\}_{i=1}^4$ are the eigenvectors of M.

6.2 Numerical experiment of the continuous model

In this part we will solve the ODE system above by matlab and compare the solution of this continuous model and the training results of discrete TD algorithm

to verify the accuracy of this continuous model. We set the initial value $Q_0(s_i, a_i) = 1$ and fix terminal value $Q(s_1, a_i) = 1$, set $\phi(i)^\top \phi(j) = \alpha = 0.1022$, discount factor $\gamma = 0.9$, reward $r = -0.05$, learning rate $\eta = 0.01$. The results are as follows:

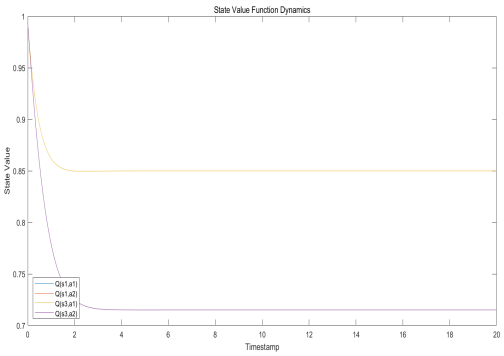
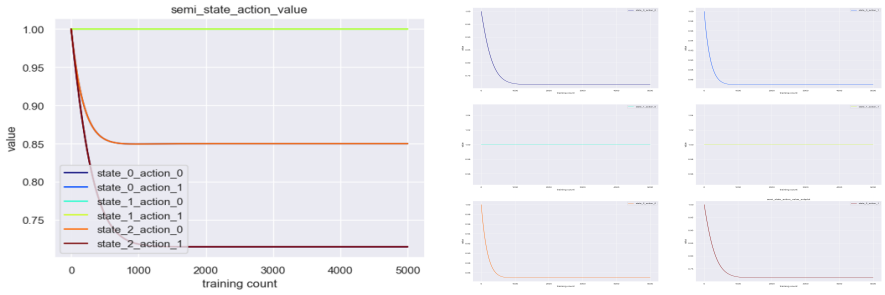


Figure 6-1 This is the result of solving the continuous ODE model by matlab. We can see that $Q(s_0, a_1) = Q(s_2, a_2) < Q(s_0, a_2) = Q(s_2, a_1)$. And the convergence point of the larger value is around 0.85, the convergence point of the smaller value is around 0.72.



(a) All $Q(s,a)$ in one figure (b) One $Q(s,a)$ in one figure

Figure 6-2 This is the result of training the model using TD method. We can see that the convergence point of all state-action-values are nearly the same as the result in the ODE model. From the comparison we can verify the accuracy of the continuous model.

6.3 Summary

In this section we give an analysis of the convergence property of TD in a simple case, explaining that states with the same distance have the same value,

which shows that the equidistant similarity we find is a reasonable observation. Then we use the gradient flow to construct a continuous model in the form of an ODE system and solve it by the method of fundamental solution matrix. Finally we compare the solution of this ODE system and the result of training by TD and verify that this model is an accurate model.

In fact, we can similarly prove that the values of states decrease as the distance increases, by just changing the terminal state from s_1 to s_2. We can also construct a continuous model of an ODE system and compute the convergence point.

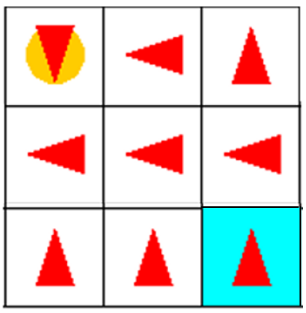
7 Chapter Seven Summary

7.1 Main conclusions

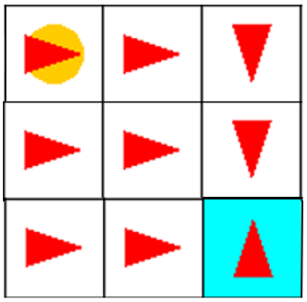
From a series of experiments in different environment settings, we first find a criterion that the values of states are closely related to their distance from the terminal state. The values of states in the same distance are close to each other and the longer the distance is, the values of states in this distance are lower. Then we construct an observation called equidistant similarity κ to verify this criterion in grid worlds of different dimensions and different embeddings. Then we give an analysis of convergence property of TD in a simple case and construct a continuous model to compute its convergence point. Finally we conduct numerical experiments to verify the accuracy of this continuous model.

7.2 Reflection and improvement

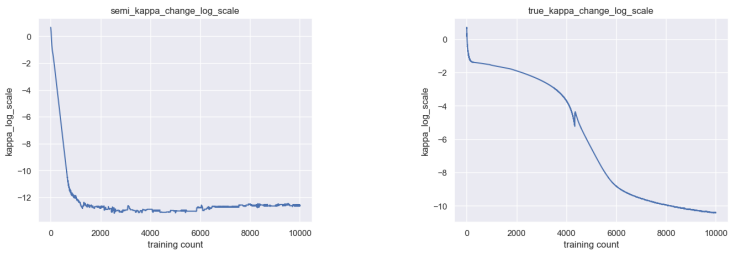
In many other repeat experiments, there are cases where both loss and κ of TD and RG are close, but TD's policy is better than RG. There are also some circumstances that TD's policy is worse than RG but its κ is still smaller than RG's. This means that equidistant similarity is not a perfect indicator of the goodness of policy. The problem is that we do not consider the order relation of state values in different distance. We will modify this κ.



(a) TD's policy



(b) RG's policy



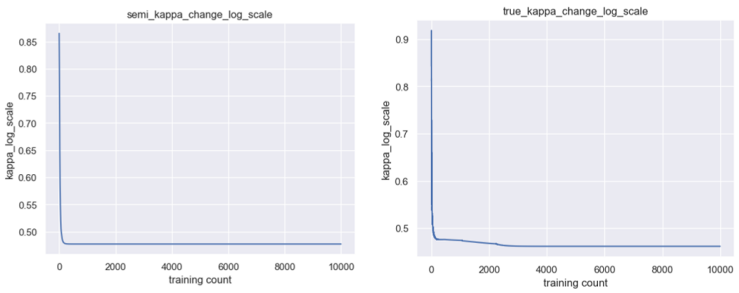
(c) κ_{TD} (log scale) (d) κ_{RG} (log scale)

Figure 7-1 A counter-example for κ's effectiveness

The first thing we want to achieve is to add an order relation term to κ. So we modify the κ as :

$$\kappa_1 = \sum_{i=1}^{D} \mathrm{var}(V_i) + \sum_{i=2}^{D} \exp(\mathbb{E}(V_i) - \mathbb{E}(V_{i-1})). \tag{7.1}$$

This κ has two parts: the similarity term and order relation term. But in this κ the order term takes a much larger weight than the similarity term.



(e) κ_1^{TD} (log scale) (f) κ_1^{RG} (log scale)

Figure 7-2 The magnitude of order relation term is $O(1)$, much larger than similarity term

The balance between the similarity term and the order relation term is a difficult issue. The weights of the two terms might vary as the environment setting changes.

The second thing is that in experiments we notice that the performances of κ and loss are highly similar. Does this mean that the κ is actually an implicit term hidden in loss? We need more theoretical analysis of the relationship between κ and loss.

Third, we hope to give an analysis of convergence property of TD in more complex environments, such as 2-dim grid worlds and non-constant reward function.

References

[1] SILVER D, HUANG A, MADDISON C J, et al. Mastering the game of go with deep neural networks and tree search[J/OL]. Nature, 2016, 529(7587): 484-489. DOI: 10.1038/nature16961.

[2] VINYALS O, BABUSCHKIN I, CZARNECKI W M, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning[J/OL]. Nature, 2019, 575(7782):350-354. https://doi.org/10.1038/s41586-019-1724-z.

[3] KOBER J, BAGNELL J A, PETERS J. Reinforcement learning in robotics: A survey[J/OL]. The International Journal of Robotics Research, 2013, 32 (11):1238-1274. https://doi.org/10.1177/0278364913495721.

[4] PATHAK D, AGRAWAL P, EFROS A A, et al. Curiosity-driven exploration by self-supervised prediction[C/OL]//PRECUP D, TEH Y W. Proceedings of Machine Learning Research: volume 70 Proceedings of the 34th International Conference on Machine Learning. PMLR, 2017: 2778-2787. https://proceedings.mlr.press/v70/pathak17a.html.

[5] HOPFIELD J J. Neural networks and physical systems with emergent collective computational abilities.[J/OL]. Proceedings of the National Academy of Sciences, 1982, 79(8):2554-2558. https://www.pnas.org/doi/abs/10.1073/pnas.79.8.2554.

[6] MINSKY M. Steps toward artificial intelligence[J]. Proceedings of the IRE, 1961, 49(1):8-30.

[7] MICHIE D. Experiments on the mechanization of game-learning part i. characterization of the model and its parameters[J]. The Computer Journal, 1963, 6(3):232-236.

[8] BELLMAN R, KALABA R. Dynamic programming and statistical communication theory[J]. Proceedings of the National Academy of Sciences, 1957, 43(8):749-751.

[9] BELLMAN R. A markovian decision process[J]. Journal of Mathematics and Mechanics, 1957:679-684.

[10] KUNZI H. Dynamic-programming and markov-processes - howard,ra[J]. METRIKA, 1962, 5(3):220-221.

[11] PUTERMAN M L. Markov decision processes[J]. Handbooks in operations research and management science, 1990, 2:331-434.

[12] HINTON G E, SALAKHUTDINOV R R. Reducing the dimensionality of data with neural networks[J]. science, 2006, 313(5786):504-507.

[13] LI Y. Deep reinforcement learning: An overview[J/OL]. CoRR, 2017, abs/1701.07274. http://arxiv.org/abs/1701.07274.

[14] MNIH V, KAVUKCUOGLU K, SILVER D, et al. Human-level control through deep reinforcement learning[J/OL]. Nature, 2015, 518(7540):529-533. https://doi.org/10.1038/nature14236.

[15] MNIH V, BADIA A P, MIRZA M, et al. Asynchronous methods for deep reinforcement learning[C/OL]//BALCAN M F, WEINBERGER K Q. Proceedings of Machine Learning Research: volume 48 Proceedings of The 33rd International Conference on Machine Learning. New York, New York, USA: PMLR, 2016: 1928-1937. https://proceedings.mlr.press/v48/mniha16.html.

[16] SCHULMAN J, LEVINE S, ABBEEL P, et al. Trust region policy optimization[C/OL]//BACH F, BLEI D. Proceedings of Machine Learning Research: volume 37 Proceedings of the 32nd International Conference on Machine Learning. Lille, France: PMLR, 2015: 1889-1897. https://proceedings.mlr.press/v37/schulman15.html.

[17] SCHULMAN J, WOLSKI F, DHARIWAL P, et al. Proximal policy optimization algorithms[J/OL]. CoRR, 2017, abs/1707.06347. http://arxiv.org/abs/1707.06347.

[18] WATKINS C J, DAYAN P. Q-learning[J]. Machine learning, 1992, 8:279-292.

[19] BAIRD L. Residual algorithms: Reinforcement learning with function approximation[M/OL]//PRIEDITIS A, RUSSELL S. Machine Learning Proceedings 1995. San Francisco (CA): Morgan Kaufmann, 1995: 30-37. ht

tps://www.sciencedirect.com/science/article/pii/B978155860377650013X. DOI: https://doi.org/10.1016/B978-1-55860-377-6.50013-X.

[20] SUTTON R S. Learning to predict by the methods of temporal differences [J/OL]. Machine Learning, 1988, 3(1):9-44. https://doi.org/10.1007/BF00115009.

[21] SCHOKNECHT R, MERKE A. Td(0) converges provably faster than the residual gradient algorithm[C/OL]//ICML'03: Proceedings of the Twentieth International Conference on International Conference on Machine Learning. AAAI Press, 2003: 680–687. https://doi.org/10.5555/3041838.3041924.

[22] LI L. A worst-case comparison between temporal difference and residual gradient with linear function approximation[C/OL]//ICML '08: Proceedings of the 25th International Conference on Machine Learning. New York, NY, USA: Association for Computing Machinery, 2008: 560–567. https://doi.org/10.1145/1390156.1390227.

[23] SCHERRER B. Should one compute the temporal difference fix point or minimize the bellman residual? the unified oblique projection view[J]. arXiv preprint arXiv:1011.4362, 2010.

[24] ZHANG S, BOEHMER W, WHITESON S. Deep residual reinforcement learning[J]. arXiv preprint arXiv:1905.01072, 2019.

[25] YIN S, LUO T, LIU P, et al. An experimental comparison between temporal difference and residual gradient with neural network approximation[J]. arXiv preprint arXiv:2205.12770, 2022.

[26] BHANDARI J, RUSSO D, SINGAL R. A finite time analysis of temporal difference learning with linear function approximation[C/OL]//BUBECK S, PERCHET V, RIGOLLET P. Proceedings of Machine Learning Research: volume 75 Proceedings of the 31st Conference On Learning Theory. PMLR, 2018: 1691-1692. https://proceedings.mlr.press/v75/bhandari18a.html.

[27] SUN T, LI D, WANG B. Finite-time analysis of adaptive temporal difference learning with deep neural networks[C/OL]//KOYEJO S, MOHAMED S,

AGARWAL A, et al. Advances in Neural Information Processing Systems: volume 35. Curran Associates, Inc., 2022: 19592-19604. https://proceedings.neurips.cc/paper_files/paper/2022/file/7b9ebf1a1c149960c3452dc94cbd158e-Paper-Conference.pdf.

[28] LEE D. Control theoretic analysis of temporal difference learning[J/OL]. CoRR, 2021, abs/2112.14417. https://arxiv.org/abs/2112.14417.

[29] BOTTOU L, CURTIS F E, NOCEDAL J. Optimization methods for large-scale machine learning[J]. SIAM review, 2018, 60(2):223-311.

[30] DUCHI J, HAZAN E, SINGER Y. Adaptive subgradient methods for online learning and stochastic optimization.[J]. Journal of Machine Learning Research, 2011, 12(7).

[31] TIELEMAN T, HINTON G. Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning[R]. Technical report, 2017.

[32] KINGMA D P, BA J. Adam: A method for stochastic optimization[J]. arXiv preprint arXiv:1412.6980, 2014.

[33] WONHAM W M. Some applications of stochastic differential equations to optimal nonlinear filtering[J/OL]. Journal of the Society for Industrial and Applied Mathematics Series A Control, 1964, 2(3):347-369. https://doi.org/10.1137/0302028.

[34] OKSENDAL B. Stochastic differential equations: An introduction with applications[M/OL]. Second edition. ed. Berlin, Heidelberg: Berlin, Heidelberg: Springer Berlin / Heidelberg, 1989: 10-15. https://doi.org/10.1007/978-3-662-02574-1.

[35] LI Q, TAI C, WEINAN E. Stochastic modified equations and adaptive stochastic gradient algorithms[C/OL]//PRECUP D, TEH Y W. Proceedings of Machine Learning Research: volume 70 Proceedings of the 34th International Conference on Machine Learning. PMLR, 2017: 2101-2110. https://proceedings.mlr.press/v70/li17f.html.

[36] AN J, LU J, YING L. Stochastic modified equations for the asynchronous stochastic gradient descent[J]. Information and Inference: A Journal of the IMA, 2020, 9(4):851-873.

[37] Wikipedia contributors. Neural network — Wikipedia, the free encyclopedia [EB/OL]. 2023. https://en.wikipedia.org/w/index.php?title=Neural_network&oldid=1152977107.

[38] SRIVASTAVA N, HINTON G, KRIZHEVSKY A, et al. Dropout: a simple way to prevent neural networks from overfitting[J]. The Journal of Machine Learning Research, 2014, 15(1):1929-1958.

[39] IOFFE S, SZEGEDY C. Batch normalization: Accelerating deep network training by reducing internal covariate shift[C/OL]//BACH F, BLEI D. Proceedings of Machine Learning Research: volume 37 Proceedings of the 32nd International Conference on Machine Learning. Lille, France: PMLR, 2015: 448-456. https://proceedings.mlr.press/v37/ioffe15.html.

[40] Wikipedia contributors. Reinforcement learning — Wikipedia, the free encyclopedia[EB/OL]. 2023. https://en.wikipedia.org/w/index.php?title=Reinforcement_learning&oldid=1153070423.

[41] SUTTON R S. Reinforcement learning : an introduction[M]. Cambridge, Mass.: Cambridge, Mass. : MIT Press, 1998: 1-7.

[42] FAN J, WANG Z, XIE Y, et al. A theoretical analysis of deep q-learning [C/OL]//BAYEN A M, JADBABAIE A, PAPPAS G, et al. Proceedings of Machine Learning Research: volume 120 Proceedings of the 2nd Conference on Learning for Dynamics and Control. PMLR, 2020: 486-489. https://proceedings.mlr.press/v120/yang20a.html.

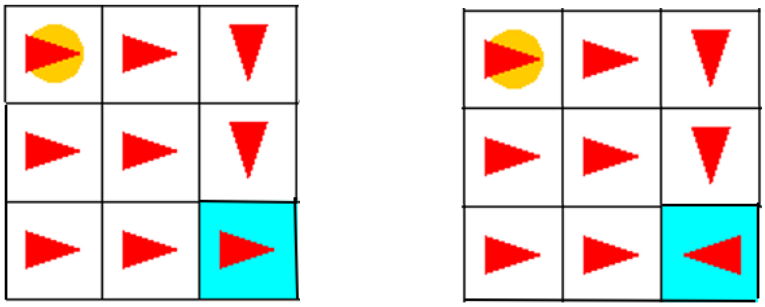
[43] E , Weinan. Principles of multiscale modeling[M]. Cambridge: Cambridge, 2011: 45-50.

Appendix 1
Attempt to train with a modified loss function

Inspired from the observation we difine, we try to train the model using a modified loss function, which is defined as:

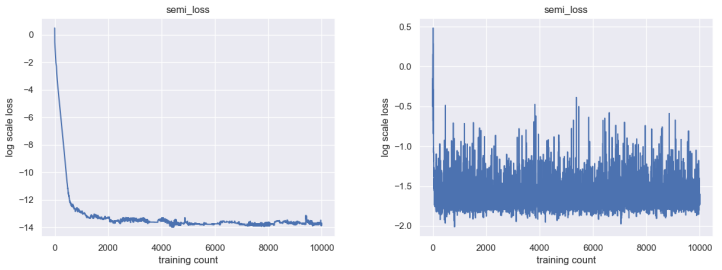
$$L^*(\theta) := \frac{1}{N}\sum_{i=1}^{N}(Q(s_i,a_i) - r - \gamma \max_{a_i'} Q(s_i',a_i'))^2 + \kappa. \tag{A.1}$$

We expect that we can improve the training by using this modified loss function. However, the result of training is not good.

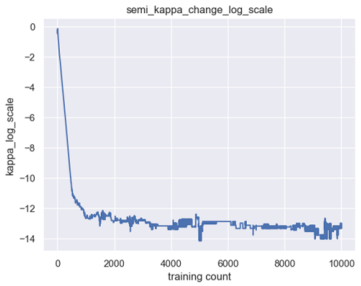


(a) TD's policy (b) TD's policy with L^*

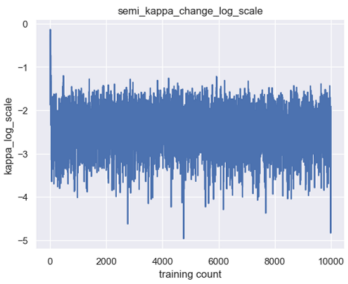
Figure A-1 Training TD with the modified L^*, policy becomes unstable.



(a) TD's loss (log scale) (b) TD's loss with L^* (log scale)

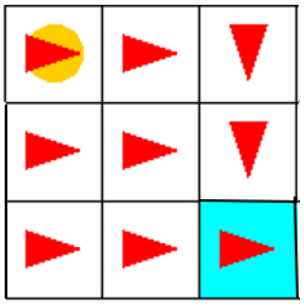


(c) κ_{TD} (log scale)

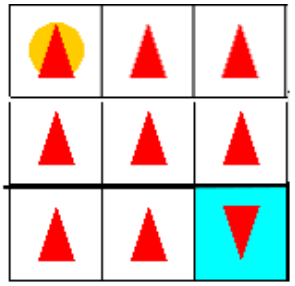


(d) κ_{TD} with L^* (log scale)

Figure A-2 Training TD with L^* even makes the loss and κ larger. In this experiment we use two layers of linear networks with middle size 100. We set training epoch as 10000, learning rate as 0.01. And we use one-hot embedding.

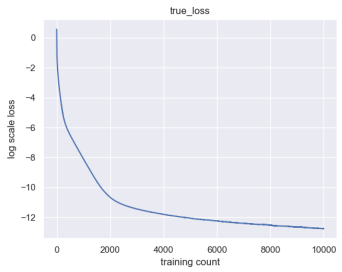


(a) RG's policy

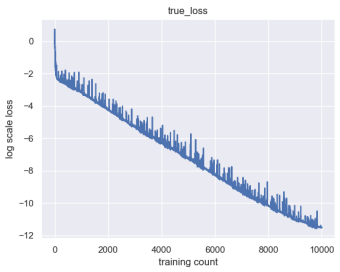


(b) RG's policy with L^*

Figure A-3 Training RG with the modified L^*, policy becomes even worse.

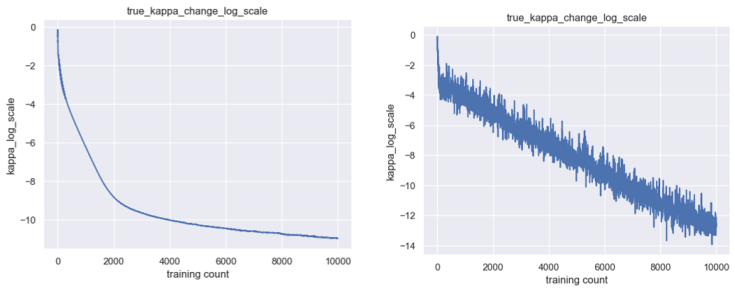


(a) RG's loss (log scale)



(b) RG's loss with L^* (log scale)

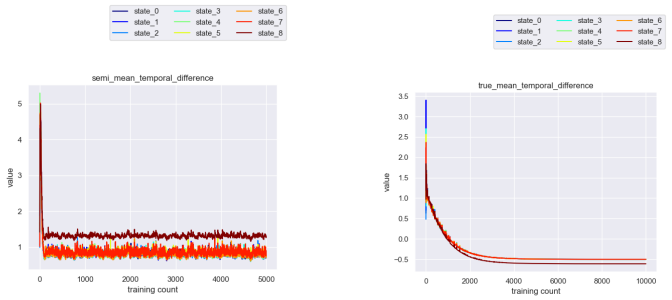
We think the reason of this failure is that κ only takes the variance of values of the same distance into account but neglects the order relation of state values of



(c) κ_{RG} (log scale) (d) κ_{RG} with L^* (log scale)

Figure A-4 Training RG with L^* even makes the loss and κ larger.

different distances. So the result of using this L^* can easily let the values of all states go to the same, which usually implies a bad policy.



(a) Values of TD with L^* (b) Values of RG with L^*

Figure A-5 Training with L^* makes all the non-terminal state values condense.

Appendix 2
Modeling by stochastic differential equation in target network setting

A trick usually used for TD method is the target network. A target network means that we use a θ^* (current estimate of θ) to compute Q_{θ^*}. With training samples $\{s_i, a_i, s_i', r_i\}_{i=1}^N$, we first compute a target

$$Y_i = r_i + \gamma \max_{a'} Q_{\theta^*}(s_i', a), \tag{B.1}$$

and then update θ by the gradient of

$$L(\theta) = \frac{1}{n}(Y_i - Q_\theta(s_i, a_i))^2, \tag{B.2}$$

while θ^* is updated every T steps by letting $\theta^* = \theta$.

Target network is first pproposed by Mnih et al.[14]. Fan et al.[42] give a statistical analysis about the advantage of target network by using bias-variance decomposition. They state that without target network θ^*, the bias term will also depend on θ. Minimizing $L(\theta)$ can be drastically different from minimizing the mean-squared Bellman error (MSBE) $||Q_\theta - TQ_\theta||_\sigma^2$. Target network makes the optimization close to solving $||Q_\theta - TQ_{\theta^*}||_\sigma^2$. When the update steps T is large, minimizing this objective can be viewed as a convex problem.

In this section, we try to give a continuous version of stochastic gradient descent of TD with target network by SDE, and explore the transition time t^* that seperates the descent phase and the fluctuation phase[35]. Then we try to explore

Stochastic gradient descent is often used in optimization problems of the form:

$$\min_{x\in\mathbb{R}^d} f(x) := \frac{1}{n}\sum_{i=1}^n f_i(x), \tag{B.3}$$

where $f, f_i : \mathbb{R}^d \to \mathbb{R}$ for $i = 1, 2, \ldots, n$ and n is the training sample size. For a gradient descent, we compute the full gradient of f:

$$\nabla f(x) = \frac{1}{n}\sum_{i=1}^n \nabla f_i(x).$$

This can be computationally expensive when n is large. So we try to find an alternative gradient, which can be computed easilier. The simplest SGD can be written as

$$x_{k+1} = x_k - \eta \nabla f_{\gamma_k}(x_k), \tag{B.4}$$

where $\{\gamma_k\}$ are i.i.d. uniform viarates taking value in $\{1, 2, \ldots, n\}$, η is the learning rate. In each iteration, SGD only computes the gradient of one f_i so the computation is much smaller than GD.

Stochastic differential equation (SDE) is the continuous version of SGD. First, we rewrite the SGD iteration as

$$x_{k+1} = x_k - \eta \nabla f(x_k) + \sqrt{\eta} V_k, \tag{B.5}$$

where $V_k = \sqrt{\eta}(\nabla f(x_k) - \nabla f_{\gamma_k}(x_k))$ has mean 0 and covariance matrix

$$\eta \Sigma(x_k) = \frac{1}{n} \eta \sum_{i=1}^{n} (\nabla f(x_k) - \nabla f_i(x_k))(\nabla f(x_k) - \nabla f_i(x_k))^\top. \tag{B.6}$$

Now we can write the stochastic differential equation as following:

$$dX_t = b(X_t)dt + \sigma(X_t)dW_t, X_0 = x_0, \tag{B.7}$$

where $b \sim -\nabla f, \sigma \sim (\eta\Sigma)^{\frac{1}{2}}$.

When it comes to the target network setting, we can see the training process in a continuous view. We consider the training process before the first update of the target θ^*. During this period, the loss function can be written as

$$L(\theta, \theta^*) = \frac{1}{n} \sum_{s,a,s',r} (Q_\theta(s,a) - r(s,a) - \gamma \max_{a'} Q_{\theta^*}(s', a'))^2, \tag{B.8}$$

where n is the training sample size. So we can write a stochastic version of gradient descent:

$$\theta_{k+1} = \theta_k - \eta \nabla_\theta L(\theta_k, \theta^*) + \sqrt{\eta} V_k. \tag{B.9}$$

The SDE version of this SGD is

$$d\theta_\tau = -\eta \nabla_\theta L(\theta_\tau, \theta^*) d\tau + \sigma(\theta_\tau) dW_\tau. \tag{B.10}$$

In this expression, $d\tau$ is a extremely small time scale compared to the update time step T ($\tau \ll T$). We now introduce a large time scale t, and assume that $d\tau \ll dt$. The target θ^* updates on this large time scale continuously. Although dt is a small time step in the whole training, it is sufficiently large for θ's update.

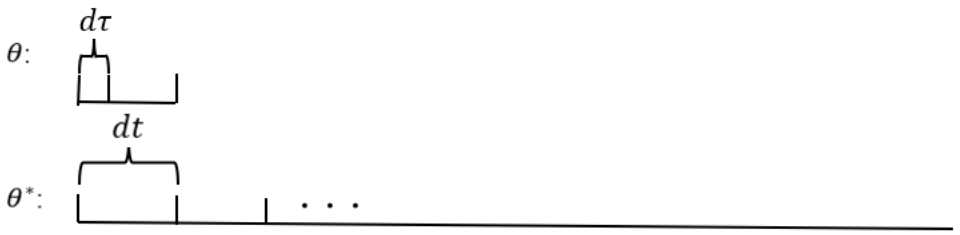


Figure B-1 Illustration of the training process in continuous version. θ updates each $d\tau$ step length, but $d\tau$ is much smaller than dt, the large time scale. We can denote $dt = Md\tau$, where $M \gg 1$

As we consider the update of θ^* as a continuous process, we can write its dynamics:

$$d\theta_t^* = -(\int \nabla_\theta L(\theta(M\eta + t), \theta^*(t)) d\delta(\theta(M\eta + t))) dt.$$

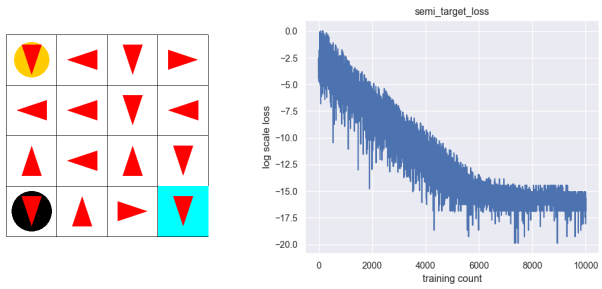
Thus we can write the multiscale problem as follows:

$$\begin{aligned} d\theta_\tau &= -\eta \nabla_\theta L(\theta(\tau), \theta_t^*) d\tau + \sigma(\theta_\tau) dW_\tau, \\ d\theta_t^* &= -(\int \nabla_\theta L(\theta(M\eta + t), \theta^*(t)) d\delta(\theta(M\eta + t))) dt. \end{aligned} \tag{B.11}$$

If we assume the update step T is sufficiently large for $L(\theta(\tau), \theta^*(t))$ to reach the minimum, then we can solve the first equation above. By Principles of Multiscale Modeling[43], we can write $\theta(\tau) = F(\tau, \theta^*(t), W_\tau)$, and $\theta(t + dt) = \lim_{\tau \to \infty} \theta(\tau)$ because T is sufficiently large w.r.t. τ. Then we can solve the second equation by substitute the $\theta(t + dt)$ into this equation.

However, solving this can be difficult, since there exists a max operator in

the first equation. Although we can use Smooth Max technique to eliminate the max operator, the second equation will become a non-linear SDE, which is hard to solve. Here we only give some experimental results of this multiscale model.

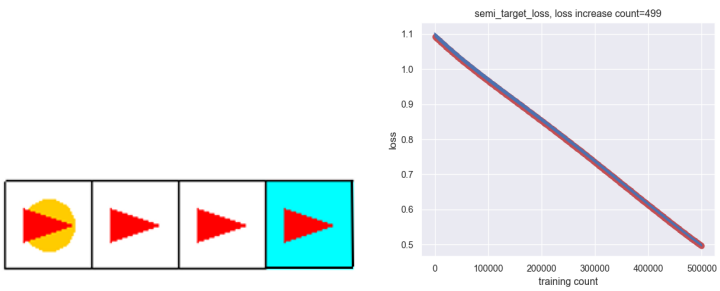


(a) Environment setting (b) Target loss (log scale)

Figure B-2 We use two layers of linear network with a ReLU in the middle to train the model with a fixed target in a 4×4 grid world. The width of the linear network is 1000, training count is 10000, learning rate is 10^{-4}, discounting factor $\gamma=0.95$, the one-step reward is -0.01, the trap penalty is -1.

In this loss figure we see that there are two phases in the training process. In the descent phase, the loss mainly shows the dynamics of descent, which is dominating. After about 6000 time steps, the loss mainly fluctuates rather than descends. The point at about 6000 time steps is the seperation between two phases.

Next we observe the learning process in the case where η is much smaller than update step T and T is much smaller than the total training count.



(a) Environment setting (b) Target loss (log scale)

Figure B-3 We use one linear network to train the model with a changing target in a 1×4 grid world. The training count is 500000, update step T is 1000, learning rate is 10^{-5}, discounting factor $\gamma = 0.95$, the one-step reward is -1.

We can see that when the total time step is much larger than the update step T, and T is sufficiently large for θ to converge, the dynamics of training is as above.

Acknowledgements

I would like to express my gratitude to all those who helped me during the writing of this thesis.

First I want to express my deepest gratitudes to Professor Luo, my supervisor, for his encouragement and guidance to me. When I have no ideas about what to do, he always comes up with new ideas and encourage me to achieve them. And he has a distinctive learning spirit. During his guidance for me, he always emphasizes that we should do some original research, rather than just following others. His words have a deep influence on me and encourage me to insist on thinking independently.

Second I want to express my heartfelt thanks to Shuyu Yin, my senior collaborator. During our collaboration, he gives me a lot of guidance on how to write the code and how to conduct the numerical experiments. His work also inspires me to make progress on my own research. Chatting with him also relaxes my mind and enriched my thought.

Finally, I want to express my thanks to all the classmates who have helped me on my work, especially Pengxiao Lin. I have some discussions with him and I gain a few interesting ideas. We also share our thoughts and encourage each other. Thanks to him, I have more confidence to continue on my research.