Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Hierarchical Reinforcement Learning using Neural Networks

Master Thesis
im Arbeitsbereich Knowledge Technology, WTM
Prof. Dr. S. Wermter

Department Informatik
MIN-Fakultät
Universität Hamburg

vorgelegt von
Sohaib Younis
am
09.11.2015

Gutachter: Prof. Dr. S. Wermter
Dr. C. Weber

Sohaib Younis
Matrikelnummer: 6412088
Emil-Andresen-Straße 34c
22529 Hamburg

Abstract

Hierarchical reinforcement learning solves complex reinforcement learning tasks by dividing them into subtasks and solving them individually in order to solve the whole task. This thesis presents some approaches that try to solve complex reinforcement learning problem hierarchically using biologically inspired neural networks. The currently available hierarchical reinforcement algorithms are able to solve complex tasks hierarchically once the hierarchy has been defined by the programmer but generally are unable to find any hierarchy in the tasks themselves. The goal of this thesis, in addition to solving a complex learning task, is to discover hierarchy in the task.

The thesis then presents the architectures for each approach and how they performed. The results of these approaches show that overall they were able find some structure, although not hierarchical, and solve the defined task hierarchically.

Contents

List of Figures

Chapter 1

Introduction

Reinforcement learning is a branch of machine learning inspired by the process with which animals in nature acquire their complex behavior of seeking rewards and avoid punishments. For example, a mouse needs to explore new places in search of food. In this scenario the mouse is faced with the problem of being in an unknown environment, having no clear direction to reach its targets and going through a lengthy sequence of decisions in order to find food. If the mouse explores a place but it has no food then it is punished with hunger, otherwise if the place has food then mouse is rewarded with a full belly. After several tries the mouse learns to avert hunger by avoiding the places with no food and instead favors the places with food. The mouse learns this behavior through a trial and error technique called reinforcement learning.

This branch of machine learning algorithm that learns through reinforcement learning is called an agent. An agent can learn optimal or near-optimal plan or behavioral strategy while interacting directly with the external environment. It is an on-line trial and error learning method which means the reinforcement learning agent updates its behavior from receiving rewards after each interaction with the environment. Reinforcement learning algorithms have had much success in real world applications. Reinforcement learning specially became well known in the scientific community when computer programs that learned to play checkers [28] and backgammon [32] using reinforcement learning showed results similar to many human experts in these games. Reinforcement learning has also been successfully used in many other areas like robotics, elevator scheduling [6], dynamic channel allocation in cellular telecommunications [29] and autonomous helicopter flights [19].

Reinforcement learning relies on various algorithms like dynamic programming and temporal difference for on-line learning, but these algorithms suffer from Bellman's "curse of dimensionality" [3], which means the memory and computational requirements of these algorithms for learning grows exponentially as the number of variables representing the environment increases. The reason behind this is that these algorithms treat the state space as one huge flat search space. This means that the paths from the start state to the goal state are very long because the length of these paths determines the cost of learning and planning. These

algorithms also have difficulty in scaling up complex problems and transferring knowledge between tasks. Many complex reinforcement learning tasks have some structure that allows them to be broken down and organized hierarchically into smaller sub-problems. These sub-problems can be solved easily and combined together to solve the whole problem. These sub-problems are often repetitive, which means the solution of these sub-problems can be reused multiple times in different circumstances in the environment. Therefore, a reinforcement learning agent can scale up to solve complex problems by exploiting this repetition within hierarchy. This type of reinforcement learning is called hierarchical reinforcement learning. Hierarchical reinforcement learning can be further explained using the same example of mouse. Imagine that this time the mouse not only needs to search for food but also for water and his friends, in this particular order. Therefore the mouse has three types of targets, which means the mouse, in addition to learning how to reach a specific target, also needs to learn to reach them in correct order. The process of learning for the mouse can therefore be divided into three subproblems where each subproblem contains the learning problem of reaching a specific target. When the solutions for all the subproblems are combined, it can solve the complete mouse problem of reaching all the targets in the required order. Hierarchical reinforcement learning has also many real world applications in many fields which deal with large and complex environments like computer games [23], elevator control [20] and flight simulators [27].

The decomposition of a learning problem into sub-problems is usually done manually by a programmer. Some of the hierarchical reinforcement learning methods like Options [31], Hierarchy of Abstract Machines [22] and MAXQ value decomposition [9] do it manually. The problem with these methods is that the programmer must craft appropriate sub-goals and sub-task termination conditions, decide on safe state abstractions, allocate reward hierarchically or program stochastic finite state controllers with the right choice points. The process of doing all this manually takes a lot of effort, which should be avoided as stated by Dietterich: "It is our hope that subsequent research will be able to automate most of the work that we are currently requiring the programmer to do [9]". Another method called HEXQ tries to decompose the state space to create hierarchies of subtasks by finding frequently visited states and introducing subgoals at those states [11].

A complex reinforcement learning task can be broken down into small parts manually but this process can take time and the decomposition may not be optimum, therefore the objective of this thesis is to find an approach to the discovery of hierarchical structure in reinforcement learning. The key idea is to identify a structure within the actions or tasks of a complex reinforcement problem. This structure in the environment can help to decompose the problem into small parts which can be solved independently. It can also help a programmer to identify and analyze the sub-components of the problem that are scalable, depending on their solution. This analysis can also help the programmer to make the solution more optimal by identifying the poorly solved portions of the problem.

This thesis is organized in such a way that the introduction is followed by basics about reinforcement learning and neural networks with some biological background

of reinforcement learning in Chapter 2. In Chapter 3, hierarchical reinforcement learning is introduced and some of its famous approaches are mentioned. Chapter 4 explains the objective of this thesis. It also introduces a reinforcement learning environment and different approaches made by the author to find hierarchy for reinforcement learning in that environment. The implementation of these approaches and their respective results and shown in this chapter. Finally, the conclusion of the thesis is presented in Chapter 5.

Chapter 2

Basics

2.1 Reinforcement Learning

A reinforcement learning problem is learning to reach a specific or goals by interacting with the environment. The reinforcement learning problem can be modeled as an interaction between an agent and environment. The learner and decision-maker part of the model is called the *agent*. Everything outside the agent is called the *environment*. It can be the part of the model with which the agent can interact. The objective of a reinforcement learning agent is to learn the optimal sequence of actions depending on the states in order to maximize the final reward. Markov decision process provides an essential framework for reinforcement learning.

2.1.1 Markov Decision Process

A Markov Decision Process (MDP) is a stochastic model for a random system that changes its states according to transition rule, depending on its previous state. It is the principle component in reinforcement learning. The components of an MDP are as follows.

State

State is represented by the values or configurations a system can take. It is denoted by s, whereas s_t represents the particular state of the environment at time t.

Action

Actions denoted by a is a set of possible actions an agent can perform. An action is an output from the agent which influences the environment. The action performed at time t after observing state s_t is represented as a_t.

Reward

The reward is a measure of how good an action or state is for short term performance of the agent. The agent receives reward r_{t+1} when the environment

transitions from state s_t to s_{t+1} after performing action a_t.

Markov Property

The Markov Property states that in an MDP the response of the environment at time $t+1$ is dependent only on the state and actions at time t. In other words the next state of the environment depends only on the current state and action and not on any previous states or actions.

$$Pr\{s_{t+1}, r_{t+1} \mid s_t, a_t\} \tag{2.1}$$

The Transition Probabilities

The transition probabilities determines the probability of the environment of moving to the next state depending on the current state and action. The transition probabilities follow Markov property. The probability of the environment of moving to state s_{t+1} after taking an action a_t in state s_t can be represented as follows:

$$P^a_{ss'} = Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\} \tag{2.2}$$

Similarly the the expected value of the next reward can be calculated using s, a and s'.

$$R^a_{ss'} = E\{r_{t+1} = r \mid s_t = s, a_t = a, s_{t+1} = s'\} \tag{2.3}$$

2.1.2 Reinforcement Learning Model

In reinforcement learning the agent learns about its environment through self experience using trial and error. It assumes that the world can be divided into a set of finite states and the agent can take one of many possible actions at each state. The agent interacts with the environment using *action* (a), depending on the *state* (s) of the environment and the *reward* (r) received from the environment. At each time step, the agent observes the current state of the system and takes an action accordingly. After implementing the action the agent is given a reward, after which the new system state is observed, as shown in Figure 2.1.

Return Function

As the goal of the reinforcement learning agent is to maximize the accumulated future rewards, the total sum of rewards collected can be calculated using a return function.

$$R(t) = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.4}$$

In the equation above $R(t)$ represents the cumulative discounted reward at time t and γ is the discount-rate parameter, where $0 \leq \gamma \leq 1$. The discount rate

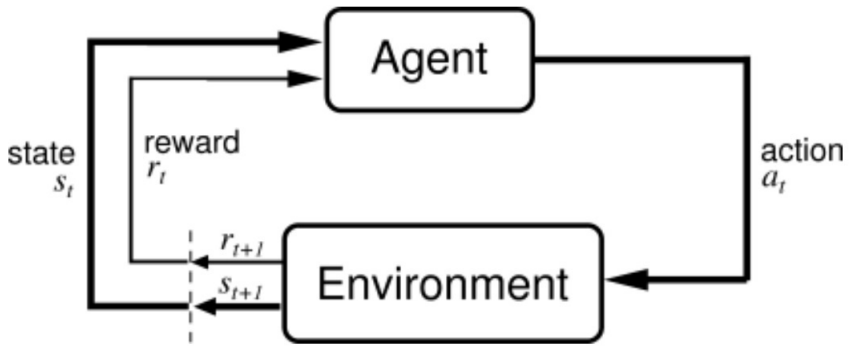

Figure 2.1: The interaction between agent and environment in the reinforcement learning model [30].

determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately.

Policy

An agent's *policy* is the probability of selecting an action a in state s at time t. In other words, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. The policy defines the behavior of the agent at any given time. The goal of the agent is to learn an optimum policy π^* that maximizes the future rewards.

$$a = \pi_t(s) \tag{2.5}$$

State Value function

The agent by taking into consideration the each state, action, reward and the resulting state, learns the long term *value* $V(s)$ of staying in each state. The value of a state, also known as state value, is a measure of how good the state is by estimating how much reward the agent can receive in the future starting from this state. The state closer to goal state (the state where the agent receives final reward) will have a higher value than the states that are farther. The state with the highest resulting value is the most optimum state for the system to stay in. The value function can be defined formally as

$$V(s) = E\{R_t \,|\, s\} = E\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,|\, s_t = s\} \tag{2.6}$$

where $V(s)$ is the accumulated expected reward for state s. When the agent is following a policy π then the value of a state is represented as $V^\pi(s)$.

$$V^{\pi}(s) = \underset{\pi}{E}\{R_t \,|\, s\} \tag{2.7}$$

where $\underset{\pi}{E}\{\}$ denotes the expected value when the agent is following policy π. The optimal state value function $V^*(s)$ is the maximum value function over all policies.

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{2.8}$$

Action Value function

The *action value* function, also known as *Q-value* $Q(s, a)$, is the measure of expected accumulated reward for taking action a in state s. Similar to the value function for state, the action value is an estimate of how good the action is in current state. The action with the highest Q-value is the optimum action for the current state. The action value function is defined as:

$$Q(s, a) = E\{R_t \,|\, s, a\} = E\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,|\, s_t = s, a_t = a\} \tag{2.9}$$

where $Q(s, a)$ is the accumulated expected reward for action a in state s. When the agent is following a policy π then the action value is represented as:

$$Q^{\pi}(s, a) = \underset{\pi}{E}\{R_t \,|\, s, a\} \tag{2.10}$$

The optimal action value function $Q^*(s, a)$ is the maximum action value function over all policies.

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a) \tag{2.11}$$

2.1.3 Temporal Difference Learning

Let us consider an agent that does not know its environment (the probability of transition from one state to another), it is still possible to learn the value for each state and action through experience without building a model of the environment by using temporal difference. The reinforcement learning methods of *temporal difference* (TD) are a combination of Monte Carlo and dynamic programming. TD methods can learn directly from raw experience without a model of the environment. The TD learning method update their estimate at each time step. At time $t + 1$ the TD method updates the value $V(s_t)$ using r_{t+1} and $V(s_{t+1})$.

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.12}$$

where α is the learning rate and γ is the discount factor.

Sarsa

Sarsa is an on-policy action value TD learning method. This means that at each time step the agent selects an action using its policy and updates the action value for that time step using Sarsa. Sarsa is an acronym for state, action, reward, state, action.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.13}$$

Q-Learning

If neither model of the environment nor the policy for the agent is known, the agent can still learn using Q-learning. *Q-Learning* is an off-policy action value TD learning method. This means that at each time step the agent selects an action using any policy-free action selection method (discussed below) and updates the action value for that time step using Q-learning. This way neither the state transition probability nor a policy for each state is required. This method of reinforcement learning was introduced by Watkins in 1989 [35].

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.14}$$

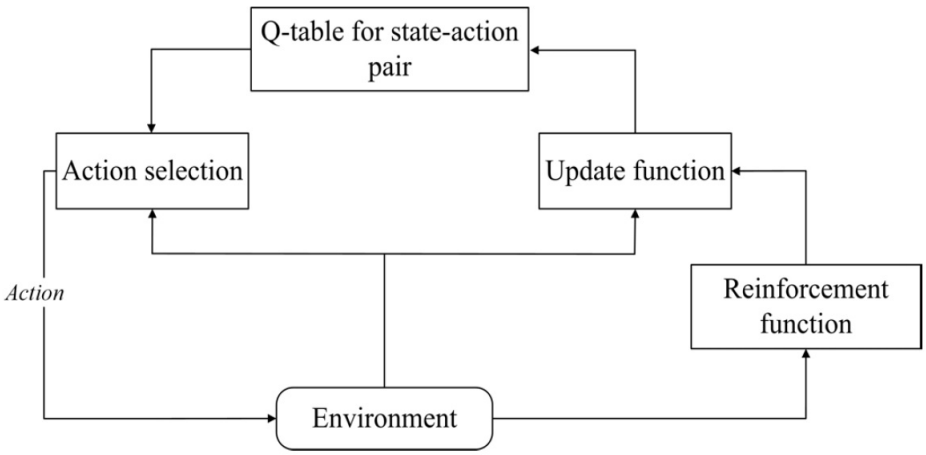

Figure 2.2: Model of Q-Learning [24].

A model for Q-learning is shown in Figure 2.2. In the figure the reinforcement function is the reward received by the agent, update function updates the Q-values by using Equation 2.14 and action selection is done using a policy-free action selection method (discussed below). The table for state-action pairs is a database that holds all the action-values for each action in each state.

2.1.4 Semi-Markov Decision Process

In an MDP the amount of time between state transition or implementing action is uniform and equal, however in a Semi-Markov Decision (SMDP) framework the

time between state transition can vary and is a random variable. In a discrete time SMDP [14], the state transition can occur at any (positive integer) time step. Due to this, time also becomes a factor in SMDP framework. The state transition probability defined in Equation 2.2 can be redefined as

$$P^a_{ss'} = Pr\{s_{t+1} = s', \tau \mid s_t = s, a_t = a\} \tag{2.15}$$

where τ is random variable that denotes the waiting time for state s when action a was being executed. Therefore, the state transition from state s to state s' occurs after τ time steps when action a was executed. The reward received after this state transition is represented by the amount of discounted reward accumulated over the waiting time τ in state s.

$$r = \sum_{t=0}^{\tau-1} \gamma^t r(s, a, t) \tag{2.16}$$

and the learning step for SMDP-based Q-learning becomes:

$$\Delta Q(s, a) = \alpha[r^\tau + \gamma^\tau \max_{a'} Q(s', a') - Q(s, a)] \tag{2.17}$$

2.1.5 Policy-free Action Selection

A common problem in reinforcement learning is finding a balance between *exploration* and *exploitation*. Exploration means selecting sub-optimal action in order to discover new features about the environment. Exploration can lead to a lower reward in short term but it's aim is to find optimal actions that are currently not known, that can lead to a higher long term reward. Exploitation means selecting the most optimum action currently known in order to achieve higher reward. Exploitation can lead to higher short term reward but as it cannot discover any better actions than those that are currently unknown, which can lead to a lower long term reward. As it is not possible to do exploration and exploitation at the same time on a single action, a good action selection method tries to find a balance between them so that it not only receives high rewards in long term but also in short term.

Random

A random action selection method always selects the action at random, regardless of its Q-value. This is a pure exploration method.

Greedy

A greedy action selection method always selects the action with highest Q-value. This is a pure exploitation method.

$$Q^*(s_t, a) = \arg\max_a Q(s_t, a) \tag{2.18}$$

ϵ-greedy

An alternative to greedy is ϵ-greedy. This method selects greedy action most of the time, but every once in a while, say with small probability ϵ, instead selects an action at random. The advantage of this method is that as time approaches infinity, the agent would have explored all the environment while selecting greedy action most of the time.

Softmax

Although ϵ-greedy action selection is an effective and popular means of balancing exploration and exploitation in reinforcement learning, it has two drawbacks. First, it keeps on exploring with probability ϵ even after the agent has the most optimum action, which leads to undesired delay for the agent to solve the reinforcement learning problem. Its second drawback is that every time it performs exploration it selects the actions randomly, which can sometimes lead to the selection of worst action. The solution is to select action based on the probability based on its action value. This method is called *softmax* action selection method and it uses Boltzmann distribution to calculate the probability of selecting an action.

$$Pr(a \mid s_t) = \frac{e^{Q(s_t,a)/\tau}}{\sum_{b=1}^{n} e^{Q(s_t,b)/\tau}} \tag{2.19}$$

Where τ is a positive parameter called the *temperature*. High temperatures cause the actions to be all (nearly) equiprobable. Low temperatures cause a greater difference in selection probability for actions that differ in their value estimates. In the limit as $\tau \to 0$, softmax action selection becomes the same as greedy action selection.

2.2 Artificial Neural Networks

Artificial neural networks are a family of statistical learning models inspired by biological neural networks. Just like their biological counterparts, the artificial neural networks consists of a large number of interconnected artificial neurons. The main objective of a neural network is to learn a pattern or an approximation function based on its input. The connections have numeric weights that controls the signals between neurons, which can be tuned based on experience making neural nets adaptive to inputs and capable of learning. Every time the network receives an input, it calculates an output. The output is compared with the expected results and the difference is declared as error. This error term is used by the network to update the weights so that the output becomes closer to the expected results. If the network is generating desirable output then there is no need to adjust the weights but if the network is generating undesirable output, or in other words if the difference between the calculated output and the expected results is large, then the network adapts, altering the weights in order to improve subsequent results.

Following are some types of artificial neural networks that are used in this thesis.

2.2.1 Multilayer Perceptron

A multilayer perceptron (MLP) is an acyclic artificial neural network that maps a a set of input onto a set of predefined output. A MLP consists of multiple layers of neurons called *perceptron* in a directed graph, with each layer fully connected to the previous one. The multilayer perceptron uses a supervised learning technique called *backpropagation* for training, which means it trains with the help of target output from the user. Compared to its basic components, the standard linear perceptron, a MLP can process nonlinear data. It does this because all its neurons except the input neurons can have a nonlinear *activation function.*

Perceptron

A perceptron is a single unit of classifier or predictor. The perceptron computes a single output (y) from multiple real-valued inputs (x_i) by forming a linear combination according to its input weights (w_i) and then possibly putting the output through some nonlinear activation function. Mathematically this can be written as:

$$y = \varphi(\sum_{i=1}^{n} w_i x_i + b) = \varphi(\mathbf{w}^T \mathbf{x} + b) \tag{2.20}$$

where $\mathbf{w}$ denotes the vector of weights, $\mathbf{x}$ is the vector of inputs, b is the bias and φ is the activation function. The whole process of this operation is shown in Figure 2.3.

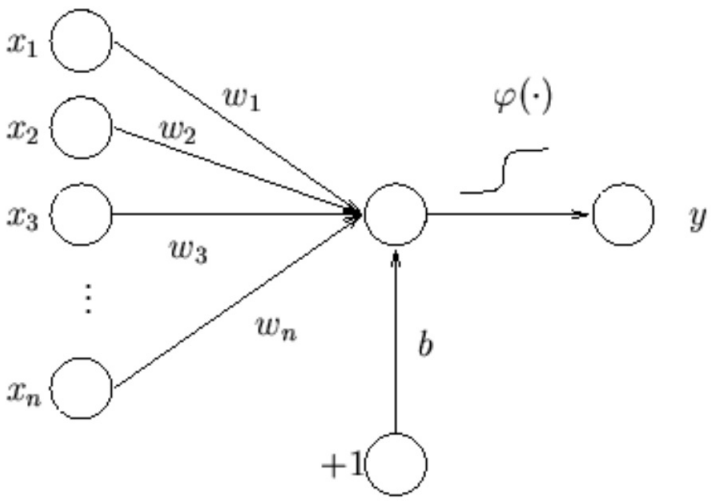

Figure 2.3: Signal-flow graph of the perceptron [13].

Activation Function

The activation function of a neuron defines the output of that neuron given the set of input. The activation function can be linear or nonlinear. The activation function is applied to the net input of the neuron as shown in Figure 2.4.

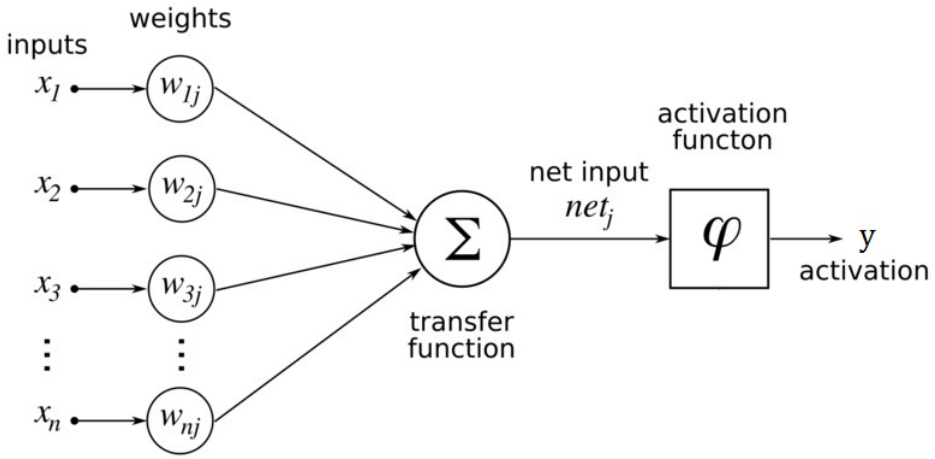

Figure 2.4: Diagram of a perceptron neuron with an activation function [16].

A linear activation function passes the network input to the output as it is.

$$\varphi(x) = x \tag{2.21}$$

Following are some nonlinear activation functions that are mostly used in multilayer perceptron.

Sigmoid activation function:

$$\varphi(x) = \frac{1}{1 + e^{-x}} \tag{2.22}$$

Hyperbolic tangent function:

$$\varphi(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.23}$$

Softmax activation function:

$$\varphi(x) = \frac{e^{x}}{\sum\limits_{i} e^{x_i}} \tag{2.24}$$

Linear and nonlinear activation functions can be used in combination as shown in Figure 2.5.

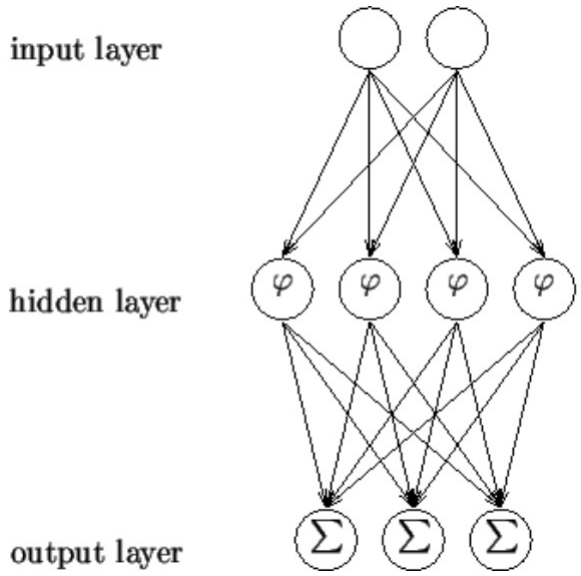

Figure 2.5: Signal-flow graph of an MLP. φ represent the nonlinear neurons and Σ represent the linear neurons.

Error

In supervised learning an error function is used to compare the current output of the network to its desired output in order to direct the training of the network. The error value is used to adjust the weights of the network using an algorithm such as backpropagation. The error is calculated as follow:

$$e = t - y \tag{2.25}$$

where:

e = error of the network
t = target value for the output
y = actual value of the output layer

Backpropagation

Backpropagation is a supervised learning algorithm that uses error function to learn and update weights. The aim of the backpropagation algorithm is to approximate the desired output by minimizing the error at each training iteration. Backpropagation, as the name suggests, propagates the error backwards through the network during the training and uses gradient descent to update the weights of the network. The gradient descent algorithm manipulates the weights of the network in order to minimize the error function. The gradient descent algorithm

works by taking the gradient of the weight space to find the path of steepest descent, in order to find the minimum of the error function. The gradient descent algorithm for weight update is shown below:

$$\triangle w = -\alpha \frac{\partial E}{\partial w} \tag{2.26}$$

where α is the learning rate of the backpropagation algorithm.

Q-Learning with an MLP

The traditional Q-learning method can be improved using a multilayer perceptron as a function approximation of Q-values for reinforcement learning. In this method the table of Q-values is replaced with a neural network. The state of the environment acts as the input for the network and the output of the network corresponds to the action values of that state. The architecture of the network for Q-learning is shown in Figure 2.6.

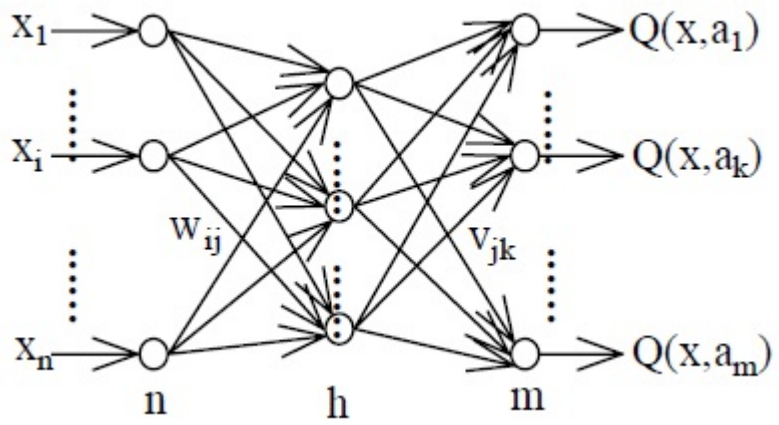

Figure 2.6: Q-Learning with back propagation neural network. The network takes current state as input and outputs the corresponding Q values of each action for the given state [15].

The RL agent selects an action based on the Q-values of the network and implements that action on the environment. The agent then updates the Q-value of that action based on the reward received using the Equation 2.27.

$$\Delta Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.27}$$

The Q-value for only the selected action is changed and all others remain the same. The error for backpropagation is calculated by using the Equation 2.25 in vector form. In this equation, all Q-values of t are equal to the Q-values of y except for the selected action. For the selected action the Q-value of that action in t changes by ΔQ. Therefore all the values in the vector e become zero except

the selected action, for which the value remains non-zero. This way the ΔQ acts as an error for the backpropagation algorithm to train the network.

2.2.2 Recurrent Neural Network

A recurrent neural network is a cyclic artificial neural network. It has multiple layers of neurons in a directed graph with at least one layer of neurons having a feedback connection to the previous layer. The feedback values are sometimes called context layer because they hold the previous time step values from the layer they originated as feedback for another layer. The context layer acts an internal memory for the recurrent network in order for the network to exhibit dynamic temporal behavior. A recurrent neural network with only feedback either from hidden or output layer is called a simple recurrent neural network. Below are two types of simple recurrent neural networks.

The Elman network is a type of simple recurrent neural network introduced by Jeffrey L. Elman in 1990 [10]. This network type consists of an input layer, a hidden layer, and an output layer, just like a three layer MLP. However, it also has a context layer that takes its values, without weights, from the output of the hidden layer. The context layer remembers these values till the next time step when they are fed back, with trainable weights, to the hidden layer as input for the next time step as shown in Figure 2.7. This gives the Elman network short term memory which is useful for predicting sequences.

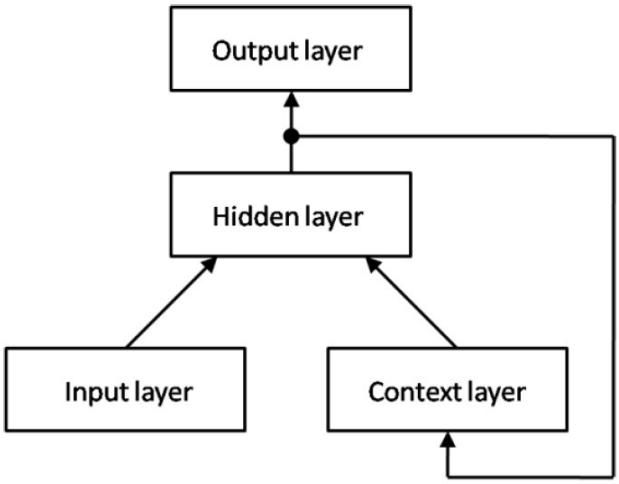

Figure 2.7: Elman Recurrent Network. Recurrent link between hidden and context layer [21].

The short term memory in Elman network comes from the feedback from the hidden layer, which consists of the internal representation in the network. This internal representation usually corresponds to different features of the input. In other words, the Elman network is able to store the features of the previous inputs

as internal representation in the context layer, it is able to exhibit short term memory.

The Jordan network is type of simple recurrent neural network developed by Michael I. Jordan in 1986 [18]. Just like the Elman network, the Jordon network also consists of an input layer, a hidden layer, an output layer and a context layer. In this network however the context layer holds the previous values of the output from the output layer instead of the hidden layer as shown in Figure 2.8.

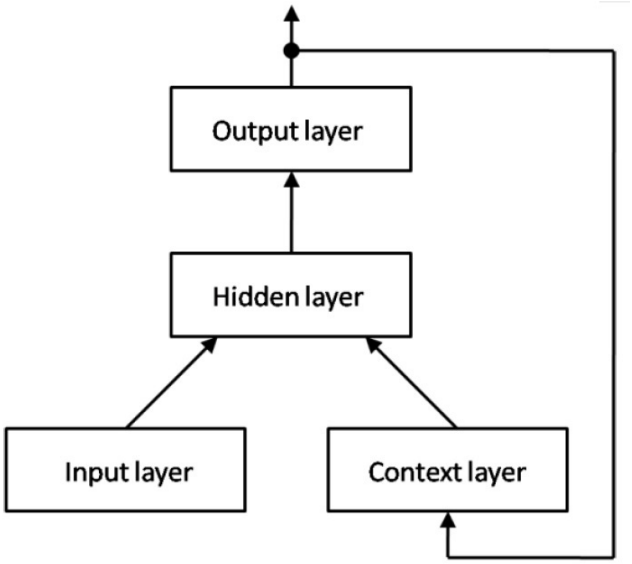

Figure 2.8: Jordan Recurrent Network.

The Jordan networks are mostly used in the field of robotics because these networks are able to remember the previous outputs of the network, which usually corresponds to adjustable parameters of the environment. This makes it different from Elman network because it not able to remember the features of previous inputs but only the values of previous outputs. Therefore, the Elman network is used to remember previous input states and Jordan network is used to remember previous output states.

2.3 Biological Background

The reinforcement learning model has a rough biological background from connections and components of the brain. The main components of reward based learning in human brain are the basal ganglia, thalamus and cortex. Although these components have their own individual functions but when viewed together they also operate as parts of a circuit that performs reinforcement learning.

2.3.1 Basal Ganglia

The basal ganglia are the part of the brain that is involved in control of voluntary motor movements and procedural learning. The basal ganglia are strongly interconnected with the cortex, thalamus and several other parts of the brain. Current research suggests that basal ganglia are associated with action selection and reward-based learning using dopamine neurotransmitter as reward [5]. One of the basal ganglia's function is to regulate motor movements by switching between behaviors. The switching between behaviors within the basal ganglia is influenced by the signal from the cortex, they determine which of several possible behaviors to execute at any given time.

2.3.2 Cortex

The cerebral cortex is a the thin outer layer of neural tissue of brain in mammals. In humans it plays a key role in memory attention and perception. The cortex is composed of three parts: sensory, motor and association areas. The sensory areas of the cortex receive and process information from senses like sight and hearing. Most of the information to the sensory areas is routed through the thalamus. The motor area of the cortex deals with precise voluntary motion of the body. The motor area sends the motion signals through the basal ganglia. The association areas are responsible for creating a perceptual model of the world and planning complex actions and behaviors. The cortex supplies input to basal ganglia and informs about the possible actions, from which the basal ganglia select one of them. The selection between behaviors that takes place in the basal ganglia is influenced by the signals from cerebral cortex by executing higher-level actions.

2.3.3 Thalamus

The thalamus is thought to be a hub of information as it acts as a relay between the cortex and other parts of the brain. All the sensory information (except olfactory) is routed through the thalamus to the cortex. Signals from basal ganglia are projected to the thalamus, which is directly connected to the motor cortex. Latest research suggests thats the function of thalamus is more selective as it processes and relays information to the cortex [1], while also receiving feedback from the cortex through the basal ganglia.

2.3.4 Reinforcement Learning Circuit

Basal ganglia is the main component responsible for reinforcement learning. It is in direct contact with the cortex and thalamus. The cortex lies at the top of the hierarchy because the decision to perform specific actions are made in the cortex. The cortex then signals the basal ganglia about these actions. The basal ganglia then selects an action and relays it through the thalamus. The thalamus informs other parts of the brain to perform that selected action. The thalamus

also reroutes this information back to the cortex so that it can remember which action was selected by the basal ganglia and performed by the body.

This flow of signals in the brain can be modeled as a reinforcement learning circuit where signals about the decision to perform specific actions starts form the cortex. These signals then go through the basal ganglia where a specific action is selected. The signals then go the thalamus which informs other parts of the brain where the selected action is performed. The thalamus finally signals this information back to cortex to store it in memory. The reinforcement learning circuit which models these flow of signals is shown in Figure 2.9.

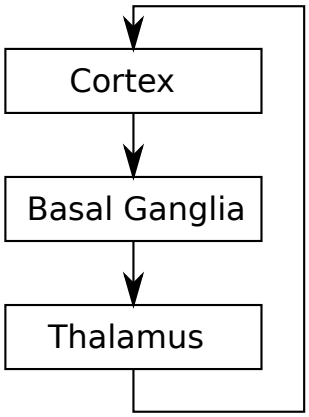

Figure 2.9: The reinforcement learning circuit.

Chapter 3

Hierarchical Reinforcement Learning

Reinforcement learning is based on the Markov decision process framework, whereas hierarchical reinforcement learning (HRL) is based on the semi-Markov decision process framework. Hierarchical reinforcement learning involves breaking the target Markov decision problem into a hierarchy of subtasks. The SMDP framework is important for HRL, as the time delays can be used to encapsulate the activity of the subtasks. The learning agent then solves the individual parts separately, which together solves the whole problem. In order for hierarchical reinforcement learning to work, the hierarchical decomposition of the problem has to be done usually by a programmer. The decomposition into subproblems has many advantages. First, policies learned in a subproblem can be shared by multiple parent tasks. Second, the values learned in a subproblem can reused for a similar subproblem. By using the learned subproblem multiple times, the learning speed of the agent greatly increases.

HRL introduces higher level actions into the RL framework, where selecting one of those actions selects a whole sequence of actions. For example, a high level action in a taxi domain might be 'Get Passenger'. Selecting this action then starts a sequence of 'Navigate to Passenger' and 'Pickup Passenger'. The action 'Navigate to Passenger' can also be further divided into a sequence of subactions to navigate the taxi to the position of the passenger. This helps to address the scaling problem by imposing more structure on the problem space. A long sequence of decisions can now be encapsulated in a single action and the value of that action can be calculated in a single learning update independently of the intervening choices. HRL also addresses the knowledge transfer problem, as the high-level actions represent natural, modular components to transfer between tasks. For example, once an agent has learned 'Navigate to Passenger', it be easily reused for another high level action such as 'Drop Passenger'.

Following are the techniques that are being used to define hierarchy and subtasks in hierarchical reinforcement learning.

3.1 Options

In the options framework the subtasks are created as preprogrammed partial policies called 'options' and can be invoked from a predefined set of states [31]. An option is described by three components: the set of states from which the option can be invoked, the policy followed by the option while it is executing and the probability of the option terminating in any state. In this SMDP, the agent chooses options instead of actions, where the original primitive actions are defined as one-step options. This framework extends reinforcement learning from learning the policy of primitive actions to a natural way of learning the policy of optimal options.

A higher level controller can decide to initiate an option in any of its invoking states. Once the option is started it follows its policy and stochastically terminates, whereupon control is returned to the controller, allowing it to select another option. Options are abstract actions in a semi-Markov decision processes. This framework extends reinforcement learning from primitive actions to a natural way to enable learning optimal option polices.

3.2 Hierarchies of Abstract Machines

Hierarchy of Abstract Machines (HAMs) uses the semi-Markov decision process framework to model abstract actions. This method permits the programmer to provide a partial policy that constrains the set of permitted actions at each point, but does not specify a complete policy for each subtask [22]. Every partial policy corresponds to one SMDP. Policies for the SMDPs are implemented as stochastic finite-state machines. This allows the policy to not only depend on the state of the world, but also on the internal state of the machine.

As this framework consists of abstract machines, which can be explained as a collection of interconnected digital logic circuits, it is not a suitable framework for this thesis because multilayer perceptrons cannot used to approximate abstract machines. Furthermore there are no known applications of HAMs in real world complex tasks at present [2].

3.3 MAXQ Value Function Decomposition

With MAXQ an episodic MDP is manually decomposed into a hierarchical directed acyclic graph of sub-tasks [7][9]. The MAXQ defines each subtask in terms of a termination predicate and a local reward function. It also specifies what it means for the subtask to be completed and what the final reward should be for completing the subtask. This structure of sub-tasks is called a MAXQ graph and has one top (root) sub-task as shown in Figure 3.1. Each sub-task is a smaller SMDP.

Terminal predicates are either terminal goal states or non-goal terminal states. The terminal states are predefined by the programmer manually. The policies for each sub-tasks are learned by giving pseudo-reward at non-goal terminal states.

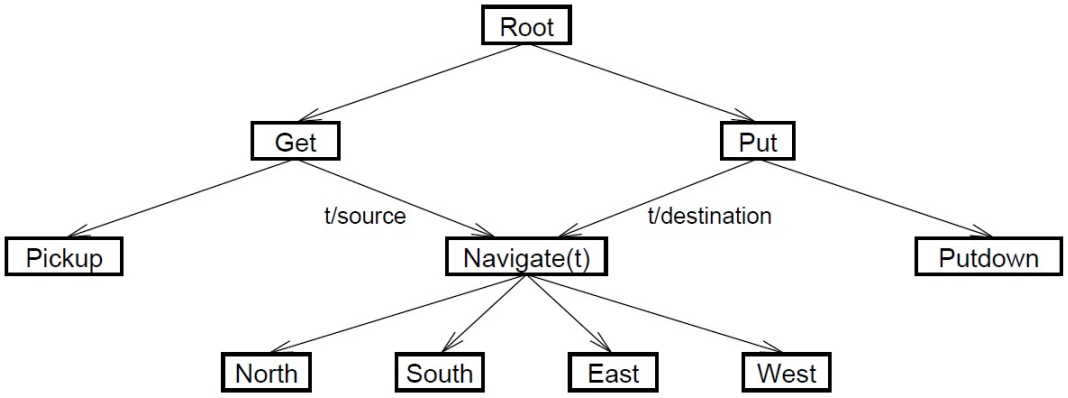

Figure 3.1: A MAXQ task graph for the Taxi problem [9].

The actions defined for each sub-task can be primitive actions or other sub-tasks. MAXQ allows sub-task policies to be reused in different contexts. Another MAXQ feature is the opportunity for state abstraction. State abstraction is key to reducing the storage requirements and improving the learning efficiency. State abstraction means that multiple states are aggregated and one completion value stored for the aggregate state in a sub-task.

3.4 HEXQ Decomposition

The HEXQ framework is similar to MAXQ but instead of defining the hierarchy manually it can automatically discover task hierarchies in an environment based on frequencies of changes in different state variables [11]. After a period of random exploration, the state variables are ordered according to how frequently they change their values. The most frequently changing state value is assigned to the lowest layer in the hierarchy. The state space of this layer is divided into a small number of strongly connected areas called *regions*. The state from where the agent can jump to another region or a distant state within the region is called *exit state* and the state from where the agent can enter the region from another region or exit state is called *entry state* as shown in Figure 3.2. The agent first learns a sub-policy to reach the regions. The sub-policy is reused to accelerate the learning of other goals that have the regions as intermediate points.

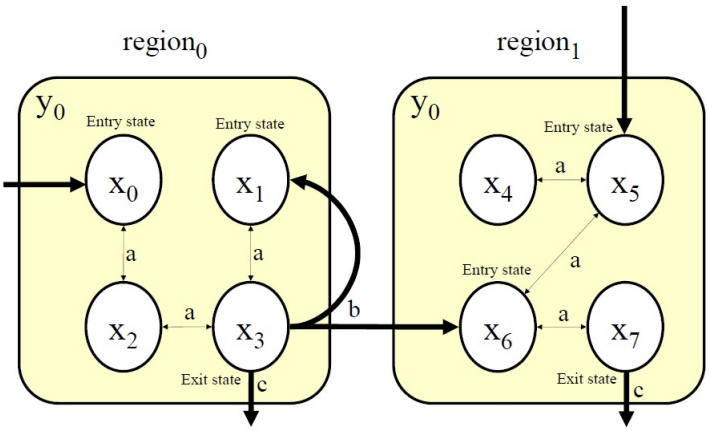

Figure 3.2: A HEXQ partition of states into two regions with exit and entry states [12].

Chapter 4

Implementation and Results

The aim of this thesis is to find a method that can scale up simple reinforcement learning. Two methods for scaling up reinforcement learning algorithms are function approximation and hierarchical decomposition. These methods are not mutually exclusive and function approximation may be used within hierarchical representations. The objective of the research is to find a hierarchy in a RL problem using function approximation. Neural networks are biologically inspired function approximation methods and can also be used to create a hierarchical structure in RL [4]. The results of some behavioral experiments have proven that tasks in the human brain are also learned and organized hierarchically [26].

4.1 Assumptions and Parameters

4.1.1 Taxi Environment

This thesis uses the *Taxi Domain* as a simulation environment for reinforcement learning, inspired from the original taxi environment created by Dietterich [9] to demonstrate his MAXQ framework. As shown in Figure 4.1 the taxi domain consists of a 5 x 5 grid for a taxi agent. The taxi problem is episodic. There is a passenger at one of the four marked locations (chosen randomly), who wishes to be transported to another marked location (also chosen randomly). The aim of the taxi agent is to go to the passenger location, pick up the passenger, carry it to its destination and drop off the passenger. Each episode starts with the taxi in a randomly-chosen square and ends when the taxi has dropped off the passenger to its destination.

The taxi can perform six types of actions in order to complete its task. These actions include four navigation actions to move one square in *North*, *South*, *East* or *West* direction, and two passenger actions, *Pickup* and *Drop off*. A move into a wall or barrier leaves the taxi location unchanged. The agent receives a reward when it successfully completes the episode by picking up the passenger and dropping it at the drop off location. The taxi environment used in the experiments has no obstacles except the environment boundary. The inner walls are not simulated as obstacles in this thesis.

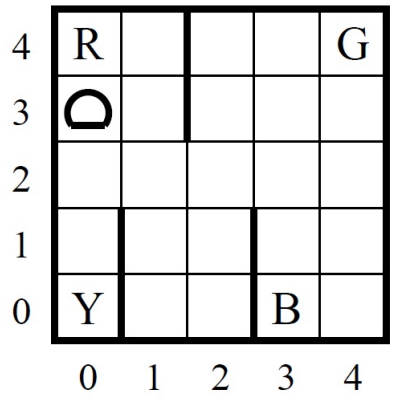

Figure 4.1: The taxi domain [8].

The representation of state space is shown in Equation 4.1. The *taxi location* represents the position of the taxi in x and y coordinate, *passenger* represents whether the passenger is on board the taxi or not, and *pickup location* and *drop off location* is the index of the pickup and drop off location respectively.

$$\text{ss (state space)} = \begin{cases} \text{taxi location}, & [0-4](\text{x coordinate}) \\ \text{taxi location}, & [0-4](\text{y coordinate}) \\ \text{passenger}, & [0,1](\text{boolean}) \\ \text{pickup location}, & [0-4](\text{index}) \\ \text{drop off location}, & [0-4](\text{index}) \end{cases} \tag{4.1}$$

Each time the agent picks up the passenger from the pickup location, it receives a small reward as shown in Table 4.1.1. The agent also receives a very small reward each time it reaches the current pick or drop off location, shown as *current goal* in the table. The agent receives a final big reward denoted by *drop off* in table when it successfully completes the episode/epoch by dropping the passenger at the drop off location. The rewards represented as *current goal* and *pickup* are called intermediate rewards, whereas the reward represented by *drop off* is called the final reward. All the approaches defined below use the final reward to learn the whole task while the intermediate rewards are used for learning subtasks. The idea to introduce intermediate rewards to learn subtasks is inspired from biology [34].

Approach \ Rewards	current goal	pickup	drop off
Multilayer Perceptron	0.02	0.1	1
Recurrent Neural Network	0.02	0.1	1
Stacked Multilayer Perceptron	0.0001	0.1	1

4.1.2 Q-learning Algorithm

The Q-learning algorithm starts by initializing a neural network, by defining the number of neurons in the input, output and hidden layer. The input for the neural

network is set as the state parameters of the environment. The network then produces an output of Q-values Q for each action available in the current state that can be implemented by the agent on the environment. These Q-values are given to the softmax action selection as shown in Equation 2.19 in order to select an action for the agent. The temperature τ for the softmax action selection starts with the value of 0.5 until it gradually decreases to the minimum allowed value of 0.05. The temperature is calculated using the following equation.

$$\tau = \frac{1}{1 + epoch}$$

The action selected by softmax is implemented by the agent on the environment. The state of the environment changes after implementing the action. The agent also receives reward r after the transition, depending on the new state. The parameters of the new state are given again to the neural network as input and the network outputs the Q-values for the new state. The maximum action value among these Q-values is set as $\max_a Q$. These new Q-values combined with reward are used to find ΔQ in order to update the Q-value of the action selected in the previous state using the following equation.

$$\Delta Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{4.2}$$

4.1.3 Neural Network

The neural network is set up by first initializing the values of weights and biases. The values are set randomly with normal distribution with mean at zero at very small variance. The neural network has two stages of functioning, first the feed forward step and then the backpropagation step. In the first stage, only the values for input layer ss are known as shown in Equation 4.1 . These values are multiplied with the weights and passed through an activation function in order to get the values for the next layer as shown below. Finally this process calculated the values for the output layer. In this experiment the output layer represents the Q-values for the actions.

$$\mathbf{s}_1(inputlayer) = ss$$
$$\mathbf{s}_2(hiddenlayer) = \varphi(\mathbf{w}_{hid}^T \mathbf{s}_1 + b_{hid}) =: \varphi(\mathbf{s}_{hid})$$
$$\mathbf{s}_3(outputlayer) = \mathbf{w}_{out}^T \mathbf{s}_2 + b_{out} =: \mathbf{s}_{out}$$

In the above equation φ represents the activation function, which in this case is hyperbolic tangent function as shown in Equation 2.23. The reinforcement learning agent uses these Q-values to select and implement an action. After implementing that action the agent calculates ΔQ for that action to update its Q-value. The network updates the Q-value of that action by considering ΔQ as the error for backpropagation. The network then uses that error to update its weights using backpropagation as shown below.

$$\delta^{out} = \Delta Q(s_t, a_t)$$
$$\delta^{hid} = \varphi'(\mathbf{s}_{hid}) \cdot (\delta^{out}\mathbf{w}_{hid})$$

In the above equation φ' represents the inverse of the activation function, which in this case is inverse hyperbolic tangent function. The following equations shows how the weights of the network are updated using the deltas calculated in the above equations.

$$\Delta\mathbf{w}^{out} = \alpha\delta^{out}\mathbf{s}_2$$
$$\Delta\mathbf{w}^{hid} = \alpha\delta^{hid}\mathbf{s}_1$$
$$\Delta b^{out} = \alpha\delta^{out}$$
$$\Delta b^{hid} = \alpha\delta^{hid}$$

4.2 Multilayer Perceptron

The initial approach towards discovering a hierarchy in a reinforcement learning task was to find a structure in the environment's state space. This would allow the state space to be broken into individual parts and reorganized into a hierarchy for the RL agent. The network architecture that was used to find a structure or pattern in the state space is shown in Figure 4.2. The multilayer perceptrons have been previously used by many people for flat Q-learning [33]. Multilayer perceptrons are excellent at function approximation, therefore they were used as a substitute of table of Q-values. The first attempt of hierarchical reinforcement learning used a single network for the flat reinforcement for learning task.

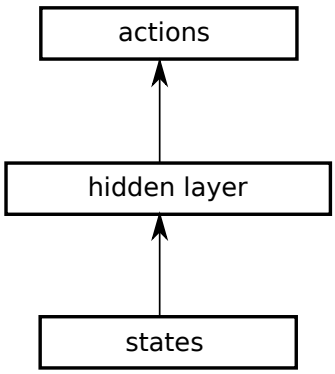

Figure 4.2: Single MLP.

For the taxi environment, the input layer represents the five parameters of the state space and the output layer represents the Q-values of the six actions that can be performed by the taxi as shown in Figure 2.6. At each time step the agent selects an action based on the Q-values at the output layer and updates these values by

backpropagation after receiving the reward. This approach converts any type of reinforcement learning problem into flat because all the states of the environment are given to the network as input all at once and the network outputs the Q-values of all the actions at once. Since there is no time delay or feedback on any kind, it is not a hierarchical approach.

The approach of multilayer perceptron for flat Q-learning starts with initializing a neural network with 5 input neurons for the states space, 6 output neurons for the actions. The number of hidden neurons were set as 30, depending on the number of input and output neurons. The experiment is run for 50000 epochs and the number of actions taken by the agent for each epoch, averaged over 50 epochs, is plotted as shown in the Figure 4.3.

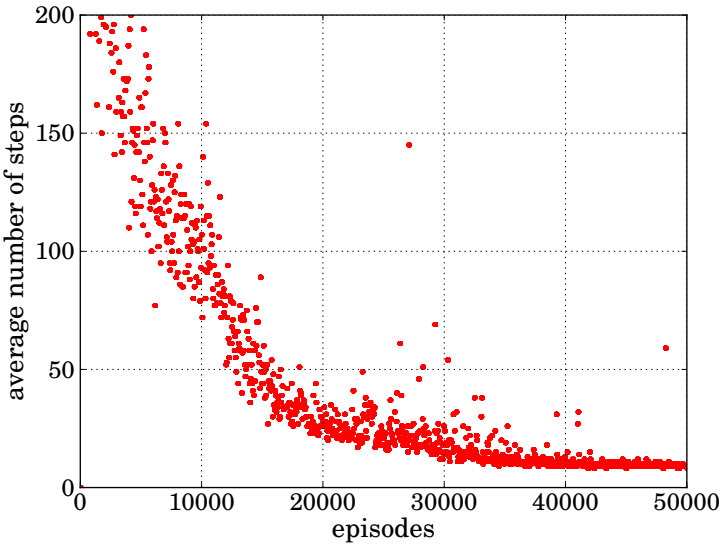

Figure 4.3: Flat Q-learning using MLP.

The figure shows that in the beginning of the experiment the agent takes an unsteadily large number of steps or actions for each epoch but this number declines gradually until 20000 epochs, at which the number of steps become steadily small. The number of actions taken by the agent for each epoch keeps decreasing until 30000 epochs when it reaches an average (for 50 epochs) of about 8 steps. The results shows that the average number of actions taken by the agent are erratic with many outliers. The cause for concern is between 20000 and 40000 epochs when the network has almost learned but the average keeps fluctuating. These results show that flat reinforcement using a multilayer perceptron is not only incapable of discovering any of hierarchal structure because the network unable to take advantage of intermediate rewards to distinguish subtasks but more importantly too unstable to solve complex tasks due to many outliers. The multilayer perceptron was only able to implement flat reinforcement learning and could not find any pattern within tasks or actions.

4.3 Recurrent Neural Network

The second approach towards discovering a hierarchy in a reinforcement learning task was not just finding a spatial structure but also a temporal structure in the environment in order to analyze the time delays created by SMDP framework. This means the agent would have to remember its previous states in order to search for a structure in time, which no longer makes it a standard Markov decision process but a semi-Markov decision process. The use of time delay allows the network to model SMDP [31], which enables the neural network to be capable of hierarchical reinforcement learning. What all of the hierarchical approaches have in common is that they use the SMDP framework as their underlying structure. The unknown time delay between action and state transition can be used to encapsulate the time when the high-level action is executing. The SMDP framework allows the agent to incorporate those time delays and rewards, and to learn how to correctly select between its complex set of actions.

The recurrent neural network architecture is inspired by the connections between basal ganglia, cerebral cortex and thalamus shown in Figure 2.9. The RL circuit in the brain can be modeled as a neural network, where the input layer acts as the cerebral cortex by encoding the states of the environment. The hidden layer can be regarded as a representation of the basal ganglia as it selects an action depending on the environment's states [37]. The output layer which is partially connected to the system output can be modeled as the thalamus as it selectively reroutes signals to the motor cortex and other parts of the cortex. The architecture of the network, which models this circuit as an RL process, is presented in Fig. 4.4. This network is similar to the multilayer perceptron except at each time step the action-values are copied to the context layer so that they can be used as input to the network in the next time step. As the recurrent neural network used in this approach is Jordan network, it is able to incorporate the time delays necessary for SMDP by remembering its previous action-values. The reason for this is because unlike the Elman network, which can remember previous states of the input, Jordan network can remember the previous values of output, which in this case are the action-values. Therefore the output of the network for each time step becomes a part of the input for the next time step. This network is able to perform the hierarchical reinforcement learning but is also able to separate the actions into different groups based on their hierarchy.

The solution to discovering hierarchy is a modular recurrent neural network. In this network the output layer is divided into two parts. The first part is the same as before containing the Q-values to six primitive actions that can be implemented in the environment. The reward for these actions is also received directly from the environment and is given as intermediate reward when subgoals have been successfully completed. The second part is defined by the programmer. Each neuron in this part represents a portion of the task. Each portion represents a high-level action, which encompasses one or more primitive or low-level actions. The number of neurons in this part determines the number of high-level actions that the learning task is to be divided into. The second part thus represents the

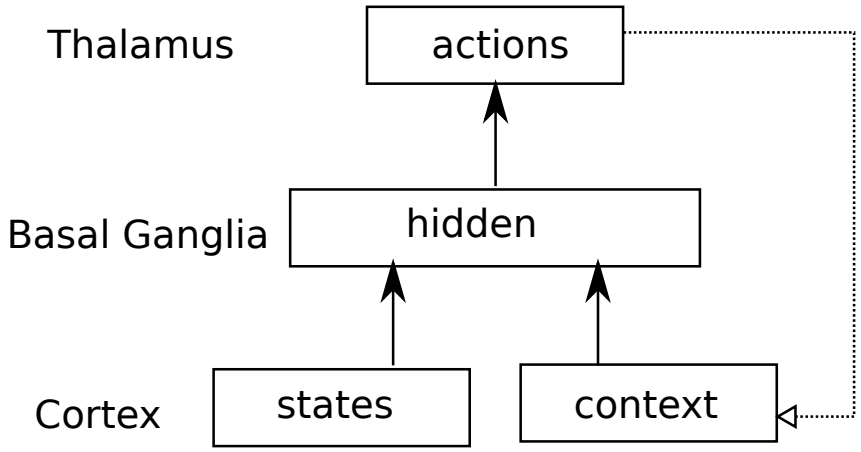

Figure 4.4: A simple recurrent neural network. The solid arrows indicate the trained weights and the line with open arrow head denote a copy of the output layer.

action-value of these high-level actions. The Q-values of this part are only updated when the agent receives any reward, since this behavior involves rare changes of activation. The agent can receive final reward for completing the global task and intermediate reward for achieving subtasks. The time delay in the network is incorporated by using feedback from the output layer, but in this network the activation on the second part of the output layer are copied to the context layer as shown in Figure 4.5. This is not a copy of the Q-values but of the selected actions. Therefore only a single node with activity 1 representing the selected action is copied, all other nodes have activation 0. The context layer determines the current high-level for the network along with the input layer.

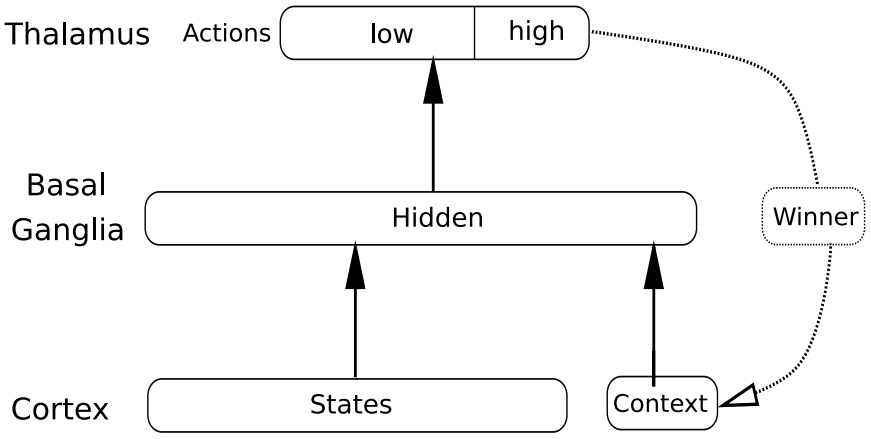

Figure 4.5: A representation of the proposed neural network. The solid arrows indicate the trained weights and the line with open arrow head denotes a copy of the high-level part of the output layer after selecting a winner.

Each part of the output layer, low and high has separate action selection mod-

ules. The action selection is done using softmax. In the first part of the output layer the selected action is implemented by the agent in the environment while in the second part the selected action is never implemented. The selected actions from both parts of the output layer are then copied to the context layer. The network learns the Q-values using the backpropagation algorithm. This means two parallel learning updates are required one for low-level executable actions and one other for higher-level actions. For example, in the taxi environment [9] where a taxi has to pick up and drop off a passenger from and to various locations, the first part of the output layer represents atomic actions to steer the taxi, whereas the second part of the output can represent the next destination for the taxi, either be it the pick up or drop off location. This would act as a higher level action for navigation of the taxi.

The approach of recurrent neural for reinforcement learning starts with initializing the network with 5 input neurons for the state space, 6 output neurons for low-level actions, 3 output neurons for high-level actions and the context layer with the same number of neurons as the output neurons which in this case is 9. As the number of input and output neurons are almost double than the previous approach, they are set as 60. At each time step the state parameters of the environment are given as input to the network. The network then calculates simultaneously the outputs for the high and low level part of the output layer. An action for each part of the output layer selected separately using softmax action selection. The action selected in the low level part of the output layer is implemented by the agent on the environment while the action selected in the high level part of the output layer is not. At this time the selected actions from both part of the output layer are also copied to the context layer. After implementing the action the state of the environment changes and the agent sometimes receives a reward depending on the new state. The new state is again given to the network as input in order to find $\max_a Q$ for each part of layer. Each time after implementing action, the Q-value of the action selected in the low level part of the output layer and the subsequent $\max_a Q$ also from the low level part are used to update the Q-value of that action using Equation 4.2. The Q-value of the action selected in high level part of the output layer however is only updated when the agent receives any reward. This update is done using Equation 2.17, where Q and $\max_a Q$ are both from high level part of the output layer only. τ in this equation is the number of low-level actions implemented by the agent after the last time the Q-value of high level action was updated, or in other words after the last time the agent received any reward. By counting the number of low-level actions implemented, τ is fact counting the number of time steps since the last time the agent received reward, this is because each low-level action takes 1 time step to execute. In case the agent receives any reward, the update in Q-values for both high level action and low level action are performed in parallel. The experiment is run for 50000 epochs and the average number of actions taken by the agent for each 50 epochs is noted as shown in the Figure 4.6.

The figure shows that in the beginning of the experiment the agent takes a large number of actions for each epoch but this number declines steeply until

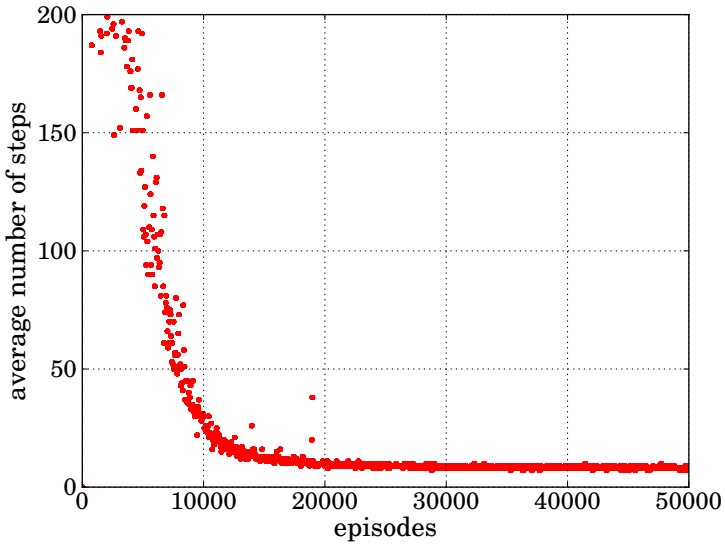

Figure 4.6: Q-learning using Recurrent Neural Network.

10000 epochs, at which the average number of steps become smaller. The results for this experiment are very smooth with very few outliers. These results show that the recurrent neural network is not only faster but also much more stable than the multilayer perceptron to solve the same complex task.

In order to find out if the network was able to learn any hierarchy, each time a low level action was implemented, the high level action selected at the time was noted and counted for a histogram. This counting was started only after the agent had completed 30000 epochs, since by that time the network had become steady and fully learned. The histogram created at the end of epoch 50000 is shown in Figure 4.7. The histogram shows the number of times each high level action was active for each time the low level action was active respectively.

In the histogram low level actions 0-3 are the four navigation actions (north, south, east and west) and low level actions 4 and 5 are pickup and drop off respectively. The histogram clearly shows that the recurrent neural network in this experiment is able to discover an emerging structure in the implementation of low level actions with the help of high level actions. The histogram shows that for the action 'drop off' the high level actions 0 and 1 are active often but in contrast to these actions the high level action 2 is almost never active. This kind of behavior of high level actions for drop off is different from all other low level actions. Similarly the high level actions also show a similar distinct behavior for the action 'pickup' as the high level actions 0 and 1 are active for a small number and in reverse to these actions the high action 3 is active for a large number of times. The histogram distinctly shows that the behavior of high level actions for pickup and drop off is undoubtedly different from their behavior for navigation actions. This also proves that the network was not only able to find a structure in the environment but was also able to segregate the actions into at least two groups depending on their

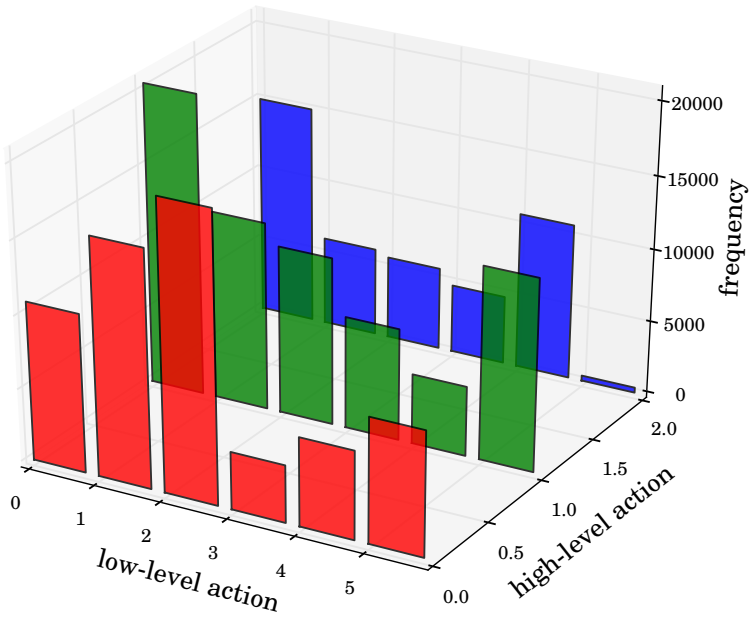

Figure 4.7: Histogram of low-level actions for each high-level action.

purpose and functionality. Due to the use of time delays through feedback, necessary for SMDP, the recurrent neural network was able to implement hierarchical reinforcement learning, but it was unable to display any visible distinction between high level and low level tasks, making it unsuitable for fulfilling the goals of this thesis.

4.4 Stacked Multilayer Perceptron

The final attempt was an inspiration from hierarchical structure designed by Rasmussen and Eliasmith [25] as it uses two networks in cascade. These two networks implement flat reinforcement but when combined in sequence the overall reinforcement learning becomes hierarchical. Each network receives a portion of state space as input, divided in to high-level and low-level parts. The low level network receives only the part of the state space relevant for performing the low level atomic actions which are pickup, drop off and navigation actions. The high level network deals with the high level part of the state space and provides the low level network a summarized version of that state space necessary for the low level network to perform high-level tasks. Each part of the network works independently to solve the reinforcement learning and is unaware of the other part. From the outside it may seem that these two parts are working together but the high level network

does not know that the summarized version of the state space relevant high-level task is going to the low level network and the low level network does not know that the summary of state space it is receiving that is relevant for high-level task is coming from high level network. This technique of separating the state space for input between high level and low level networks causes state abstraction for the low level network, which is an essential requirement for hierarchical reinforcement learning. State abstraction allows the low level part of the network to ignore the irrelevant portions of the high level part of the state space by receiving the summarized version of it. The architecture of the combination of these two networks is shown in Figure 4.8 where the first network (left) contains the low level portion of the reinforcement learning task while the second network (right) contains the high level portion of the reinforcement learning task.

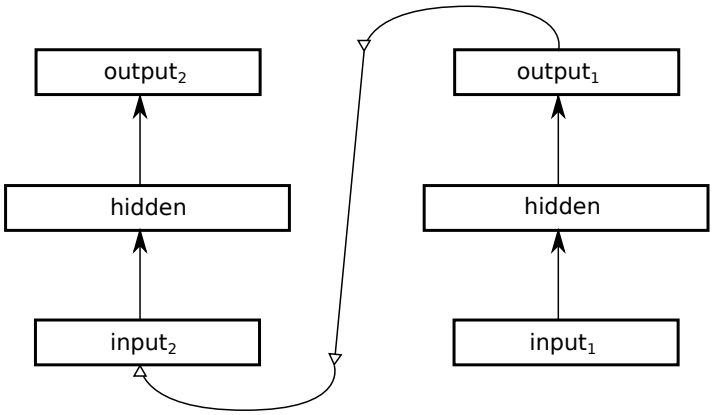

Figure 4.8: Stacked MLP.

The main difference from Rasmussen and Eliasmith's design in this attempt is that here the learning happens in two stages separately; first the low level then the high level. In their design, the high level network can modify the rewards for the low level network, but in our approach, by the time the high level network starts learning, the low level network has already learned. Therefore, the high level network cannot modify the rewards for the low level network. In the proposed approach each network consists of a multilayer perceptron which learns separately and are stacked in a sequence of high to low, which means the output of the high level network goes the the low level network as input. The learning in stacked multilayer perceptron happens in two stages. In the first stage only the low level network learns the low level tasks and when these tasks are completely learned, the network stops learning. In the second stage the high level network starts learning the high level tasks while using the low level network for implementing low level tasks. The idea of creating two stages is biologically inspired, from the observation that if a behavior or skill is learned repeatedly long enough times it becomes a habit [17]. The knowledge of this behavior is then moved to a different part of the brain where learning it again is no longer required and the knowledge can be accessed any time needed [36]. The learned behavior can be considered as the first stage in which the first network after learning is frozen and accessed as learned

behavior of low level tasks.

The taxi environment has two main tasks, transporting the passenger and navigating the taxi. Transporting passenger is a high level task and navigating the taxi is a low level task, because the destination for the navigation task is determined by the passenger being transported. Therefore the state space of the taxi environment task was divided into two parts. The high level network received the high level portion of the state space, which deals with transporting the passenger. The parameters of state space from this portion can be used to find a destination or goal for navigation part of taxi environment. These parameters of state space, as shown below, are given to the networks as input.

Input and output parameters for low level network:

$$input_2 = \{Taxi\,location(x), Taxi\,location(y), Current\,goal, Passenger\,on\,board\} \tag{4.3}$$

$$output_2 = \{North,\ South,\ East,\ West,\ Pickup,\ Drop\ off\} \tag{4.4}$$

Input and output parameters for high level network:

$$input_1 = \{Passenger\ on\ board,\ Pickup\ location,\ Drop\ off\ location\} \tag{4.5}$$

$$output_1 = \{Current\ goal\} \tag{4.6}$$

where *Current goal* is the current destination of the taxi. If the passenger is on board the taxi then it is the drop off location, otherwise it is the pick up location.

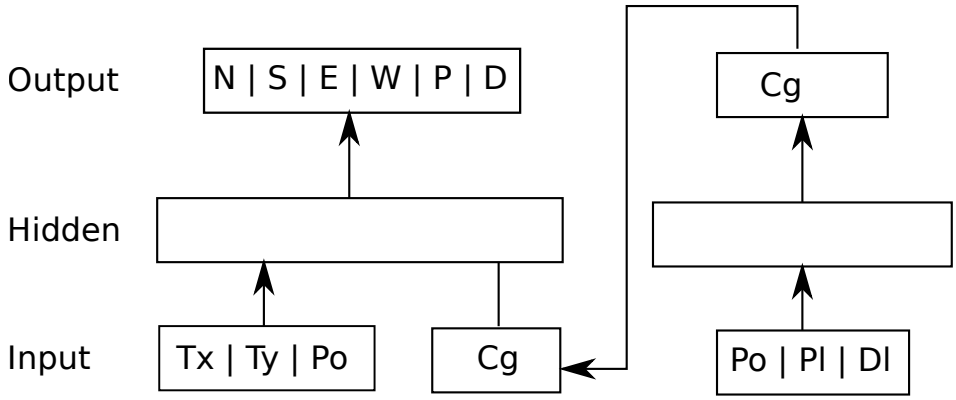

Figure 4.9: The stacked multilayer perceptron shows the representation of input and output values for the taxi environment.

The approach of stacked multilayer perceptron consists of two networks as shown in Figure 4.9. The Q-learning starts with initializing the first network with 4 input neurons represented as $input_1$ in Equation 4.3, 6 output neurons and 30 hidden neurons for the first stage of learning. This network learns for 25000 epochs by which the average number of actions taken by the agent becomes steady after 20000 epochs at around 7. At this stage the learning for the first network is stopped and a second network is initialized with 3 input neurons 4.5, 4 output

neurons (number of pick and drop locations) and 20 hidden neurons for the second stage of learning. The output from this network is given to the input of the first network for the neuron of *Current goal*. The second network then starts learning with Q-learning by receiving rewards from environment in the same manner as the first network. This network also learns for 25000 epochs, after which the average number of actions performed by the agent in last 50 epochs is around 5. The actions taken by the agent for each epoch, averaged over 50 epochs, for both stages are shown in the Figure 4.10.

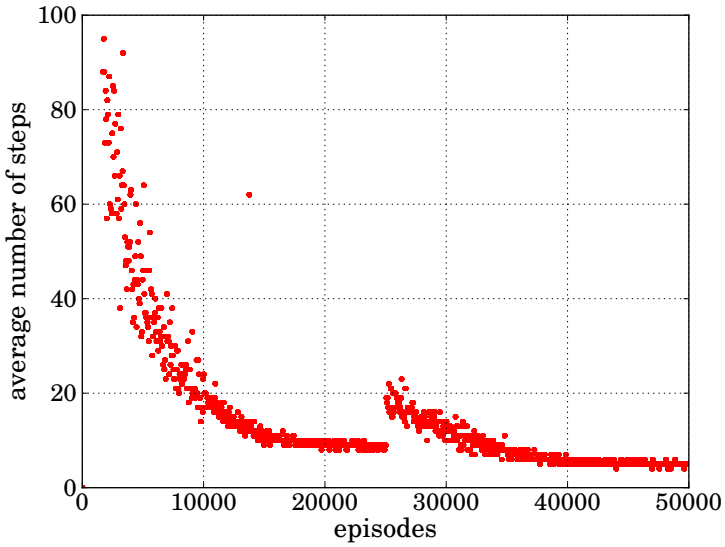

Figure 4.10: Two stage hierarchical Q-learning using stacked multilayer perceptron.

In this experiment the reward functions for the first stage and second stage are different. The first stage has the same reward function as described in Section 4.1.1 whereas in the second stage the reward for pick up is reduced to 0.01. The other change to the reward function in the second stage is that instead of receiving the reward for picking up and dropping off the passenger at the pickup and drop off positions, respectively; the agent also receives the reward for performing the pickup and drop off action at the current goal position correctly, depending whether the passenger is on board.

The results in the Figure 4.10 shows that the first stage starts learning from scratch as demonstrated by the high initial average of actions for each epoch, which gradually decreases till 10000 epochs after which the number of outliers become very small and finally fully learns till around 20000 epochs. At 25000 epochs the first stage stops learning and the second stage starts learning. The results show that the initial average of actions taken by the agent each 50 epochs during second stage is almost 5 times lower than the first stage. This proves that the second stage does not start learning from scratch but in fact uses the previous knowledge learned in the first stage. This previously learned knowledge of the first stage not only gives the second stage a small initial average but also a smooth performance with very

few outliers. This approach of stacked multilayer perceptron was not only able to implement hierarchical reinforcement learning by using state abstractions, which resulted from the division of the state space into two parts, it was also clearly able to decompose the taxi problem into high level and low level tasks, thus achieving all the goals for this thesis.

Chapter 5

Conclusion

In this thesis three neural approaches are presented, which tried to solve the taxi reinforcement learning problem hierarchically. In the taxi environment the agent can implement two types of actions, navigation and passenger transport, while also receiving intermediate rewards for completing subtasks.

In the first approach of multilayer perceptron, the network was unable to solve the taxi domain task hierarchically, because the network was not able to take advantage of intermediate rewards for dividing the problem into subtasks. The second approach of recurrent neural network did show some promise as it was able to solve the taxi task hierarchically and separate the action groups depending on their functionality i.e navigation and passenger transport, but was still unable to discover any hierarchy among them. Finally, the approach of stacked multilayer perceptron was able to divide the taxi domain task into high level and low level tasks in order to solve it hierarchically. The stacked multilayer perceptron was capable of doing this by allocating one part of the network for the high level task and the other for low task. This approach also used state abstraction by having a minimal interaction between the networks, which is also a requirement for hierarchical reinforcement learning.

The approaches of multilayer perceptron and and stacked multilayer perceptron were inspired from the previous work of other authors, but the approach of recurrent neural network is novel. This approach divides the output layer of the network into two parts, so that the network can learn to represent the features in the action space and state space of the task by encoding them in the high level part of the output layer. The network was successfully able to encode some distinct characteristics in the action space of the learning task, but was unable to learn any distinct features in the state space.

The learning processes in all the approaches presented in this paper were inspired from biology, by mimicking the connections between the cortex, the basal ganglia and the thalamus. Although the approaches presented in this thesis were unable to discover hierarchy without any manual intervention of intermediate rewards, hopefully they can be improved in the future so that they can automatically decompose any complex reinforcement task hierarchically in order to solve it efficiently. One possible solution to do this is by first improving the recurrent neural

network architecture so that it can separate the state space of the problem into at least two groups. The groups of inputs can then be used as input for multilayer perceptron networks so that each network receives each group of state space as input. The networks then connected together like stacked multilayer perceptron can solve the task hierarchically so that each multilayer perceptron network solves a part of the hierarchy determined by its inputs. This way the overall architecture can automatically decompose the hierarchy for any complex reinforcement task such that each individual network represents one subtask, whose place in the hierarchy is determined by the position of the network in the overall architecture, similar to the stacked multilayer perceptron where the first multilayer perceptron network represents the low level task and the second the high level task.

Bibliography

[1] GM Alexander, NC Kurukulasuriya, J Mu, and DW Godwin. Cortical feedback to the thalamus is selectively enhanced by nitric oxide. *Neuroscience*, 142(1):223–234, 2006.

[2] Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379, 2003.

[3] R Bellman. Curse of dimensionality. *Adaptive control processes: a guided tour. Princeton, NJ*, 1961.

[4] Matthew M Botvinick, Yael Niv, and Andrew C Barto. Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective. *Cognition*, 113(3):262–280, 2009.

[5] V Srinivasa Chakravarthy, Denny Joseph, and Raju S Bapi. What do the basal ganglia do? A modeling perspective. *Biological Cybernetics*, 103(3):237–253, 2010.

[6] Robert H Crites and Andrew G Barto. Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33(2-3):235–262, 1998.

[7] Thomas G Dietterich. The MAXQ method for hierarchical reinforcement learning. In *ICML*, pages 118–126. Citeseer, 1998.

[8] Thomas G Dietterich. State abstraction in MAXQ hierarchical reinforcement learning. *arXiv preprint cs/9905015*, 1999.

[9] Thomas G Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[10] Jeffrey L Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[11] Bernhard Hengst. Discovering hierarchy in reinforcement learning with HEXQ. In *ICML*, volume 2, pages 243–250, 2002.

[12] Bernhard Hengst. *Discovering Hierarchy in Reinforcement Learning*. PhD thesis, University of New South Wales, Australia, 2003.

[13] Antti Honkela. *Nonlinear switching state-space models*. PhD thesis, Citeseer, 2001.

[14] Ronald A Howard. *Dynamic Probabilistic Systems, Volume II: Semi-Markov and Decision Processes*, volume 2. Courier Corporation, 2013.

[15] Bing-Qiang Huang, Guang-Yi Cao, and Min Guo. Reinforcement learning neural network to the problem of autonomous mobile robot obstacle avoidance. 1:85–89, 2005.

[16] Dirk Hünniger. *Artificial Neural Networks*. Wikibooks.org, 2013.

[17] Mandar S Jog, Yasuo Kubota, Christopher I Connolly, Viveka Hillegaart, and Ann M Graybiel. Building neural representations of habits. *Science*, 286(5445):1745–1749, 1999.

[18] Michael I Jordan. Serial order: A parallel distributed processing approach. *Advances in Psychology*, 121:471–495, 1997.

[19] HJ Kim, Michael I Jordan, Shankar Sastry, and Andrew Y Ng. Autonomous helicopter flight via reinforcement learning. In *Advances in neural information processing systems*, page None, 2003.

[20] Wei Li, Qingtai Ye, and Changming Zhu. Application of hierarchical reinforcement learning in engineering domain. *Journal of Systems Science and Systems Engineering*, 14(2):207–217, 2005.

[21] Tellez Paola. Recurrent neural prediction model for digits recognition. *International Journal of Scientific & Engineering Research*, 2, 2011.

[22] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Systems*, pages 1043–1049, 1998.

[23] Marc Ponsen, Pieter Spronck, and Karl Tuyls. Hierarchical reinforcement learning in computer games. *ALAMAS*, 6:49–60, 2006.

[24] Junfei Qiao, Zhanjun Hou, and Xiaogang Ruan. Q-learning based on neural network in learning action selection of mobile robot. In *Automation and Logistics, 2007 IEEE International Conference on*, pages 263–267. IEEE, 2007.

[25] Daniel Rasmussen and Chris Eliasmith. A neural model of hierarchical reinforcement learning. In *Proceedings of the 36th Annual Conference of the Cognitive Science Society*, pages 1252–1257, 2014.

[26] Jose JF Ribas-Fernandes, Alec Solway, Carlos Diuk, Joseph T McGuire, Andrew G Barto, Yael Niv, and Matthew M Botvinick. A neural signature of hierarchical reinforcement learning. *Neuron*, 71(2):370–379, 2011.

[27] Malcolm Ryan and Mark Reid. Learning to fly: An application of hierarchical reinforcement learning. In *In Proceedings of the 17th International Conference on Machine Learning*, pages 807–814. Citeseer, Morgan Kaufmann, San Francisco, CA, 2000.

[28] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[29] Satinder Singh and Dimitri Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. *Advances in neural information processing systems*, pages 974–980, 1997.

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

[31] Richard S Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181–211, 1999.

[32] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[33] C Touzet. Q-learning for robots. *The Handbook of Brain Theory and Neural Networks,*, pages 93–937, 2003.

[34] Mark A Ungless. Dopamine: the salient issue. *Trends in neurosciences*, 27(12):702–706, 2004.

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

[36] Cornelius Weber, Stefan Wermter, and Mark Elshaw. A hybrid generative and predictive model of the motor cortex. *Neural Networks*, 19(4):339–353, 2006.

[37] Weidong Zhou and Richard Coggins. A biologically inspired hierarchical reinforcement learning system. *Cybernetics and Systems: An International Journal*, 36(1):1–44, 2004.