Eindhoven University of Technology

MASTER

Deep reinforcement learning

case study with standard RL testing domains

Wang, X.

Award date:
2016

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Web Engineering Research Group

Philips Research

Deep Reinforcement Learning

Case Study with Standard RL Testing Domains

Xu Wang

Supervisors:
Mykola Pechenizkiy (TU/e)
Vlado Menkovski (Philips Research)

Eindhoven, 21-03- 2016

Abstract

Reinforcement Learning (RL) is a type of Machine Learning algorithms and it enables the agent to determine the ideal behavior from its own experience. From robot control to autonomous navigation, Reinforcement Learning algorithms have been applied to address increasing difficult problems. In recent studies, a number of papers have shown great success of RL in the field of production control, finance, scheduling, communication and auto vehicle control. However, in most cases, the performance of these algorithms heavily rely on the quality of the handcrafted features. This drawback limits the application scope of traditional Reinforcement Learning algorithms, since some problems have high dimensional state space and are difficult to hand-engineered. For instance, it is a long standing challenge for traditional RL algorithms to process high dimensional sensory data like vision and voice.

In 2006, the Deep Learning (DL) algorithms were established and have been further developed in recent years. The Convolutional Neural Network is one of the Deep Learning models that could extract high dimensional features direct from the raw pixels, and have been successfully applied in computer vision. It is nature for us to think whether the traditional Reinforcement Learning algorithms could benefit from it.

In 2015, Google proposed a novel algorithm called Deep Q-network (DQN) that could learn behavior direct from raw pixels of images. As we known, there are stability issues when we use a non-linear approximator function, such as a neural network, in a RL algorithm. There are three major reason: first, the inputs are correlated, but a neural network normally requires Independent Identically Distributed (IID) data; second, the policy may oscillate with a neural net; third, the Q-learning gradients can be large and unstable when we back propagate the network. By experience replay, freezing the target Q-network and reward normalization, the DQN algorithm address these problem and can learn how to behave in diverse environments with no adjustment of the architecture and the corresponding parameters. With experiments in Atari 2600 games, they offer evidence that the DQN algorithm surpasses all previous RL algorithms and achieve a level comparable to a professional human game tester across a set of 49 games.

In this thesis, we apply the Deep Q-network on standard RL testing domains, such as Grid World, Mountain Car Problem, and Inverted Pendulum Problem. In contrast with Atari games, these domains have low dimensional state space and have been well-addressed by other RL algorithms. We would like to see the performance of DQN in theses domains compare to other RL algorithms like traditional Q-learning and SARSA. To conduct the experiments, we implement a new agent for traditional RL testing domains which could return the frame, the reward value, and the terminal signal at each time step. Form the results, we demonstrate the DQN algorithm could learn the policy in these environments with the same architecture. It is worth noting that the DQN shows no better result compared to other RL algorithms in these simple domains, since it requires more time resource for complex computing.

Keywords: Deep Q-Network, Reinforcement Learning, Convolutional Neural Network

Acknowledgements

I would like to thank Mykola Pechenizkiy, my supervisor, for guiding me through the thesis project. He offered great help in the preliminary work that were very useful for me to understand the algorithms. I am also thankful to Vlado Menkovski, who guided me at Philips Research. Thanks for his suggestions and valuable feedback on my research. Finally, I want to express my gratitude to my parents who provide constant support throughout my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

If we think in terms of evolutionary history, all animals exhibit some kind of behavior: they do something in response to the inputs that they receive from the environment they exist in. Some animals change the way they behave over time: given the same input, they may response differently later on than they did earlier. Such learning behavior is vital to the survival of species. As technology develops, how to enable machines to mimic the learning ability is one of the long-standing challenges for scientists. Reinforcement learning (RL)[30] [1] is such an area of machine learning inspired by behaviorist psychology, which has revolutionized our understanding of learning in the brain in the last 20 years.

Reinforcement learning is a type of Machine Learning and thereby also a branch of Artificial Intelligence. It enables the machine and software agents to automatically determine the ideal behavior within a specific context [2], in order to maximize its performance. Based on the algorithm, a reinforcement learning agent learns from the consequences of its actions, rather than from being explicitly taught. It selects actions on basis of its past experiences and also by new choices, which is essentially trial and error learning. The automated learning scheme implies that there is little need for a human expert who knows about the domain of application. Much less time will be spent designing a solution, since there is no need for complex sets of rules as with Expert Systems. In spite of remarkable achievements of RL in specific domains, it is still a challenge for traditional RL algorithms to process high-dimensional inputs like vision and speech [24]. Most successful RL applications that operate on these domains have relied on hand-crafted features combined with linear value function or policy [18] representations. In 2006, new techniques so-called deep neural networks [14] [20] [19] were discovered and have been further developed in recent years. The deep neural networks extract high level features directly from raw sensory inputs and have achieved outstanding performance on many important problems in computer vision [29] [23], speech recognition and natural language processing. Apparently, it seems natural to ask whether similar techniques could also be beneficial for RL with sensory data [24].

Nevertheless, there are some stability issues with deep reinforcement learning. The nave reinforcement learning like Q-learning [32] oscillates with neural nets. Initially, data is sequential in reinforcement learning. The successive samples are correlated and do not follow the Independent Identical Distribution (IID) which is normally required for training the neural networks. In addition, policy changes rapidly with slight changes to Q-values. So it implies policy may oscillate and the distribution of data can swing from one extreme to another. Furthermore, the scale of rewards and Q-values is unknown. The nave Q-learning gradients can be large and unstable when we back propagate the neural nets.

In 2015, the Google Deepmind group published the paper Human-level control through deep reinforcement learning [25] in Nature, which leads to a breakthrough in this area. They demonstrate a deep Q-network that could overcome these challenges and learn control policy directly from raw pixels data and game scores. Their approach has been applied on multiple Atari 2600 games. With experiments they offer the evidence that the new network was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human game tester across a set of 49 games. In this thesis, we apply the deep Q-network algorithm on a few standard RL testing domains, e.g. the Grid-World, Mountain Car problem and the Inverted Pendulum problem. These domains have low dimensional state space and their features are easily handcrafted. In fact, these domains have been well-addressed by many reinforcement learning approaches, such as Q-learning and SARSA. We would like to demonstrate how well the deep Q-network performs compared to other reinforcement learning approaches.

1.2 Thesis objective and methodology

The objective of the thesis is:

We apply the deep Q-network to a range of standard RL domains such as Grid Word Navigation, Mountain Car problem and the Inverted Pendulum problem. Our goal is to create a single agent that is able to successfully learn to solve these challenges and maximize the future rewards with high-dimensional sensory inputs.

In order to accomplish this objective, we conduct a study of existing approaches and compare their pros and cons. Initially, the Markov Decision Process provides a standard formalism of describing the decision making situation in the environment. it indicates the Markov property that the next state is determined by the current state and action and is independent of all previous states and actions. The Markov Decision Process could be solved by value interaction algorithm but it requires we have the predefined knowledge of transition function and reward function. The Q-learning algorithm does not essentially know how the world works and can directly learn the policy through its own experience. However, the performance of Q-learning is still relied on the quality of handcraft features. This drawback results in a limited application scope, which always require low-dimensional state space and easily hand-engineered features. Now, the convolutional network [21] leads a breakthrough in the computer vision which could extract the high level features from raw high-dimensional inputs. This brings us the idea that it could be benefit if we could merge the two different techniques.

In 2015, Google demonstrates the novel algorithm called deep Q-network. This method connects the traditional Q-learning algorithms with a convolutional neural net that could learn how to behave directly from the raw sensory input. That is exactly the algorithm we need to fulfill our goal. Google has proved that the deep Q-network is capable of processing various complex tasks whose features are hard hand-engineered. On the contrary, we will apply this algorithm to standard RL domains which have low dimensional space state. We would like to compare the results and performance of Q-learning with other RL methods.

1.3 Main Results

We implement the agent, the Deep Q-Network, with Torch using the scripting language Lua on Ubuntu platform. The whole network consists of 3 convolutional layers and 2 fully-connected layers. During the training procedure, the DQN agent processes 4 recent frames at each time and separates output units for each possible action. By implementing experience replay, reward normalization and periodically freezing the target network, the DQN agent we create could address the stability issues that normally occur when we use a non-linear approximator function in the

Reinforcement Learning algorithms.

To conduct the experiments, we construct a new environment for standard RL domains, such as Grid Word Navigation, Mountain Car problem and the Inverted Pendulum problem. The environment implementation is based on an open source Java library BURLAP. Since there exist languages barriers between Java and Lua, we use socket programming by TCP/IP protocols. When receiving an action, the environment is able to return the reward, terminal signal and the pixels of frames. From the experiment, we demonstrate that the DQN agent could process 3 diverse tasks with same architecture without adjustments of the parameters.

In the paper of Google, they show that the DQN has achieved outstanding results in Atari games. These problems normally have high dimensional state space and are difficult to hand-engineered. In our experiments, on the contrary, the standard RL domains we use have low dimensional state space and have already been well-addressed by other RL algorithms. Thus, we compare the performance with traditional Q-learning algorithm and SARSA algorithm. From the results, SARSA algorithm achieved the highest average rewards within 100 episodes in the three domains. We show that the DQN is more suitable for complex domains whose features cannot be easily handcrafted. It is due to that DQN agent always requires much more time and resource for processing the images.

1.4 Thesis structure

The structure of this thesis is organized as follows. Section 2 conducts a study of Markov Decision Process, Q-learning and Convolutional neural networks and explains the pros and cons in the meanwhile. Section 3 formally defines the problem we will address and give a brief introduction of the standard RL testing domains. In Section 4, we have a detailed explanation of the deep Q-network. Section 5 shows the case study and the corresponding results. In Section 6, we draw a conclusion of the work. In the appendix, we provide a review on Deep Learning implementation frameworks.

Chapter 2

Background

Before starting a detailed discussion on the deep Q-network, this chapter provides a short background information on key concepts of Reinforcement Learning and Deep Convolutional Neural Networks. It is by no means an exhaustive tutorial for these algorithms, but we believe it is essential to understand these main concepts before we introduce the deep Q-networks. Furthermore, we provide the pros and cons of these algorithms and explain the rationality of combining Q-learning with a deep network, which leads to the new algorithm of Google.

2.1 Markov Decision Process (MDP) & Value Iteration Algorithm

Markov Decision Process [3] offers a standard formalism for describing multi-state decision making in probabilistic environment. More precisely, a Markov Decision Process is a discrete time stochastic control process. At each time step, the process is in some state s, and the decision maker may choose any action a that is available in state s. The process responds at the next time step by randomly moving into a new state s' and giving the decision maker a corresponding reward $R(s, a, s')$.

The probability that the process moves into its new state s' is influenced by the chosen action. In math, it is given by the state transition function $T(s'|s, a)$. The state transitions of a Markov Decision Process satisfy the Markov Property which implies that, given the current state s and an action a, the next state s' is conditionally independent of all previous states and actions.

The core problem of MDPs is to find a policy for the decision maker: a function π that specifies the action $\pi(s)$ that the decision maker will choose when in state s. The goal of MDP is to choose a policy π that will maximize some cumulative function of the random rewards. In math:

$$\sum_{t=0}^{\infty} \gamma_t R(s_t, a_t, s_{t+1})$$

Where t indicates the time step and γ is a discount factor between 0 and 1 that affects how much immediate rewards are referred to later ones.

The Markov Decision Process can be solved by Value Iteration (VI) which is an algorithm that finds the optimal value function (the expected discounted future reward of being in a state and behaving optimally from it), and consequentially the optimal policy. The central idea of Value Iteration algorithm is the Bellman EquationM, which states that the optimal value of a state is the value of the action with the maximum expected discounted future return (the action with
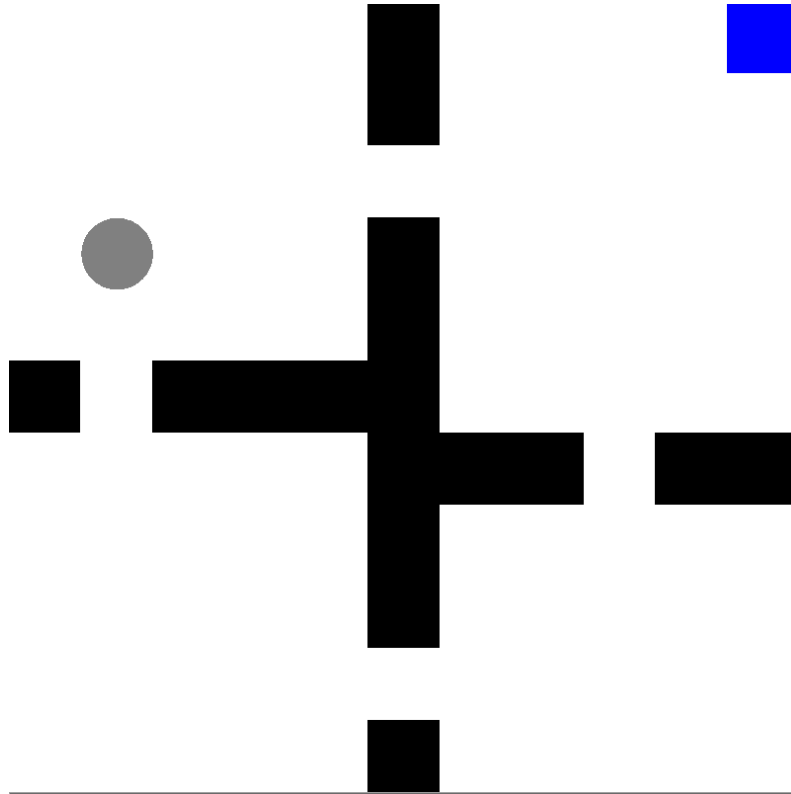
Figure 2.1: **An Example of the Grid World.** A grid world is a 2D world in which an agent can move north, south, east or west by one unit. In the image, the agent postion is represented by a gray circle and the walls of the world are painted black. The goal in a grid world is for the agent to navigate to the goal location which has been depicted in blue rectangle on the top right corner.

maximum Q-value). And the Q-value for a state-action pair is defined as the expected value over all possible state transitions of the immediate reward summed with the discounted value of the resulting state. The formula is shown below:

$$V(s) = \max_a Q(s, a)$$

$$Q(s, a) = \sum_{s'} T(s'|s, a)[R(s, a, s') + \gamma V(s')]$$

A typical example that Value Iteration can solve (to a certain extent) is the Grid World problem. a grid world is a 2D world in which an agent can move north, south, east or west by one unit, provided there are no walls in the way. Figure 2.1 below shows a simple grid world with an agents position represented by a gray circle and the walls of the world painted black. The goal in a grid world is for the agent to navigate to the goal location which has been depicted in blue rectangle on the top-right corner.

In the case of Value Iteration, Bellman updates are performed in entire sweeps of the state space. That is, at the start, the value of all states is initialized to some arbitrary value. Then, the Bellman Equation updates the value function [28] estimate sweeping over the entire state space. These steps are repeated for some fixed number of iterations or when the maximum change in the
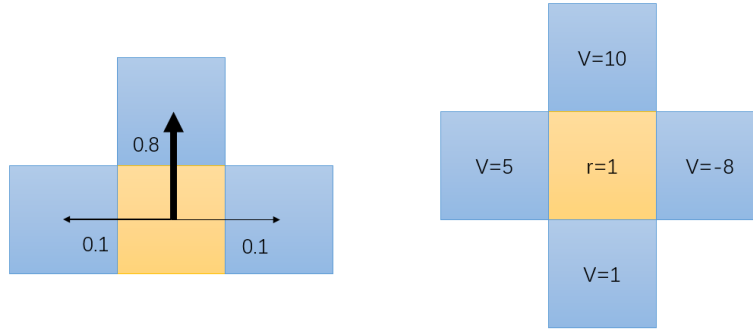
Figure 2.2: Value Iteration Example

value function is small. The pseudocode of VI is shown below.

Algorithm 1 Value Iteration [3]

1: Initialize value function $V(s)$ arbitrarily for all state s.
2: Repeat until convergence
3: **for** each state s **do**
4:     $V(s) := \max_a \sum_{s'} T(s'|s,a)[R(s,a,s') + \gamma V(s')]$
  end for

Example: Suppose the agent will move in the intended direction with probability 0.8 and the agent will move in the adjacent direction with probability 0.1. We assume the agent is currently located in the yellow cell and the discounted factor is set to 0.9. For convenience, the direct reward function is set to the constant value 1. And the current value function in other cells have been shown in Figure 2.2. The value function of the yellow cell at the next timestep is calculated as follows:

$$
\begin{aligned}
V(s) &:= \max_a \sum_{s'} T(s'|s,a)[R(s,a,s') + \gamma V(s')] \\
&= reward + \max_a \sum_{s'} T(s'|s,a)\gamma V(s') \\
&= 1 + \max\{ \\
&\quad 0.1*0.9*1 + 0.8*0.9*5 + 0.1*0.9*10 \quad (left), \\
&\quad 0.1*0.9*5 + 0.8*0.9*10 + 0.1*0.9*-8 \quad (up), \\
&\quad 0.1*0.9*10 + 0.8*0.9*-8 + 0.1*0.9*1 \quad (right), \\
&\quad 0.1*0.9*-8 + 0.8*0.9*1 + 0.1*0.9*5 \quad (down)\} \\
&= 1 + \max 4.59(left), 6.93(up), -4.77(right), 0.45(down) \\
&= 1 + 6.93(up) \\
&= 7.93
\end{aligned}
$$

Therefore, the new value for the yellow cell is 7.93.

To have a better understanding of Value Iteration algorithm, we visualize the entire estimated value function along with the corresponding policy in Figure 2.3. In this case, the reward function always returns -1 for every state-action-state transition. A fairly large value of 0.99 is set to the discount factor parameter. The value function assigns a value to each state that represents the expected future discounted reward when following the optimal policy from the state, which is depicted by arrow glyphs.
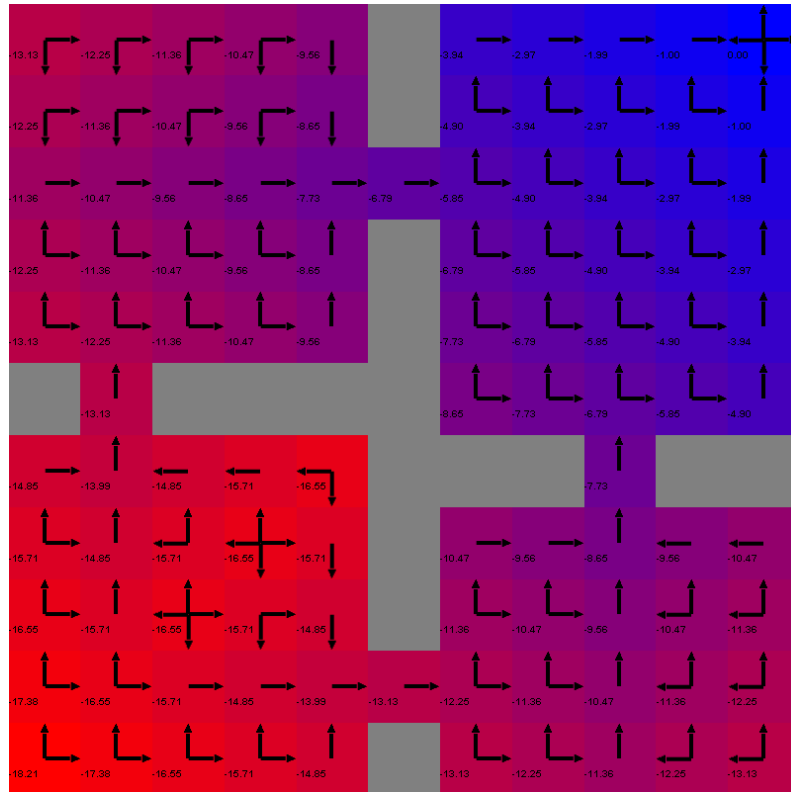

Figure 2.3: **Value iteration & policy visualization for grid worlds.**The value of a cell is rendered in the grid world with a color that blends from red (minimum value) to blue (the maximum value). In this case, the direct reward function always returns -1 for every state-action-state transition before the agent reaching the goal position. The discount factor is set to 0.99. We visualize the estimated value function along with the corresponding policy in this figure. The value function assigns the discounted reward when following the optimal policy from the state that is depicted by arrow glyphs.

The Value Iteration is as a planning algorithm that makes use of the Bellman Equation to estimate the Value function. However, if the probabilities or reward function is unknown, which is common in real systems, Value Iteration algorithms can no longer be computed. The root cause of this problem is that the planning algorithm need the access to a model of the world or at least a simulator. The other drawback of VI is that when state space is large or infinite, which may exceed the capability of modern computer.

2.2 Q-Learning

Unlike a planning algorithm, a learning algorithm like Q-learning [4] involves determining behavior when the agent does not know how the world works and can learn how to behave from direct experience with the world. Figure2.4 illustrates a typlical example of how the agent interacts with the environment. As the name suggested, Q-learning estimates of the optimal Q-values of an MDP, which means that behavior can be learned by taking actions greedily with respect to the learned Q-values. In Q-learning algorithm, the most common way to choose an action in the current world state(s) is to use greedy policy. ϵ is a fraction between 0 and 1. Based on the policy, the agent randomly selects among all action a fraction of time, whereas the action with respect
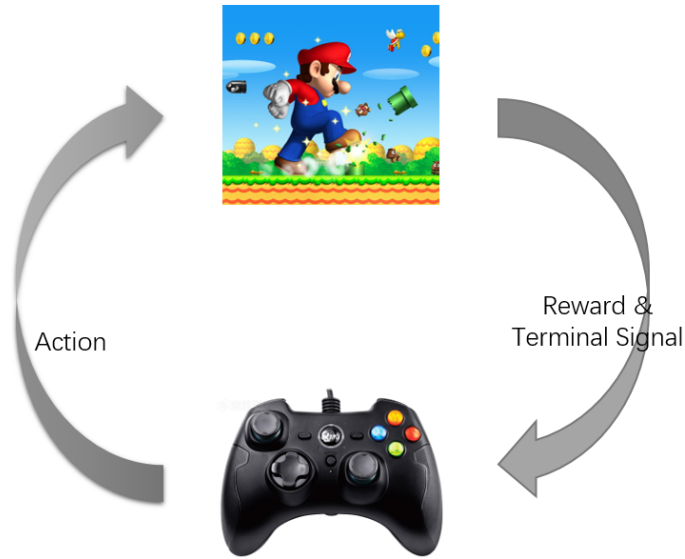
Figure 2.4: Q-Learning Example [5]. Unlike the planning algorithm, the Learning agent has no predefined knowledge of the envrionemnt, which means the reward function and the transition function are unknown. Instead, the agent learns how to behave by interacting with enviornment.

to the Q-value estimates a fraction of $(1 - \epsilon)$ time. The update rule for Q-learning is below.

$$Q(s, a) := Q(s, a) + \alpha[R(s, a, s') + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

The Q-value is updated by the Q-value of the last state-action pair (s, a) with respect to the observed outcome state s' and direct reward $R(s, a, s')$. The parameter between 0 and 1 stands for the learning rate.

The difference of update rules between Value Iteration and Q-learning algorithm is that the Q-value of a state in VI is the maximum Q-value which is the expected sum of reward and discounted value of the next state, whereas the Q-value of Q-learning algorithm is the sum of rewards and discounted max Q-value of the observed next state, which implies that we only use the states and rewards we happen to get by interacting with the environment. As long as we keep trying random actions on the same state, we could reach all possible states of next. After multiple times of aggregation, we should finally move close to the true Q-value. In order to have guaranteed convergence, some tips for the parameter setting could be very useful in practice. Firstly, the greedy policy should anneal linearly from 1.0 to a small fraction, for instance 0.1, over a certain training steps, and fixed at the small fraction thereafter. This setting enables the agent to explore more action-state pairs at the beginning of the training, and reduce the randomization when the agent gains more experience. The other trick is slowly decreasing the learning rate α over time. The Q-learning algorithm can be summarized in the following pseudocode.

As stated above, the basic idea of Q-learning is to estimate the action-value function by using Bellman Equation as an iterative update. In that case, the value function converges to optimal the value function as the iteration i tends to infinity. However, it is impractical since action-value function is estimates separately for each sequence without any generalization. Instead, it is common to use a function approximator to estimate the action-value function. The other problem is that traditional reinforcement learning algorithms heavily rely on the quality of hand-crafted feature representations, which limits the application scope of these algorithms. There is no double that we could benefit more if features can be directly extracted from raw high-dimensional sensory inputs, for instance, the human-like visual and auditory information. In recent studies,

---

Algorithm 2 Q-Learning

1: Initialize Q-values $Q(s, a)$ arbitrarily for all state-action pairs.
2: For life or until the learning is terminated
3:      Choose an action a in the current world state s based on current Q-value estimates $Q(s, \cdot)$
4:      Take an action a and observe the outcome state s' and reward $R(s, a, s')$
5:      Update $Q(s, a) := Q(s, a) + \alpha[R(s, a, s') + \gamma \max_{a'} Q(s', a') - Q(s, a)]$

---

deep learning models are effective feature extractors over high-dimensional data, which will be discussed in detail in next section.

2.3 Deep Convolutional Neural Networks

Convolutional neural networks are biologically-inspired variants of multilayer perceptrons [16] which are widely used models for image and video recognition. Like most every other neural networks, they are trained with a version of the backpropagation algorithm. Each hidden layer consists of a set of neurons that have learnable weights and biases and each neuron receives some inputs performs a dot product and optionally follows it with a no-linearity. The main difference between Convolutional neural nets and other multilayer networks is in the architecture. Convolutional neural networks make explicit assumption that the inputs are images, which allows scientists to encode certain properties into the architecture. It enables the forward function more efficient to implement and reduce the amount of parameters in the network.

Figure 2.5 depicts the general architecture of a convolutional neural network. Commonly, there are five types of layers in Convolutional Neural Nets: **Input Layer, Convolutional Layer, Rectified Linear Units Layer, Pooling Layer and Fully-Connected Layer**. We will describe individual layers and their connections in detail.

2.3.1 Input Layer

The first layer takes raw pixels of the observed image as inputs. Normally, an image is described as a structure of $W * H * C$, where W, H are the width and height of the image, and C denotes the number of channels, for instance an RGB image has 3 channels.

2.3.2 Convolutional Layer

The convolutional layer and the pooling layer are the heart of convolutional neural network models. In fact, the architecture of a convolutional neural network is designed to take advantage of the
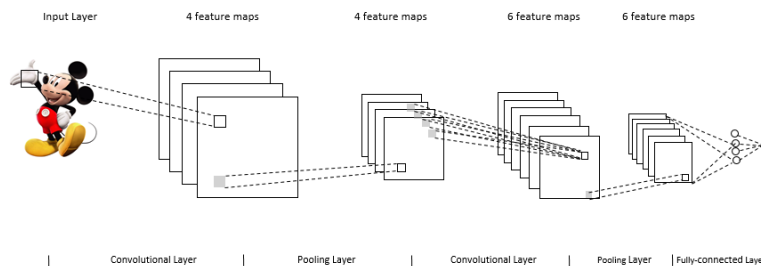

Figure 2.5: The Architecture of convolutional Neural Networks

---

structure of an image and capture its invariant features. This is achieved by the dot production between local region and weights followed by a certain pooling method. Now, we will explain the major terms first.

Spatially-local correlation: When dealing with the high dimensional input, the main challenge is to improve the learning efficiency. Apparently, it is impractical to build a traditional full-connected network which results in too many parameters to be trained. Instead, in the convolutional layer, each neuron is connected to a local region in the input volume, called a **receptive field**. And the whole architecture represents a spatial-local correlation which means the correlation only exist between neurons of adjacent layers. This structure ensures that the learned weights are only responsible for a spatially local input pattern. For instance, suppose we have an input volume with the size of 32*32*3. If the size of the receptive filed is 5*5, then each unit in the convolutional layer refers to weights of a region 5*5*3, which denotes 75 weights to be trained.

Feature Map: A feature map is obtained by repeated application of a function across subregions of the entire image, in other words, by convolutional of the input image with a linear filter, adding a bias term and then apply a non-linear function. It is important to note that the units in the specific feature map share the same weight vector and bias. The weight sharing policy tremendously facilitates the learning efficiency by reducing the amount of learning parameters. The formula below describes a typical function that could detect the feature map.

$$h_{ij}^k = \tanh((W^k * x)_{ij} + b_k)$$

Where h_{ij}^k stands for a unit in the k-th feature map of the hidden layer h. The corresponding filter of this feature map is determined by the weights matrix W^k and the bias b_k. And tanh denotes the non-linearity function. Normally, a convolutional neural net may have multiple feature maps in the same layer to achieve rich representations of data, for instance one map may detects curly loops for some letters and another detects straight lines.

Besides of the filter size, there are other parameters that determine the size of output volume. By default, a convolutional will use **stride 1** to sweep over the input volume, which results in overlapping receptive fields between adjacent neurons in the same feature map and large size of output volume. Therefore, it is a common way to specify a larger stride number when dealing with large size image. Another technique that could influence the output size is called **Zero-padding**. As the name suggested, we pad zeros on the border of the input volume to regulate the size of output volume. It provides a way to preserve size of the input volume. The process of the convolutional layer could be summarized in the following pseudocode.

Algorithm 3 Convolutional Layer

1: Receive the input volume with the size of $W * H * C$
2: Compute the output neurons that are connected to the receptive fields. The size of the output volume is determined by a quad (K, F, S, ZP), which denotes the number of filters, the filter size, the stride number and the size of zero-padding separately.
3: The size of output volume could be described as $W' * H' * C'$ where:
4:      $W' = (W - F + 2P)/S + 1$
5:      $H' = (H - F + 2P)/S + 1$
6:      $C' = K$
7: Based on the weight sharing policy, each feather map has weights with a size of $F * F * C$. Thus, it will amount to $F * F * C * K$ weights and K bias to be trained in the convolutional layer.

2.3.3 Rectified Linear Units Layer (RELU)

!// Normally, RELU layer implements a fixed elementwise activation function, for instance the $\max(0, x)$. It increases the nonlinear properties of the decision function without affecting the receptive fields. As stated above, it is also possible to use other functions, such as tanh. Nevertheless, it requires more computing time due to the algorithm complexity.

2.3.4 Pooling Layer

The pooling layer is another critical component for the convolutional neural nets. It performs a non-linear down-sampling operation, such as max pooling, average pooling and L2-norm pooling, over the entire feature maps. The most commonly used function is max pooling due to the simplicity and performance. In recent studies, some experiments offer the evidence that the max pooling works better in practice compared to the averaging pooling method. It is common to use 2*2 size of filters with a stride of 2 in the pooling layer. Hence, it applies an max operation over 4 adjacent neurons, which results in 75% size reduction for the upper layers. For easy understanding, we describe the pooling process in pseudocode as the convolutional layer.

Algorithm 4 Pooling Layer

1: Receive the input volume with the size of $W * H * C$
2: Specify the pooling size F and the stride S as we did in the convolutional layer. Then perform a down-sampling operation over the input volume.
3: The size of output volume could be described as $W' * H' * C'$ where:
4:     $W' = (W - F)/S + 1$
5:     $H' = (H - F)/S + 1$
6:     $C' = C$
7: Unlike convolutional layer, zero-padding is not commonly used on pooling layer.
8: It is worth noting that the amount of feature maps remains unchanged.

There are two benefits to the pooling process. First, the max pooling operation reduce the size of parameters and then reduce the computation for the future processes. In addition, it is worth noting that the pooling method provides a smart way to capture transportation invariance and ensure the networks robustness to the position.

As stated above, the convolutional layer and pooling layer are the key components of a convolutional neural network. Figure 2.6 provides an intuitive description of the combined architecture of these two layers. In fact, it is common that this structure is periodically repeated in a convolutional net. The underlying assumption behind a deep learning algorithms is that the observed data is the interaction of factors of different levels, corresponding to different level of abstractions. Precisely, the varying numbers of the combined structure determine the amount of abstraction in a convolutional neural net. Meanwhile, this structure requires little pre-processing and ensures that a convolutional neural net could learn filters that were handcrafted in traditional algorithms.

2.3.5 Fully-connected Layer

The fully connected layer is commonly the last layer in a convolutional neural net. Each neuron in this layer is fully-connected to all neurons in the last pooling layer like other traditional neural nets. This layer computes the final class scores where each class is corresponded to a label in the training dataset. Now we have explained all components we need to implement a convolutional neural net. As can be seen in Figure 3, the pixel values from original input are transformed layer by layer to the final class scores. The whole network could be trained with stochastic gradient descent.
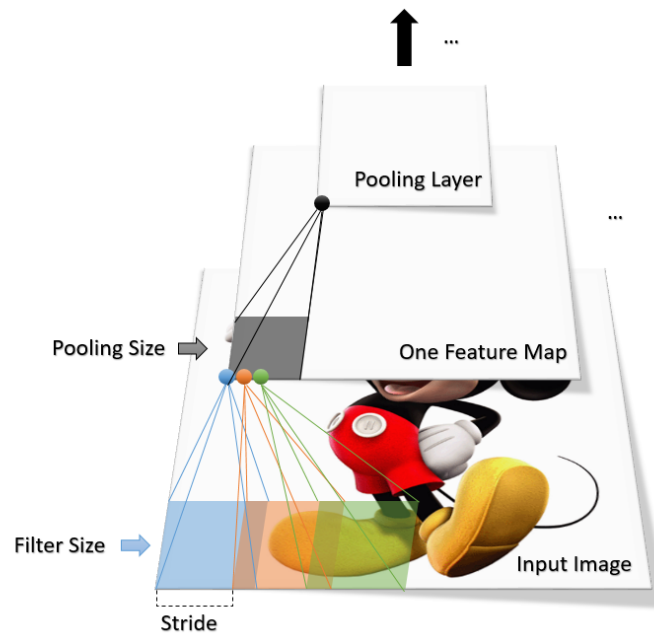
Figure 2.6: The structure of a convolutional layer followed with pooling

The convolutional neural networks have lead a breakthrough in the computer vision that could extract the high level features directly from high-dimensional data. Apparently, if we could combine the strong points of reinforcement learning and deep learning, scientists could free themselves of creating handcraft features and extend the application scope of these algorithms. In 2015, Google published their work in Nature and have a success exploration in this area. We will discuss it in the following sections.

Chapter 3

Problem Formulation

The goal of this paper is to create an agent that could learning control policy that will maximize the rewards or reach the goal positions. Suppose there is an agent that could interact with an environment E by executing a sequence of actions and receive rewards. More precisely, at each time step, the agent selects an action a from the set of legal actions and then transmits it to the environment E. Thereafter, the environment may return the updated reward with raw pixels of the screenshot of the environment. It is worth noting that the interval process of the environment cannot be observed by the agent. In addition, it is also possible that the environment uses a stochastic transition with a certain success rate. Therefore, the agent could only learn how to behave from direct experience with the environment. In this paper, we intend to solve two types of the reinforcement learning problem, the discrete domain and the continuous domain. We will pick the grid world problem, mountain car problem and cart pole problem as illustration.

3.1   Grid World Problem

The grid world problem has been stated in detail when we explained the Markov Decision Process. So it will not be repeated here. The only difference is that the transition function and stochastic rate cannot be observed any more.

3.2   Mountain Car Problem

Mountain car problem is a standard testing domain for traditional reinforcement learning algorithm. Initially, an underpowered car parks in the valley and it intends to drive up along the steep slope of the hill. Since the gravity exists, the car cannot accelerate up to the right most. Instead, it needs to leverage the gravitational potential energy by driving down from the opposite slope before it reaches the goal position of the top right corner. Unlike the grid world problem, the mountain car problem need to learn two continuous variables that are the position and velocity. The agent will continuous receiving the negative rewards before it reaches the final position. As the grid world problem, the agent must learn from its own experience, have no information of the goal position before it first reaches it. Figure 3.1 illustrates a typical screenshot of the mountain car problem.
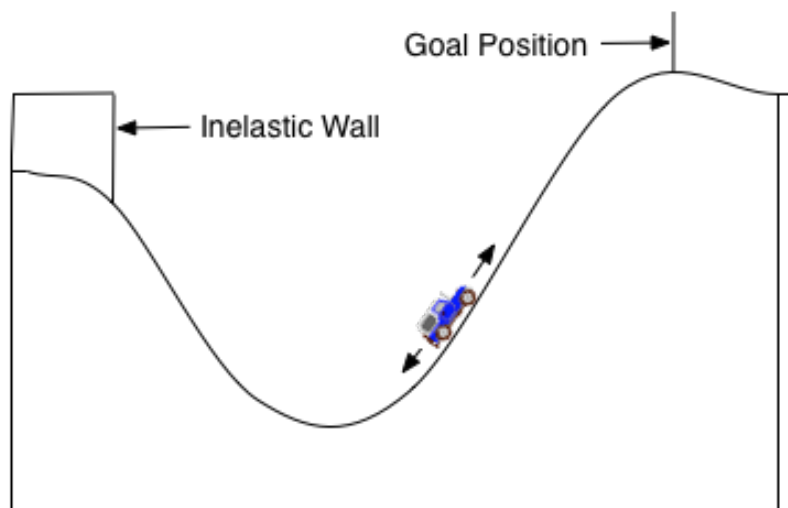
Figure 3.1: A Sketch of the Mountain Car Problem [6]
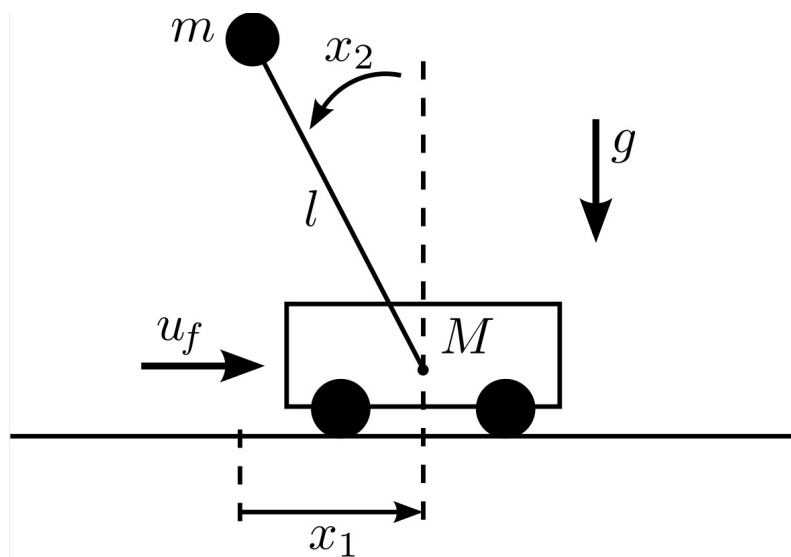
Figure 3.2: A sketch of the inverted pendulum problem [7]

3.3 Inverted Pendulum Problem

The inverted pendulum problem is also a continuous domain problem. The pendulum has its center of mass on its pivot point. Since the gravity exist, the inverted pendulum is unstable and has to move actively to remain balance and keep the mass on its pivot point. The legal actions in this problem are moving left and moving right. Figure 3.2 illustrates a sketch of the inverted pendulum problem.

Chapter 4

Deep Q-Networks

As stated above, a convolutional neural net could extract better representations than handcrafted features. This success motivates the new approach to the reinforcement learning called Deep Q-Networks [12], published by Google deep-mind group. This method connects the traditional Q-learning algorithm to a convolutional neural network that could directly process RGB images. Since less hand-engineered work is needed, it is possible to construct a more general architecture to hand a bunch of similar tasks.

Recall the problem formulation, the three problems have some common properties. The agent interacts with an environment by sending a sequence of actions and receiving the rewards. The main difference from the traditional Q-learning algorithm is that the agent could also return screenshots as well. Certainly, traditional Q-learning provides a starting point, but we need to reformulate some notions.

At each time step t, a_t denotes the action the agent will take and we use x_t, r_t to describe the received screenshot and direct reward. In fact, the current reward value relies on the whole previous sequence of actions. It is not appropriate to simply using current screenshot, which is considered as a partial observation, as the state representation. Instead, the state function is defined as $s_t = x_1, a_1, x_2, , a_{t-1}, x_t$, where s_t is sequence of previous actions and observations. Since the agent is supposed to terminate at goal position, each sequence s_t is considered as a distinct state which satisfies the property of the finite Markov Decision Process. Recall the Bellman Equation, the optimal state-action value function is defined as follows:

$$Q^*(s, a) = E_{s'}[r + \gamma \max_{a'} Q^*(s', a')]$$

Where $Q * (s', a')$ is the optimal state-value of the next time step, is the discount factor that is normally set to 0.99. The optimal state-action value is defined as the sum of direct reward and the maximize of the discounted state-action value of the next step. As stated above, this formula could be solved by value iteration algorithms. We assume that the iterative update converges to the optimal action value function when the iteration tends to infinity. However, it is impractical since the action-value function is estimated separately for each sequence, without any generalization. Instead, it is common to use a function approximator the estimate the action-value function. There are two types of approximator: a linear function or a non-linear function. The linear is widely used in many reinforcement learning problems. Here we choose a non-linear approximator [31], a neural network. The new Q function is defined below, where w are the weights of the neural networks.

$$Q(s, a; w) \approx Q^*(s, a)$$

In order to train the neural networks, we define the objective function by mean-squared error in

Q-values. The loss function changes at each iteration i.

$$L_i(w_i) = E[(r + \gamma \max_{a'} Q(s', a', w_i^-) - Q(s, a, w_i))^2]$$

The optimal target values have been substituted with approximate target values, $y = r + \gamma \max_{a'} Q(s', a', w_i^-)$. Now the Q-learning gradient could be deduced as follows.

$$\frac{\partial L_i(w_i)}{\partial w_i} = E[(r + \gamma \max_{a'} Q(s', a', w_i^-)) \frac{\partial Q(s, a, w_i)}{\partial w_i}]$$

Like other neural networks, the objective could be optimized by stochastic gradient descent by using Q-learning gradient.However, it is generally known that there exist some stability issues of reinforcement learning with a non-linear approximator. In other words, the nave Q-learning algorithm oscillates or diverges with neural nets.

First, the data is sequential and successive samples are correlated and do not follow the independent identical distribution. This issue could be addressed by a mechanism called **experience replay** [27] [22]. At each time step t, we take an action a_t according to ϵ-greedy policy and then store the transition experience $e_t = (s_t, a_t, r_t, s_{t+1})$ in the replay memory $D_t = e_1, e_2, , e_t$. During the training process, a mini-batch of transitions (s, a, r, s') is sampled randomly form the experience data set D_t to optimize mean-squared error between Q-network and Q-learning targets, e.g. $L_i(w_i) = E_{(s,a,r,s') \sim D}[(r + \gamma \max_{a'} Q(s', a', w_i^-) - Q(s, a, w_i))^2]$. Experience replay provides a way to remove correlation in the observed sequence and smooth over the changes in the data distribution.

Second, policy may change rapidly even though we only have slight changes to the Q-values. In other word, the policy may oscillate and the distribution of data can swing from one extreme to another. It is worth noting that the Q-learning targets are different from other supervised learning algorithms which are fixed before the training starts. In fact, the Q-learning targets heavily rely on the network weights w^-. In order to solve this problem, the parameter w^- is fixed from previous iteration when we optimize the loss function. And then, the target network weights w^- are periodically updated with the parameter w every n steps.

Third, since the scale of rewards and Q-values may vary from different environment, it may result in unstable issues when we back propagate the networks. To address this, it is a common way to clip rewards or normalize networks adaptively to sensible range, for instance between -1 and 1. -1 denotes all negative changes in the rewards and 1 denotes all positive changes. When the reward stays unchanged, it will return 0. This approach limits the error derivations. It also ensures that gradients are well-conditioned and makes it possible to train multiple agents with the same learning rate. However, this setting decreases the performance since it can no long differentiate between small and large rewards.

The main notations of the deep Q-networks are fully explained above. In the next section, we will show implementation details for training the standard testing domains. For easy understanding, the deep Q-networks can be summarized in the following pseudocode.

Algorithm 5 Deep Q-Networks: [12]

1: Initialize the replay memory D to a fixed capacity N.
2: Initialize the value function Q with random weights w.
3: initialize the target value function Q^- with weights w^-, where $w^- = w$.
4: **for** $episode = 1, M$ **do**
5:     Initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
6:     **for** time step t=1,T **do**
7:         Select a random action a_t based the ϵ- greedy policy. Otherwise select an action $a_t = argmaxQ((s_t), a; w)$
8:         Send the action a_t to the environment and receive the screenshot x_{t+1} with the corresponding reward r_t
9:         Set $s_{t+1} = s_t, a_t, x_{(t+1)}$ and preprocess $\phi_{t+1} = \phi(s_1)$
10:         Store the transition experience $e_t = (\phi_t, a_t, r_t, \phi_{t+1})$ in the replay memory D
11:         Set $y_k = \begin{cases} r_k & \text{if episode terminates at the step } k+1, \\ r_k + \gamma \max_{a'} Q^-(\phi_{k+1}, a'; w^-) & \text{otherwise.} \end{cases}$
12:         Minimize the loss function $(y_k - Q * \phi_j, a_j; w)$ by stochastic gradient descent
13:         Update the weights of target networks $Q^- = Q$ every C steps
        end for
    end for

Chapter 5

Case Study

5.1 Implementation details of the deep Q-network

Initially, we apply a preprocessing function ϕ aimed at reducing the parameters and computation cost. It extracts the Y channel, knows as the luminance channel, from the RGB screenshot and crops to an 84*84 region. To facilitate learning efficiency, we stack 4 most recent frames as the input of the Q-function. In traditional reinforcement learning algorithms, Q-values is a mapping from the state-action pair to a scale number. This method results in a linear increase of computing cost due to the number of available actions. Instead, we separate output units for each possible action and only the state representation is the input of the networks. Then it leads only a single forward pass to the networks.

Figure 5.1 illustrates the architecture of the convolutional neural net we use. We take a 84*84*4 image as input which is generated by the preprocessing function ϕ. The whole network consists of 3 convolutional layers and 2 fully-connected layers. The first hidden layer computers the output neurons using 32 filters of 8*8 with stride 4. The second hidden layer convolves 64 filters of 4*4 with stride 2. Then it follows the third layer which consists of 64 filters of 3*3 with only one stride. After each single convolutional layer, we periodically insert a rectifier nonlinearity function [26] $\max(0, x)$. The next is a fully-connected layer of 512 rectifier units. Then it follows the output layer the neurons of which are linearly fully-connected to each legal action.

During the experiment, the agent selects a random action a_t based the ϵ- greedy policy, where ϵ linearly decreases from 1.0 to 0.1 during certain iterations and fixes at 0.1 thereafter. It makes more sense that the agent explores as many choices as possible, since it has little predefined knowledge at the beginning. Then the action choices become more and more relying on the experience after thousands of training episodes. The replay memory size D is 1000000. As stated above, the network could process 4 recent frames at each time. During the training process, we randomly select a mini-batch of size 32 to minimize the loss function by stochastic gradient descent. The target weights w^- are updated every 10000 steps. Since the reward is rescaled in [-1,1], we fixed the learning rate as 0.00025. The discount factor is fixed 0.99 which is the default setting for many reinforcement learning algorithms. For readability, Table 1 list the parameters and their corresponding values.
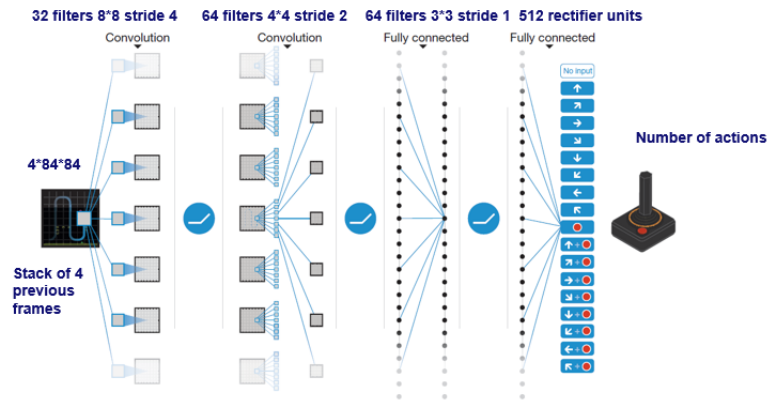
Figure 5.1: The architecture of the convolutional neural network. After the preprocessing step, the input of the convolutional neural net consists of 84*84*4 images. The inputs layer is followed with three convolutional layers and then are connected with 2 fully-connected layers. The units in the output denotes the valid actions.

Table 5.1: Parameters and values for the deep Q-network.

Parameter	Value
Reply memory size	1000000
Mini-batch size	32
The size of most recent frames	4
Dicount factor	0.99
Update frequency for target network	10000
Learning rate	0.00025
Initial ϵ-greedy policy value	1.0
Final ϵ-greedy polci value	0.1
Gradient momentum	0.95
Squared gradient momentum	0.95

The Deep Q-Network is implemented with Torch [8], which is a scientific computing framework with wide support for machine learning algorithms. The code is written with a scripting language Lua on Ubuntu platform.

5.2   Testing domain implementation

The domain implementation is based on an open source Java library BURLAP [9] that provides a wide range of planning and learning methods with standards testing domains. Since there are language barriers between Java and Lua, we decide to use socket programming over TCP/IP networks. We will demonstrate how we implement the client/server applications in the two platforms.

Initially, the sever of testing domains starts and listens for a connection request. Then the DQN asks for a connection as a client. When the connection is established, the DQN platform requests the domain status including the set of legal actions. After sending back the required info, the agent waits for the start command. During the learning process, the deep Q-network keeping sending actions and receiving tuples in a form of (x_t, r_t, ts_t), where x_t denotes the frame, r_t denotes the direct rewards and ts_t is Boolean signal that indicates whether it reaches the terminal

position. As long as the agent reaches to goal position, the deep Q-network will send a command to reset the domain status and starts a new episode. Figure 5.2 shows the sequence chart of socket communication.
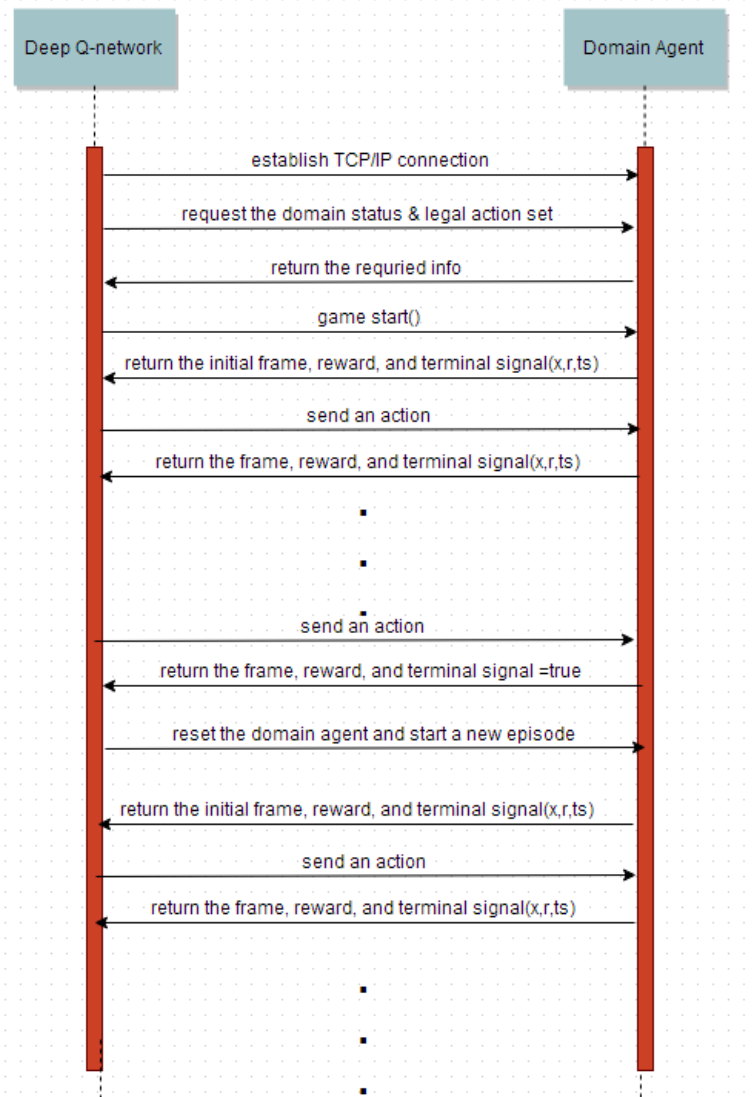

Figure 5.2: A sequence chart for socket communication. After establishing the TCP/IP connection, the deep Q-network will request the domain status and the legal action set. After receiving the info, the DQN agent will send command to start the game. During the trainign process, the DQN will keep sending actions and receiving the frame, reward and the terminal signal until the reaching the goal condition. If the terminal signal is equal to true, the DQN agent will request to reset the domain and start a new episode.

For the domain itself, the frame is repainted every 10 HZ which is the fast reaction time of a human. Considering a normal human behavior, we ask the deep Q-network select a new action every 60 HZ and repeating the last action in the interval frames.

5.3   Results

We have performed the experiments on one discrete domain Grid World, and two continuous domains: Mountain Car Problem and Inverted Pendulum Problem. During three different experiments, we use identical architecture of the deep Q-network and the exactly the same settings. It shows that this method is robust to apply in various environments without the effort of creating the specific handcrafted features. Figure 5.3 5.4 5.5 provide illustrations of how the deep Q-network learns to solve the mountain car problem, the grid world problem and the inverted pendulum problem.
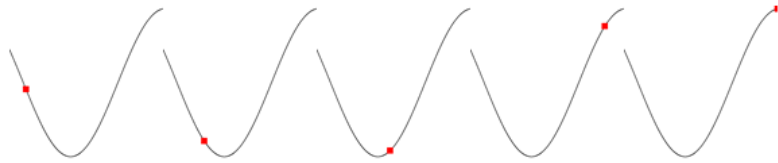

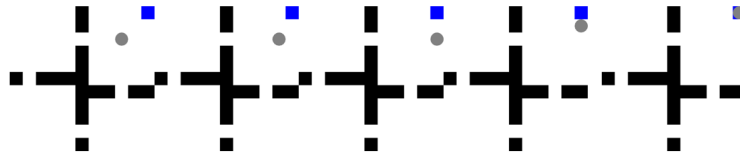Figure 5.3: An illustration of how DQN solves the mountain car problem

Figure 5.4: An illustration of how DQN solves the grid world problem

Figure 5.5: An illustration of how DQN solves the inverted pendulum problem

We compare the results with traditional Q-learning algorithms and Sarsa algorithms which learns linear policies on hand-engineered feature sets. Figure 5.6 shows the statistic results of traditional Q-learning and SARSA methods in the Grid World domain. In fact, the deep Q-network does not show better result in these domains. Here are the reasons. In this thesis, we only focus on the standard testing domains which are much simpler tasks that have been well-addressed by many reinforcement learning methods. Actually, these testing domains have lowdimensional state space and their features can be easily handcrafted. In addition, since the deep Q-network takes images as input, it always requires large memory size (6GB in our experiment) for complex computing.
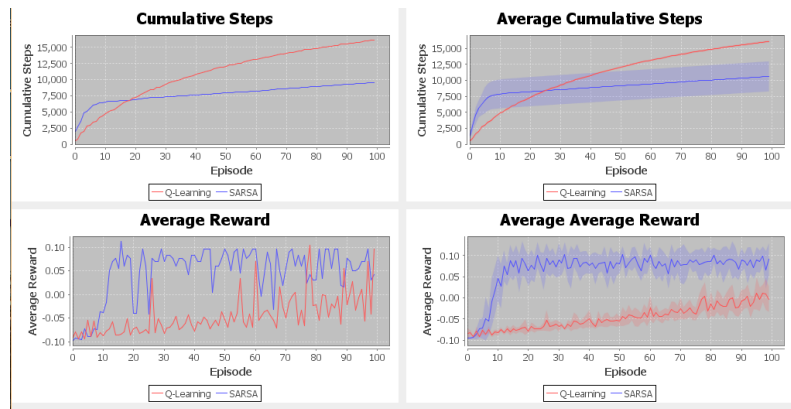
Figure 5.6: Statics results of traditional Q-learning and SARSA in the Grid World domain

The main advantage of the deep Q-network is that it builds connections between actions and high-dimensional sensory input. Meanwhile, it ensures the agent could learn diverse challenging tasks with the same architecture with no adjustment of the parameters.

Chapter 6

Conclusions

In this paper, we have studied the traditional Reinforcement Learning algorithms and analyzed their limitations in processing the high dimensional data. Then, we mainly focused on the implementation of the novel algorithm, the Deep Q-network, which combined the traditional Q-Leaning with a deep convolutional neural net. For the case study, we construct the new testing environment that could return pixels of the frame at each time step. We demonstrate the deep Q-network could successfully learn the control policy and compare its performance with traditional reinforcement learning algorithms.

6.1   Main contribution

As we know, the performance of traditional reinforcement learning algorithms heavily relies on the handcrafted features. This drawback limits the application scope of these algorithms, since some problems have high dimensional state space and are difficult to hand-engineered. Google presents a novel algorithm called Deep Q-network that leads a breakthrough in this area and could learn control policy in diverse tasks using pixels as input. It has shown evidence that the DQN could outstanding results in different Atari games which are challenging tasks for traditional RL algorithms. In this work, we apply the deep Q-network model on a few standard RL testing domains, which have been well-addressed by other RL approaches and compare their performance

The DQN agent is implemented with Torch on the Ubuntu Platform. To conduct experiment, we design and implement the environment for a few standard RL testing domains which could return the frame, reward function, and terminal signal at each time step. Since the agent and the environment are coded in different languages, we establish the communication using TCP/IP protocol. By using experience relay, reward normalization and periodically freezing the target network, we demonstrate that it is possible for the DQN to learn control policy in different domains without any adjustment of the architecture. The DQN algorithm resolves the stability issues which normally occur when we use a non-linear approximate in the reinforcement learning algorithms.

In practice, we also conduct a few experiments to facilitate the learning efficiency. By linearly decreasing the value of the ϵ -greedy policy, we ensure that the agent could randomly explore the possible states at the beginning and then more rely on its experience in the later time. We realized that the frames in these domains only contains simple curves and shapes, thus, using the stride over 1 could tremendously speed the training process without degrading the quality of the results.

6.2   Limitations and future work

Based on the current setting in our model, the DQN agent could not achieve better results than the traditional reinforcement learning algorithms. In fact, the deep Q-networks always require much

---

more time and resources for processing the images. The testing domains we use only contain simple curves and shapes. Thus, we could dramatically decrease the training time by using lower resolution images as inputs, which will further lead to a simpler architecture. Additionally, the update frequency for the target network should have impact on the learning efficiency. This parameter is set to 10000 in our model which is the same value when Google trained Atari games. The large value for update frequency may lead to the long training time, on the opposite, the small value may result in the oscillation problem. The proper value for this parameter may also relay on the domain we use. Our discussion on the optimization is not as elaborate as it could be. A detailed study in this topic could be performed in the later time. It is worth noting that we should avoid excessive customizing. It is more valuable to have a general model that could learn to behave in diverse tasks than putting effort to restrict it to a specific area.

For the experimental study, we spent a lot of time implementing the agent and the testing domains. At the same time, some tools like Torch are not well-documented. It would be beneficial to have more testing domains to show how the DQN performs in diverse tasks. This challenging job has to been fulfilled in the future.

The thesis project offers us a great chance to get an overall understanding of the reinforcement learning and deep learning area. The emerge of the deep Q-network is a major step forward of the Artificial Intelligence. This studies give us insights in the algorithms and the potential application prospect.

Chapter 7

Appendix

The rise in popularity of deep learning is due not only the cheaper and more powerful hardware, but also the proliferation of software which makes the deep learning algorithms easier to implement. Many of deep learning tools are open source and freely available on the web, such as Theano, Torch, Caffe and Deeplearning4jt. In this appendix, we provide an exploration of these tools based on their homepages, tutorials and the third party reviews. It serves to introduce an outline of some important features for differentiating between them. At last, we will explain the choice we made for implementing our convolutional neural nets.

7.1 Deep Learning Tools Review

This section gives a brief introduction for each tool. We will evaluate them based on their application scope, modeling capability and the performance.

7.1.1 Theano

Theano [13] [15] is developed at University of Montreal with developers in the LISA lab. It is a Python library that lets you define, optimize and evaluate mathematics using symbolic expressions. In recent years, most academic researchers in the field of deep learning rely on Theano, which is considered as the grand-daddy of deep learning frameworks. Compared with other deep learning tools, Theano is a very well-polished piece of software with exemplary documentation. Currently, many open-source deep libraries have been built on top of Theano, such as Keras, Lasagne and Blocks. It has shown evidence that more and more libraries have been established based on Theano because of the familiar and rich Python ecosystem, which means it can also be used on windows platform.

Modeling Capability: Theano is good for implementing algorithms from scratch and is well suited to data exploration. In fact, it provides a framework in terms of network architecture and transfer functions. Specifically, you just focus on designing the architecture of networks and the loss function, then you get the gradients for free. This declarative framework enables users to easily experiment with various exotic loss functions, nonlinearities and architectures. It is worth noting that Theano is not intended for neural network layer structure but the symbolic function expressions. This may result in a low level of abstraction and provide more possibility for designing. Some users complain that Theano is complicated to get the hang of since it is definitely not an intuitive way of thinking. However, considering the good documentation and the abundance of tools in the Python community, the effort in learning Theano really pays off.

Performance: Theano code can be translated into machine languages for efficient CPU and GPU computation. It shows evidence that the theano implementation of neural networks on one core

of an E8500 CPU is up to 1.8 times faster than NumPy, 1.6 times faster than MATLAB, and 7.6 times faster than a related C++ library. In the home page, it states that Theano can make the GPU use transparent and perform data-intensive calculations up to 140x faster than with CPU (float 32 only). In fact, Theano includes CUDA code generators for fast implementations of mathematical operations.

7.1.2 Torch

Torch [8] is a popular computational framework written in Lua that supports machine learning algorithms. It is used by New York Universities (NYU), Facebook AI lab and Google DeepMind group. Since Lua is not a mainstream language, the libraries for it are not as rich as ones built for Python. Torch claims to provide a MATLAB-like environment for machine learning algorithms. However, it is not as well-documented as MATLAB. Sometime it is very difficult for users to guess the function in the sourcecode.

Modeling Capability: Unlike Theano, Torch provides a much more intuitive way in designing a neural network which is in terms of a graph of layers. It means that developers do not have to consider the symbolic programming which makes this tool easy to new users. In Torch, we can create a network layer-by-layer and easily check its behavior (output, gradient) at every change, unlike Theano which must compile the whole expression first. In general, it is easy to modify the architecture, the loss function and to create a customized architecture in Torch. However, if we intend to use a neuron, a layer or a loss function that is not a part of the standard library, we must provide our own gradient. Fortunately, this does not happen very often since the standard library is complete. The main advantage for Torch code is the clear logic structure. The training procedure can be done very explicitly and the difference between new layer definition and network definition is minimal. In Caffe, layers are defined in C++ while networks are defined via Protobuf.

Performance: Luajit makes Torch very simple to integrate with C or C++ code. In the home page, it states that any C or C++ library can become a Lua library within a few hours. Torch also supports the GPU acceleration, including CUDA, Open CL, and cuDNN. In fact, Theano and Torch have been having a performance competition for a few years. Bergstra et.all (2010) showed that the Theano was faster than many other tools available at including Torch 5. In the following year, Collobert et al. claimed that Torch 7 was faster than Theano on the same benchmarks. Figure 7.1 illustrates the performance comparision.

7.1.3 Caffe

Caffe [10] is a well-known and widely used machine-vision library that ported MATLABs implementation of fast convolutional nets to C and C++. Caffe is not intended for other deep learning applications such as text, sound or time series data. Ingenuity of Caffe lies in its simplicity, in terms of defining architecture.

Modeling Capability: Caffe follows the layer-wise design in C++ and use protobuf interface for model definition. When designing the new layer types, it requires us to define the full forward, backward and the gradients update. Unlike Theano and Torch, the GPU computation is not transparent in Caffe, which means extra efforts are required for implementing the new GPU functions. In general, Caffe is suited to people who use deep learning for applications. While Theano and Torch are more tailored toward users who use deep learning for research purpose.

Performance: Caffe is simply fast in convolutional neural nets. Caffe also supports the GPU acceleration including CUDA, Open CL, and cuDNN. On the home page, Caffe claims to process over 60M images per day with a single NVIDIA K40 GPU with AlexNet. As stated above, Caffe is specialized for image classification while not for other deep learning tasks.
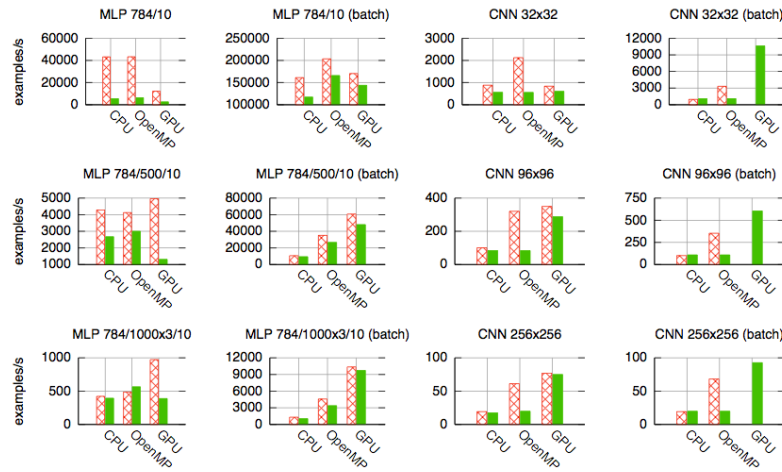
Figure 7.1: Benchmarks of Torch7 (red stripes) versus Theano (solid green) [17], while training various neural networks architectures with SGD. We considered a single CPU core, OpenMP with 4 cores and GPU alternatives. Performance is given in number of examples processed by second (higher is better). batch means 60 examples were fed at the time when training with SGD. Note that we do not handle batch convolutions using CUDA yet (but we will in few days!).

7.1.4 Deeplearning4j

Unlike the above tools, Deeplearning4j [11] is used in the business environment rather than a research tool. It is defined as a Java-based, industry-focused, commercially supported, distributed deep learning framework. Deeplearning4j aims to automate as many knobs as possible in a scalable fashion on parallel GPUs or CPUs, integrating as needed with Hadoop and Spark. Currently, Java remains the most widely used language in the enterprise environments. Deeplearning4j eliminates the language barrier for enterprise usage.

Modeling Capability: Java is a secure, network language that inherently works cross-platform on Linux, Windows and OSX desktops. It is also easy to optimize and maintain. Hence, Deeplearning4j is suitable for enterprise users with java platform required.

Performance: Deeplearning4j offers parallel GPU support for an arbitrary number of chips, as well as features that make deep learning run more smoothly on multiple GPU clusters in parallel. This is the feature many other deep learning tools do not support.

7.1.5 Tools Comparison and Our Choice

Figure 7.2 gives an outline of some important features for differentiating the deep learning tools, more specifically, the language each tool supports, the brief description. GPU acceleration and the Academic and Industry usage. SR is the abbreviation of Search Rank which is the comparative magnitude of Google Search in May, 2015.

Convolutional neural nets can be trained on these four tools. Caffe is a fast and specialized tool for image processing in deep learning field. However, it does not satisfy our requirement since it is difficult to build a Q-network within this tool. In addition, Deeplearning4j is also inappropriate. Firstly, it requires a commercial license. Secondly, Deeplearning4j is not as popular as Theano and Torch in the machine learning area which results in less tutorials and third party tools in this filed. Hence, it leaves us the choice between Theano and Torch.

Our desktop has an ATI graphic card which does not support CUDA. Therefore, the GPU perform-

SR	Tool	Language	Type	Description	Use	GPU acceleration	Distributed computing	Academic Use	Industry use	
100	Theano	Python	Library	Numerical computation library for multi-dimensional arrays efficiently	Deep and shallow Learning	CUDA and Open CL, cuDNN	Not Yet	Geoffrey Hinton, Yoshio Bengio, and Yann Le Cunn	Facebook, Oracle, Google and Parallel Dots	
78	Torch 7	Lua	Framework	Scientific computing framework with wide support for machine learning algorithms	Deep and shallow Learning	CUDA and Open CL, cuDNN	Cutorch	NYU, LISA LABS, Purdue e-lab, IDIAP	Facebook AI Research, Google Deep Mind, certain people at IBM and several smaller companies	
13	Caffe	C++	Framework	Deep learning framework made with expression, speed, and modularity in mind	Deep Learning	CUDA and Open CL, cuDNN	Not Yet	Virginia Tech, UC Berkley, NYU	Flicker, Yahoo, and Adobe	
4	Deeplearning4j	Java	Framework	Commercial-grade, open-source, distributed deep-learning library	Deep and shallow Learning	JClubas	Spark and Hadoop			

Figure 7.2: Deep Learning Tools Comparsion [12]

ance is no longer a benchmark in our evaluation. From a developers perspective, minor differences in speed are less important than other factors, like ease of use. Theano provides a comprehensive control over Neural networks formation and it is good for making algorithms from scratch and prototype them. As for the standard deep learning algorithms, it is better to use Torch since its code is non-symbolic and much easier to navigate and understand. For instance, one can write less and intuitive code in Torch to construct a convolutional neural net compared with Theano. Hence, we decide to implement the deep Q-network with Torch.

Bibliography

[1] https://en.wikipedia.org/wiki/Reinforcement_learning. 1

[2] http://reinforcementlearning.ai-depot.com/. 1

[3] http://burlap.cs.brown.edu/tutorials/bd/p1.html. 4, 6

[4] http://burlap.cs.brown.edu/tutorials/cpl/p3.html. 7

[5] "http://wii.mmgn.com/Lib/Images/News/Normal/New-Super-Mario-Bros-U-is-a-Wii-U-launch-title-1
jpg".. 8

[6] "http://library.rl-community.org/images/2/28/MountainCar-Envirornment.png"..
14

[7] "http://www.roebenack.de/content/cart-inverted-pendulum".. 14

[8] "http://torch.ch/".. 20, 27

[9] "http://burlap.cs.brown.edu/".. 20

[10] "http://caffe.berkeleyvision.org/".. 27

[11] "http://deeplearning4j.org/".. 28

[12] "http://knowm.org/machine-learning-tools-an-overview/".. 29

[13] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012. 26

[14] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009. 1

[15] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation. 26

[16] Ronan Collobert and Samy Bengio. Links between perceptrons, mlps and svms. In *Proceedings of the twenty-first international conference on Machine learning*, page 23. ACM, 2004. 9

[17] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011. 28

[18] Nicolas Heess, David Silver, and Yee Whye Teh. Actor-critic reinforcement learning with energy-based policies. In *EWRL*, pages 43–58. Citeseer, 2012. 1

[19] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006. 1

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 1

[21] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 2

[22] James L McClelland, Bruce L McNaughton, and Randall C O'Reilly. Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory. *Psychological review*, 102(3):419, 1995. 17

[23] Volodymyr Mnih. *Machine learning for aerial image labeling*. PhD thesis, University of Toronto, 2013. 1

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. 1

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. 2

[26] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010. 19

[27] Joseph ONeill, Barty Pleydell-Bouverie, David Dupret, and Jozsef Csicsvari. Play it again: reactivation of waking experience and memory. *Trends in neurosciences*, 33(5):220–229, 2010. 17

[28] Brian Sallans and Geoffrey E Hinton. Reinforcement learning with factored states and actions. *The Journal of Machine Learning Research*, 5:1063–1088, 2004. 5

[29] Pierre Sermanet, Koray Kavukcuoglu, Sandhya Chintala, and Yann LeCun. Pedestrian detection with unsupervised multi-stage feature learning. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pages 3626–3633. IEEE, 2013. 1

[30] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998. 1

[31] John N Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. *Automatic Control, IEEE Transactions on*, 42(5):674–690, 1997. 16

[32] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992. 1