Investigating Neural-Based Learning Algorithms for Control

Alex Yuan Gao

Helsinki April 13, 2016

UNIVERSITY OF HELSINKI
Department of Computer Science
Investigating Neural-Based Learning Algorithms for Control

Controlling a complicated mechanical system to perform a certain task, for example, making robot to dance, is a traditional problem studied in the area of control theory.

However, evidence shows that incorporating machine learning techniques in robotics can enable researchers to get rid of tedious engineering works of adjusting environmental parameters. Many researchers like Jan Peters, Sethu Vijayakumar, Stefan Schaal, Andrew Ng and Sebastian Thrun are the early explorers in this field. Based on the Partial Observable Markov Decision Process (POMDP) reinforcement learning, they contributed theory and practical implementation of several benchmarks in this field.

Recently, one sub-field of machine learning called deep learning gained a lot of attention as a method attempting to model high-level abstractions by using model architectures composed of multiple non-linear layers (for example [KSH12]). Several architectures of deep learning networks like deep belief network [HOT06], deep Boltzmann machine [SH09], convolutional neural network [KSH12] and deep de-noising auto-encoder [VLL+10] have shown their advantages in specific areas. The main contribution of deep learning is more related to perception which deals with problems like Sensor Fusion [ONL+13], Nature Language Processing(NLP)[CvMG+14b] and Object Recognition[LSS13][HGT+14]. Although considered briefly in Jürgen Schmidhuber’s team[MGW+06], the other area of robotics, namely control, remains more-or-less unexplored in the realm of deep learning.

The main focus of this thesis is to introduce general learning methods for robot control problem with an exploration on deep learning method. As a consequence, this thesis tries to describe the transitional learning methods as well as the emerging deep learning methods including new findings in the investigation.

ACM Computing Classification System (CCS):
I.2 [Artificial Intelligence],
I.2.6 [Learning]
## Contents

1 Introduction 4

2 Reinforcement Learning 5

2.1 Markov Decision Process 6

2.1.1 Partially Observable Markov Decision Process 7

2.1.2 Markov Decision Process with Continuous States 8

2.1.3 Value Functions 9

2.1.4 Natural Actor Critic Model 9

2.2 Reinforcement Learning Methods 10

2.2.1 Policy Evaluation 10

2.2.2 Policy Gradient Methods 12

2.3 Classification of RL Problems 13

2.4 Policy Gradient with Parameter Exploration 14

2.4.1 PGPE algorithm 14

3 Deep Recurrent Neural Networks 19

3.1 Deep Learning and its Recent Advances 19

3.2 Feedforward Neural Networks 19

3.3 Recurrent Neural Networks 21

3.3.1 Finite Unfolding in Time 21

3.3.2 Overshooting 23

3.3.3 Dynamical Consistency 24

3.4 Universal Approximation 26

3.5 Learning Long-Term Dependencies 26

3.6 Training RNN 29

3.6.1 Backpropagation 29

3.6.2 Shared Weight Extended Backpropagation 30

3.6.3 Nesterovs Accelerated Gradient 32
3.6.4 Adam ............................................. 32

4 Prior Art of Combining Deep Neuron Network and RL 33
4.1 Control-focused Learning Algorithm .......................... 34

5 Experiments 34
5.1 Cart-pole Balancing ........................................ 34
5.1.1 System Implementation ................................. 36
5.1.2 Experimental Results ................................. 37
5.2 Baxter Robot Learning an Action ............................ 38
5.2.1 System Implementation ................................. 39
5.2.2 Experimental Results Using the Baxter Robot .......... 40

6 Research on Accelerating RNN Networks 42
6.1 RNN Conventional Graph ................................ 43
6.1.1 RCG example: Vanilla Recurrent Neural Network .... 43
6.2 LSTM and GRU ............................................. 44
6.2.1 LSTM .................................................. 44
6.2.2 GRU .................................................. 46
6.3 SGU and DSGU ............................................. 47
6.3.1 SGU .................................................. 47
6.3.2 DSGU .................................................. 48
6.4 Experimental Results ....................................... 49
6.4.1 IMDB Sentiment Classification Task ................... 49
6.5 MNIST Classification from a Sequence of Pixels .......... 51
6.5.1 Text Generation ....................................... 53
6.6 Conclusion .................................................. 54

References 56
1 Introduction

Controlling a complicated mechanical system to perform a certain task, for example making robot to dance, is a traditional problem studied in the field of control theory. Many successful applications like Google BigDog [RBNP08] and Google Self-driving car [Gui11] have been made in accordance with the new theories found in this field. Evidence shows that incorporating machine learning techniques in robotics can allow researchers to get rid of tedious engineering work of adjusting environmental parameters. Many researchers like Jan Peters, Sethu Vijayakumar, Stefan Schaal, Andrew Ng and Sebastian Thrun are the early explorers in this field. Based on the Partial Observable Markov Decision Process (POMDP) reinforcement learning, they contributed first several algorithms that enable robot to learn to perform a certain task overtime.

Recently, one sub-field of machine learning called deep learning gained a lot of attention as a method attempting to model high-level abstractions by using model architectures composed of multiple non-linear layers [KSH12]. Several architectures of deep learning networks, such as deep belief network [HOT06], deep Boltzmann machine [SH09], convolutional neural network [KSH12] and deep de-noising auto-encoder [VLL10] have shown their advantages in specific areas. Especially, convolutional neural networks, invented by Krizhevsky outperformed all the traditional feature-based machine learning techniques in the ImageNet competition.

Based on the two trends, we noticed that a natural path of research is to use deep learning methods for controlling movements of robot. Until the end of 2014, the main focus of deep learning was mostly related to a category of robotics called perception, which deals with problems, such as Sensor Fusion [ONL13], Nature Language Processing (NLP) [CvMG14b] and Object Recognition [LLS13, HGT14]. Although considered briefly by Jürgen Schmidhuber’s team [MGW06], the other area of robotics, namely control, remains more-or-less unexplored in the realm of deep learning.

Currently, the main machine learning algorithm used for learning an action is based on reinforcement learning. Specifically, it is based on a category called policy gradient algorithm, which means the algorithm needs to directly search the policy space instead of the state-action space. In this framework, there are normally two parts, namely an actor part and a critic part. The actor part is used for generating different actions based on policy parameters, the critic part is for simulating the
Deep Learning can be used in this case as it is a general function approximator and environment is considered as a function that takes several parameters in and outputs an action. One class of deep learning structure called Recurrent Neural Network (RNN) has been proven to be effective for learning temporal information from data [Gra13a, GMH13, GWD14]. However, when it is applied to robotics, the widely used model called Long-Short Term Memory (LSTM) [HS97] suffers slowness of learning form environment.

In the thesis, we consider several kinds of deep learning models to approximate the environment to reduce the real word samples needed. As a consequence, two major aspects are considered. One focus of this thesis is to introduce general learning methods for the robot control problem with an emphasize on deep learning method. This thesis tries to describe the traditional learning methods as well as the emerging deep learning methods for the robot control problem. Another focus of this thesis is to show the main contribution of author to this field. With experiments, the author is able to show his own RNN structure called Simple Gated Unit (SGU) can outperform the previous structure (e.g. LSTM, GRU) on sequence classification tasks, which creates a potential for robot learning tasks that heavily require learning sequential data.

2 Reinforcement Learning

If we would like to discuss what might be the most common way of learning, learning based on interacting with our environment is a natural idea to think about. When we were born in this world, we had no teachers around us. But tens of years passed, we learned to fear, to communicate with others and to write a paper. As a consequence, it is very natural to think that our environment is a great source of information. While playing around with environment, we learn by taking actions and getting reward from it. Now when we cook, when we do exercise, we are fully aware of what the response of environment will be.

RL is an area that studies the mechanism of this kind of learning in a computational way. Generally, the goal of RL is to find a way of mapping different states with different actions so that we could maximize the reward signals.

There are two main approaches in the area of reinforcement learning, one is based on Markov Decision Process (MDP)[Bar98] and another one is recurrent neural network
Both methods have advantages and drawbacks when applied to robotics.

In the following sections of this chapter, we may consider robot as an agent in all descriptions of related techniques.

2.1 Markov Decision Process

Markov Decision Process (MDP) is a discrete time stochastic control process. We may consider a robot in a state \( s \) of discrete state space \( \mathcal{S} \). The robot can take an action \( a \) in all possible action set \( \mathcal{A} \) resulting in a state \( s' \). We can denote this process as a transition function \( P_a(s, s') \) meaning the probability of moving from state \( s \) to state \( s' \) through an action \( a \). Then after the robot executes action \( a \) which results in \( s' \), it will receive a reward \( r \) according to a reward function denoted as \( R_a(s, s') \). The goal of reinforcement learning is to optimize cumulative reward of the whole process.

The problems of MDP is more clear to researchers as it is based on mathematical formalizations. On one hand, MDP-based methods together with optimization methods, such as Gradient Partially Observable Markov Decision Processes (GPOMDP) [BB01], projection method or nature gradient are state-of-the-art in robot trajectory learning. On the other hand, as data collected from robot is different from other types of data, it was pointed out the MDP-based methods suffers from several curses, [KBP13] including:

- Curse of Dimensionality
- Curse of Real-World Samples
- Curse of Under-Modeling and Model Uncertainty
- Curse of Goal Specification

We will explain in more detail in section 2.3. The data is normally high-dimensional, continuous and erroneous data in robot systems. It is considered to be a difficult question to neglect these issues and it is also hard to specify the goal of the system i.e. what robot needs to be.
2.1.1 Partially Observable Markov Decision Process

A Partially Observable Markov Decision Process (POMDP) \cite{Son71} is a generalization of a Markov decision process (MDP). A POMDP models an agent decision process in which it is assumed that the system dynamics are determined by an MDP, but the agent cannot directly observe the underlying state. Instead, it must maintain a probability distribution over the set of possible states, based on a set of observations and observation probabilities and the underlying MDP.

The POMDP framework is general enough to model a variety of real-world sequential decision processes. Applications include robot navigation problems, machine maintenance and planning under uncertainty in general. The framework originated in the operations research community, and was later taken over by the artificial intelligence and automated planning communities.

An exact solution to a POMDP yields the optimal action for each possible belief over the world states. The optimal action maximizes (or minimizes) the expected reward (or cost) of the agent over a possibly infinite horizon. The sequence of optimal actions is known as the optimal policy of the agent for interacting with its environment.

More precisely, a POMDP is a discrete time stochastic control process. We may consider a robot in a state $s_t$ of discrete space $S$ where $t$ means iteration number. Now, the robot can take an action $a$ in all possible action set $A$ resulting in a state $s_{t+1}$. We can denote this process as a transition function $T_a(s_t, s_{t+1})$ meaning the probability of moving from state $s_t$ to state $s_{t+1}$ through action $a$. Then after the robot executes action $a$ and results in $s_{t+1}$, it will receive a reward according to reward function denoted as $R_a(s_t, s_{t+1})$.

Formally, we define MDP as follows:

A Markov decision process is a 4-tuple denoted as $(S, A, T \cdot (\cdot, \cdot), R \cdot (\cdot, \cdot))$ In this tuple

- $S$ is a finite set of states. It describes all possible states of the space.
- $A$ is a finite set of actions. It describes all possible states that the robot can take.
- $T \cdot (\cdot, \cdot)$ is a function that takes three arguments, e.g $T_a(s_t, s_{t+1})$ means probability of transferring from $s_t$ to $s_{t+1}$ with action $a_t$.
- $R \cdot (\cdot, \cdot)$ is also a function that takes three arguments, e.g $R_a(s_t, s_{t+1})$ means
reward of transferring from $s_t$ to $s_{t+1}$ with action $a_t$.

The main problem of reinforcement learning is to find a policy function $\pi(s) : s \rightarrow a$ to map every state $s$ with action $a$ so that a cumulative reward $R$ is maximized, where $0 \leq \gamma \leq 1$ is a discount factor. As the discount factor generalizes discounted situation and undiscounted situation, it broadens our theory.

2.1.2 Markov Decision Process with Continuous States

Now if we consider the environment of robots, we need some modifications to the original POMDP. First, we still assume the process to be discrete time but we consider continuous space $\mathbb{S} \subseteq \mathbb{R}^n$ and continuous action set $\mathbb{A} \subseteq \mathbb{R}^m$, where $n$ is dimensionality of space and $m$ is dimension of actions. For the initial state $s_0$, we assign a distribution $p(s_0)$, where $s_0 \in \mathbb{S}$. At any state $s_t \in \mathbb{S}$, we have a continuous policy $\pi(a_t|s_t) = p(a_t|s_t, \theta)$ parametrized by $\theta$. Transfer function now also becomes continuous. It corresponds to a probability distribution $T_{a_t}(s_t, s_{t+1}) = p(s_{t+1}|s_t, a_t)$. After this step is completed, the process will generate a reward function $R_{a_t}(s_t, s_{t+1})$ which is defined as $R : \mathbb{A} \times \mathbb{S} \rightarrow [0, \infty)$. After these modifications, we can now formalize continuous MDP.

Continuous Markov Decision Process with Infinite States (CMDPIS) [Mil68] is a modified version of ordinary POMDP. Mathematically, it is defined as a 5-tuple $(P_{init}, S, A, T \cdot (\cdot, \cdot), R \cdot (\cdot, \cdot))$ where

- $P_{init}$ is a initial distribution of the states.
- $\mathbb{S} \subseteq \mathbb{R}^n$ is a infinite set of states. It describes all possible states of the space.
- $\mathbb{A} \subseteq \mathbb{R}^m$ is a infinite set of actions. It describes all possible actions of the agent.
- $T \cdot (\cdot, \cdot)$ is a function that takes three arguments. e.g $T_{a_t}(s_t, s_{t+1})$ means probability of transferring from $s_t$ to $s_{t+1}$ with action $a_t$.
- $R \cdot (\cdot, \cdot)$ is a function that takes three arguments. e.g $R_{a_t}(s_t, s_{t+1})$ means reward of transferring from $s_t$ to $s_{t+1}$ with action $a_t$.

With this continuous setting, we have objective function defined as follows:
\[
J(\theta) = E_{\tau}\{(1 - \gamma)\sum_{t=0}^{\infty} \gamma^t R_t|\theta\} \\
= \int_{S} d^\pi(s) \int_{A} \pi(\bar{a}|s) R(\bar{a}, s)d\bar{a}ds,
\]

where \(d^\pi(s)\) refers to \(1 - \gamma^t p(s = \bar{s})\), \(0 \geq \gamma \geq 1\) refers to discount factor and \(\pi(\bar{a}|s)\) is parametrized by \(\theta\).

### 2.1.3 Value Functions

We define two functions to further describe this process. First function is a state value function denoted as \(V^\pi(\bar{s})\). This means expected value of an agent that follows a policy \(\pi\) with initial value \(\bar{s}\). It characterizes the rewards of following a policy \(\pi\). Mathematically, it is defined as:

\[
V^\pi(\bar{s}) = E_{\tau}\{\sum_{t=0}^{\infty} \gamma^t r_t|s = s_0\},
\]

where \(\tau\) stands for the trajectory of the agent.

\[
Q^\pi(\bar{s}, \bar{a}) = \bar{E}_{\tau}\{\sum_{t=0}^{\infty} \gamma^t r_t|s = s_0, a = a_0\},
\]

where \(Q^\pi(\bar{s}, \bar{a})\) is named Q function that map \(\bar{s}\) and \(\bar{a}\) to a Q-value.

State-value function only depends on the first state of the agent. After that the system is governed by policy \(\pi\). Another value function we need to define is state-action value function.

### 2.1.4 Natural Actor Critic Model

The first thing to demonstrate in Natural Actor-Critic is to explain the term. Natural stands for gradient framework called natural gradient, while Actor-Critic means an iterative method of evaluating and improving the objective function.

Robot learning is based on Markov Decision Process with discrete time and continuous states. The objective function is defined in Section 2.1 by Formula 1. A normal gradient of objective function is defined as:

\[
\nabla J(\theta) = \int_{S} d^\pi(s) \int_{A} \nabla_\theta \pi(\bar{a}, s) R(\bar{a}, s)d\bar{a}ds
\]
Please note $\theta$ is related to $\pi(\bar{a}|\bar{s})$ as it is defined by $\pi(\bar{a}|\bar{s}|\theta)$.

However, here we need to redefine this gradient as vanilla gradient. As pointed out by Shun-ichi Amari [Ama98], another kind of gradient called natural gradient is more efficient in many machine learning application than the vanilla gradient. Mathematically we define following formula as natural gradient:

$$\nabla J(\theta) = G^{-1} \nabla J(\theta), \quad (5)$$

where $G$ is a fisher information metrix [Ama98]. Fisher information matrix is a matrix defined on Romanian space. This metrix is interesting on several aspects. One of them is that it shows true direction of a function’s steepest direction while the vanilla gradient does not. The long proof of the previous statement is based on showing natural gradient descent method is Fisher efficient. We recommend a further reading of Amari’s paper [Ama98] for more details.

We rewrite Formula 4 as:

$$\nabla \theta J(\theta) = \int_{\mathcal{S}} \int_{\mathcal{A}} \nabla_\theta \pi(\bar{a}\bar{s})(Q^\pi(\bar{a},\bar{s}) - b^\pi(\bar{s}))d\bar{a}d\bar{s}, \quad (6)$$

where $Q^\pi(a,s)$ is state-action value function and $b^\pi(x)$ is the baseline. [Ama98] demonstrates why $R(s,a)$ is replaced by $Q^\pi(a,s) - b^\pi(x)$. It can be shown that $R(s,a)$ is further approximated by $(\nabla_\theta \log \pi(a,s)^\pi(s))^T \bar{w}$ and parametrized by $\bar{w}$.

2.2 Reinforcement Learning Methods

In this section, we introduce common ideas in robotics including temporal difference learning, Episodic learning and policy gradient method.

2.2.1 Policy Evaluation

Policy evaluation is a process that computes state-value function based on a policy $\pi$. Generally, there are three ways of doing policy evaluation.

The simple every-visit Monte-Carlo (MC) method used for evaluating policy is defined as:

$$V^\pi(\bar{s}) \leftarrow V^\pi(s_t) + \gamma(R(s_t) - V(s_t)), \quad (7)$$

where $V^\pi(s_t)$ is the state-value of state $s_t$, $\gamma$ is a discount factor and $R(s_t)$ is the reward of state $s_t$ [SB98a].
The most well-known one is based on dynamic programming. It computes the best state-value for each state and iteratively updates the value of each possible state. This update process can be defined as:

\[ V^\pi(\vec{s}) \leftarrow \mathbb{E}\{r_{t+1} + \gamma V(s_{t+1})\}, \]  

(8)

Temporal Difference (TD) learning is another important concept in the area of reinforcement learning. It uses temporal difference information to learn a value function. As a result, it does not need to know every state.

As we can see, the simple every-visit Monte-Carlo method only incooperates information of the current state. However, considering future information may result in a better convergence rate.

Based on this idea, the simplest Temporal Deference (TD) method (TD(0)) is developed and defined as:

\[ V^\pi(\vec{s}) \leftarrow V^\pi(s_t) + \gamma (r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \]  

(9)

We may notice that each method requires different kind of data to be used. The simple every-visit Monte-Carlo method only incooperates information about the current state, so it can be updated just by using the information observed at each state. The TD(0) requires successive two states and dynamic programming requires information of every state.

We may further consider the differences between these three methods based on characteristics they have. Generally, bootstrapping means using estimate of successor state in reinforcement learning. In this case, we can classify these three methods as follows:

- MC does not bootstrap, as it always uses current state for updating
- DP bootstraps, as it needs to calculate expectation of all the successive states.
- TD(0) bootstraps, as it needs to know the state-value function of next state.

If we consider whether they use sampling method or not, we also can classify them as follows:

- MC samples
Algorithm 1: Tabular TD(0) policy evaluation algorithm

Require: Policy π Initialize $V(s)$ arbitrarily;

while until $s$ is terminal (for each episode) do

    $a \leftarrow$ action given by π for state $s$;

    take action $a$, observe reward $r$ and next state $s'$

    $V^{π}(\vec{s}) \leftarrow V^{π}(s_t) + \gamma (r_{t+1} + \gamma V(s_{t+1}) - V(s_t))$ $s \leftarrow s'$

• DP does not sample

• TD samples

Now I will use TD(0) policy evaluation algorithm (shown as algorithm: 1.) as an example to show how a policy is evaluated.

2.2.2 Policy Gradient Methods

The previous policy evaluation algorithms update state-value for each state. However, there are also episodic algorithms to evaluate policy based on parameters. In this thesis, we introduce a policy gradient method for robotics.

Reinforcement learning is probably the most general framework where such robot learning problems can be phrased. Despite the fact that many reinforcement learning algorithms fail to scale where robots have more degrees of freedom, policy gradient methods is one of the exceptions.

There are several advantages of policy gradient algorithms. According to Jan Peters’ paper about policy gradient method [PS06a], firstly the policy representations can be chosen to be meaningful. Secondly, the parameters can incorporate previous domain knowledge. The third reason is that policy gradient algorithm has a rather strong theoretical underpinning and additionally, policy gradient algorithms can be used in a model-free fashion.

All these advantages ensure that with only a few parameters, robots can learn a decent policy for certain task. Mathematically, a policy gradient algorithm tries to optimize policy parameters $\theta \in \mathbb{R}^n$ so that the expected return

$$J(\theta) = \mathbb{E}\left\{ \sum_{k=0}^{H} a_k r_k \right\}$$ (10)
is maximized. In Equation 10, $a_k$ is a weight factor, $H$ is all possible histories. It can be set as $\gamma^k$ in discounted case or $\frac{1}{H}$ in the average case. The steepest decent algorithm is normally set to be an optimization method as for each iteration, we would like to have small changes to the robot system.

$$\theta_{h+1} = \theta_h + \alpha_h \nabla_{\theta} J|_{\theta = \theta_h},$$  
(11)

where $\alpha_h$ is the learning rate for current update step $h$.

### 2.3 Classification of RL Problems

Most literature pointed out the complexity of RL problems in the area of robotics, [KBP13, PS06a]. In Kober’s paper [KBP13], four different aspects were mentioned. He considers these four aspects as four curses when applying RL to robotics. These four curses are the curse of high dimensionality, the curse of real world samples, the curse of under modeling, and model uncertainty, and the curse of goal specification.

**High-Dimensionality** is the first characteristic considered in the area of robotics. Robot systems normally have many degrees of freedom (DOF), especially in modern anthropomorphic robots. For example, the Baxter robot has two arms, each arm has 7 DOF, including three pitch degrees and four roll degrees. This continuity makes traditional reinforcement learning fail as many traditional methods are based on discretization of each DOF. If we discretize $n$ DOFs and discretize each DOF to $m$ states, the total states for the system is $m^n$, which is inapplicable in most of the cases.

**Need of Real Samples** is another curse of applying RL to robotics. Robot systems inherently interact with physical system in real world. During test with the environment, robot hardware may experience wear and tear. As a consequence, in many cases, this is an expensive process. Failure is costly so the test process also requires some kind of supervision from a human. For example, we used Baxter to reach certain positions as fast as possible. In the optimization process, it opt stuck in a position that is hard to set. If the optimization process happens in a more complex dynamic environment e.g. helicopter robot, a supervision process conducted by several people is needed. For all these reasons, generating real world samples requires different resources and is an expensive process.

**Under-modelling and model uncertainty** is the next problem in robot systems. In order to reduce the cost of real world samples, researchers build accurate simulators to accelerate the learning process. Unfortunately, building this kind of models
involves a lot of engineering work, which is also expensive. For small robot system, the simulator can improve the learning process to some extent. But if we use a simulator to simulate a complex system, a small turbulence can cause the learned system to diverge from real system.

Last but not the least, **goal specification** means specifying the reward function for the robot system. In a reinforcement learning algorithm, the policy optimization process depends on observing different rewards of two different policies. If same reward is always received, there is no way of telling which policy is better. In practice, it is surprisingly difficult to specify the reward function of the system.

These four areas are notorious when trying to apply RL algorithms to robotics. Here we only discuss basic ideas of these problems. However, researchers who study applying reinforcement learning in robotics have explored these problems more thoroughly than discussed here. If readers have an interest in these issues, please refer to paper "Reinforcement learning in robotics: A survey" [KBP13].

### 2.4 Policy Gradient with Parameter Exploration

As one of the policy gradient methods, Policy Gradient with Parameter Exploration (PGPE) has shown its advantages in several scenarios. This algorithm was developed by Frank Sehnke and described in his paper "Policy Gradient with Parameter Exploration" [SOR+08]. PGPE algorithm follows the basic idea of policy gradient algorithm, that is optimizing policy without using value estimation.

#### 2.4.1 PGPE algorithm

As previously mentioned, the policy uses several policy parameters $\theta$ to represent the policy of the system. In the case of PGPE, the policy is represented as a simple linear model. For formalizing the algorithm, we will uses previous symbols as a basis of the system.

Consider a robot interacting with environment at time $t$, the robot is in a state $s_t$, making action $a_t$, and which results in state $s_{t+1}$ according to a stochastic function $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$. Now with parameter $\theta$, the action is defined as $a_t \sim p(\tilde{a}_t|\tilde{s}_t, \theta)$. In PGPE, the action is determined by $n$ weight matrices $(W_i, \forall i \in \{1, 2, \cdots, n\})$. 
Mathematically, the action vector $\vec{a}$ is determined by:

$$\vec{a} = f(\vec{s}) = \sum_i^n \vec{W}_i \cdot \vec{s}, \quad (12)$$

where $f(\vec{s})$ is defined as $f(\vec{s})$. Sometimes, it is also called a multi-layer linear neural network.

Each parameter in matrix $W_{ijk}$ is defined by a Gaussian function:

$$W_{ijk} \sim \mathcal{N}(\mu, \sigma^2) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \quad (13)$$

where $\mu$ and $\sigma$ are mean and variance of the algorithm, $W_{ijk}$ means element of $j’$th row and $k’$th column of $i’$th matrix.

The learning process of PGPE is a little bit different to other algorithms. Previously, the objective function was defined according to Formula 4. However, in the episodic setting, the parameters are updated after each episode, which makes the algorithm dependent on sampling of all parameters of this linear neural network. After sampling parameters for all the weights of network, we receive a sequence of all the rewards $r_t$ in this episode, which results in a reward for this episode to be $r(h) = \sum_{t=1}^T r_t$. We then modify the algorithm to be:

$$J(\theta) = \int_H p(h|\vec{\theta}) r(h) dh, \quad (14)$$

where $H$ is the set of all possible histories $h$ of this episode, $h$ is defined as $h = [s_1:T, a_1:T]$.

Further expanding the formula by using standard identity $\nabla_x y(x) = y(x) \nabla_x \log(x)$, we get:

$$\nabla_{\vec{\theta}} J(\vec{\theta}) = \int_H p(h|\vec{\theta}) \nabla_{\vec{\theta}} \log p(h|\vec{\theta}) r(h) dh \quad (15)$$

Since the whole process is Markovian, i.e. we have $a_i \sim p(a_i|s_i, \vec{\theta})$ defined as the transition function, the following property holds:

$$p(h|\vec{\theta}) = \prod_{i=1}^T p(s_{i+1}|a_i, s_i)p(a_i|s_i, \vec{\theta})p(s_0), \quad (16)$$

$$\log(p(h|\vec{\theta})) = \sum_{i=1}^T \log(p(s_{i+1}|a_i, s_i)p(a_i|s_i, \vec{\theta})) + \log(s_0) \quad (17)$$
The term $\nabla_\theta \log p(h|\theta)$ in Formula 15 can be rewritten as:

$$\nabla_\theta \log p(h|\theta) = \nabla_\theta \sum_{i=1}^{T} \log(p(\tilde{s}_{i+1}|\tilde{s}_i, \tilde{\theta})) + \nabla_\theta \log(p(\tilde{s}_0))$$ \hspace{1cm} (18)

$$= \nabla_\theta \sum_{i=1}^{T} \log(p(\tilde{s}_i, \tilde{s}_i|\tilde{a}_i, \tilde{s}_i,p(\tilde{a}_i|\tilde{s}_i, \tilde{\theta}))) \hspace{1cm} (19)$$

$$= \nabla_\theta \sum_{i=1}^{T} \log(p(\tilde{a}_i|\tilde{s}_i, \tilde{\theta})) \hspace{1cm} (20)$$

Then, if we substitute the result of Formula 20 into Formula 15, we get more convenient form:

$$\nabla_\theta J(\theta) = \int_H p(h|\theta) \nabla_\theta \sum_{i=1}^{T} \log(p(\tilde{a}_i|\tilde{s}_i, \tilde{\theta})) r(h) dh$$ \hspace{1cm} (21)

In the continuous case, it is impossible to get all the histories for a given set of policy parameters. We need to estimate the gradient based on the samples of this distribution as a result, we further modify our formulas to:

$$\nabla_\theta J(\theta) = \frac{1}{P} \sum_{p=1}^{P} \sum_{i=1}^{T} \nabla_\theta \log(p(\tilde{a}_i|\tilde{s}_i, \tilde{\theta})) r(h_p), \hspace{1cm} (22)$$

where subscript $p$ means $p$'th sample of the distribution.

Previously, we discussed that in PGPE, each parameter of matrices is determined by a Gaussian function as defined in Formula 13. As we use weights matrices $W_i$ as hidden policy parameters, the Formula 21 becomes:

$$\nabla_{\mu, \sigma^2} J(\theta) = \int_H p(h|\theta) \nabla_\theta \sum_{i=1}^{T} \log(\int_\theta p(\tilde{a}_i|\tilde{s}_i, \tilde{\theta}) p(\tilde{\theta}|\mu, \sigma^2)) r(h) dh$$ \hspace{1cm} (23)

According to this formula, we can make an optimization process for both vectors of means and variances of Gaussian distribution. There are two ways of optimizing the function. One way is to calculate the gradient information of each mean $\mu_i$ and gradient information of each $\sigma$ with respect to $\log(p(\theta|\mu, \sigma^2))$. From definition of Gaussian function we can get:
\[ \nabla_{\mu} \log(p(\theta|\bar{\mu}, \bar{\sigma}^2)) = \nabla_{\mu} \log\left(\frac{1}{\sigma \sqrt{2\pi}} \exp\left(-\frac{(\theta - \mu)}{2\sigma^2}\right)\right) = \nabla_{\mu} - \frac{-(\mu - x)^2}{2\sigma^2} = \frac{\mu - x}{\sigma^2} \quad (24) \]

and

\[ \nabla_{\sigma} \log(p(\theta|\bar{\mu}, \bar{\sigma}^2)) = \nabla_{\sigma} \log\left(\frac{1}{\sigma \sqrt{2\pi}} \exp\left(-\frac{(\theta - \mu)}{2\sigma^2}\right)\right) = -\frac{1}{\sigma} + \frac{(x - \mu)^2}{\sigma^3} = \frac{(x - \mu)^2 - \sigma^2}{\sigma^3} \quad (25) \]

After getting gradient information of mean \(\mu\) and variance \(\sigma^2\) of the system, we can use the method proposed by Williams [Wil92]. Instead of knowing the exact step size, Williams uses a so-called reference reward to calculate the step size. Formulae 26 and 27 shows the updating rules of mean and variance.

\[ \Delta \mu = \alpha(r - b)\frac{\mu - x}{\sigma^2}, \quad (26) \]

\[ \Delta \sigma = \alpha(r - b)\frac{(x - \mu)^2 - \sigma^2}{\sigma^3} \quad (27) \]

This method was influenced by Williams’ REINFORCE algorithm and as a consequence it inherits the advantages and disadvantages of this algorithm. On one hand, this method has proof of convergence, which means that with more trials this method will finally converge. On the other hand, this method is also slow and unstable. A better solution based on Simultaneous Perturbation Stochastic Approximation (SPSA) [BPP12] provides a faster convergence rate.

This method uses a symmetric sampling process for determining rewards of a specific mean \(\mu\) and variance \(\sigma\) of random variable. Using SPSA, we first generate a small perturbation \(\epsilon\) from normal distribution \(N(0, \sigma^2)\). Then, we use two parameters \(\bar{\theta}^+ = \bar{\mu} + \epsilon\) and \(\bar{\theta}^- = \bar{\mu} - \epsilon\) for the system to get two rewards \(r^+\) and \(r^-\). Combining it with Formula 24 and Formula 25, we can get gradient information for \(\bar{\mu}\) as:

\[ \nabla_{\mu} J(\bar{\mu}, \bar{\sigma}^2) \approx \frac{\alpha \epsilon (r^+ + r^-)}{\sigma^2} \quad (28) \]
Then, we use the central difference method and apply the same step size and the update rule becomes:
\[ \Delta u_i = \alpha \epsilon \left( \frac{r^+ - r^-}{2} \right) \] (29)

However, we cannot use a similar method for updating \( \sigma \), since the selected two parameters \( \theta^+ \) and \( \theta^- \) have the same variances. As a consequence, instead of using information from the sample parameter \( \epsilon \), we also consider the average value of two rewards to update \( \sigma \). This updating algorithm is called SyS sampling method in some literature [Seh13]:
\[ \Delta \sigma_i = \alpha \left( \left( \frac{r^+ + r^-}{2} - b \right) \left( \frac{\epsilon_i^2 - \sigma_i^2}{\sigma_i} \right) \right), \] (30)
where \( b \) is the baseline and is updated before updating the hidden policy parameters \( \bar{\mu} \) and \( \sigma^2 \).

Formula 31 shows how the baseline is updated. Literature mentions that the variance of the base can affect quite a lot the learning speed of the system [ZHNS11], but here we only use the simplest way of updating the algorithm.
\[ b = 0.1 \cdot b + 0.9 \cdot \frac{r^+ + r^-}{2}, \] (31)

In the following section, we use pseudo code for the whole PGPE algorithm. (Presented in Algorithm 1)

**Algorithm 2:** Policy Gradient with Parameter Exploration (PGPE)

**Data:** The MDP Process Parameters

**Result:** Optimal Policy \( \pi^* \)

1. while average reward not converged do
2. Get mean \( \bar{\mu} \) and variances \( \bar{\sigma}^2 \) of random variables.
3. For each parameter, sample \( \epsilon \) from Normal Distribution \( \mathcal{N}(0, \sigma^2) \) and form a vector \( \bar{\epsilon} \).
4. Get two parameters using \( \bar{\theta}^- = \bar{\mu} - \bar{\epsilon}, \bar{\theta}^+ = \bar{\mu} + \bar{\epsilon} \)
5. Use two parameters as policy parameters, get cumulative reward \( r^+, r^- \) from system.
6. Update baseline \( b \) of algorithm according to Formula 31.
7. Update \( \mu \) and \( \sigma \) using Formula 29 and Formula 30.
3 Deep Recurrent Neural Networks

Recurrent Neural Network (RNN) is a special structure of neural network that has recurrent connections. Deep recurrent neural networks are neural network models that have more than one hidden layer. In the following sections, we will discuss details of this kind of model.

3.1 Deep Learning and its Recent Advances

Deep learning has attracted a lot of attention since 2006. In 2006, Geoffrey Hinton, one of the founders of the idea of deep learning, published a paper called "Reducing the Dimensionality of Data with Neural Networks" [HS06]. In this paper, Hinton and Slakhutdinow showed how a many-layered feedforward neural network can be pre-trained layer by layer. Since this paper, the phrase "deep learning" become famous in the community. However, before Hinton’s work, several deep learning algorithms were developed. One of the initial work belongs to Kunihiko Fukushima who invented a model called Neocognitron [Fuk75] in 1980. After that, in 1989, Lecun was able to use an optimization algorithm called back-propagation to train a neural network. Then, Yann Lecun further simplified network to be the so-called Convolutional Neuron Network(CNN) [LGTB97] in 1998. As CNN became very successful in the area of computer vision and so did the popularity of deep learning. Other models that have more than one hidden layer are also usually discussed in the context of deep learning. For example, Deep Belief Network (DBN) [HOT06] and Deep Boltzmann Machine (DBM) [SH09] are not normally considered as neural network models, but they also have application in the area of computer vision.

3.2 Feedforward Neural Networks

In 1957, the simplest structure of a neural network called perceptron was introduced by Frank Rosenblatt in Cornell’s Aeronautical Laboratory [Ros58]. This model consists of only one cell and multiple connections. Figure 1 shows the basic structure of the perceptron.

The formula of representing the input and the output is defined as:

\[ y = \vec{w} \cdot \vec{x} + b \]  

(32)
The $i$th input is shown as $x_i$ and the corresponding weights are denoted as $w_i$. There is also a bias term $b$ connected to the cell, which, with an addition function, generates the output.

If we also consider $b$ as one of the inputs, then the formula can be defined as:

$$y = \vec{w}_{\text{new}} \cdot \vec{x}_{\text{new}},$$

where $\vec{w}_{\text{new}} = [\vec{w}; b]$ and $\vec{x} = [\vec{x}; 1]$.

This simple structure was considered promising initially from several points of view, but after further investigation, the perceptron was shown unable to classify many non-linear classes of patterns. However, its discovery led to a field of research called neural networks in the area of artificial intelligence. As a consequence, since 1957, researchers started trying different methods to modify this model to adapt to different problems. One important modification is to add a non-linear transformation function to the system i.e. after getting $y$ from Formula 3.2, we use a function like $\tanh$ to get a new $\hat{y}$ to ensure the output is restricted within a range. Another important modification of the system is to stack many perceptrons together to build a large and complex model for the classification purpose. This kind of networks is normally called Feed-forward Neural Networks (FFNN) [HSW89] as the information send to this system is propagated only from lower layer to higher layer. Figure 2 shows the structure of FFNN.

The formula describing each layer is then defined as:

$$f(\vec{x}) = \sigma(W \vec{x}),$$
3.3 Recurrent Neural Networks

Recurrent Neural Network (RNN) contains at least one neuron that has at least one recurrent connection i.e. a connection that connects to itself or to a lower layer. This special structure makes memory cell to be the internal memory, which enables the network to memories the change of sequential data. Unlike FFNN, the recurrent neural network is used for predicting the next data point.

3.3.1 Finite Unfolding in Time

When considering the structure of a neural network, researchers usually apply finite unfolding in time for RNN. It means introducing another dimension for RNN. In this way, the recurrent connection can be dealt with more easily. In the following section, we will use a simple example to explain this idea. The model we are going to use contains one input neuron, one hidden neuron and one output neuron.

In this figure, we mark the input layer as $\mathbf{\vec{x}}$, the hidden layer as $\mathbf{\vec{s}}$ and the output...
Figure 3: The structure of an recurrent neural network. There are two layers in this network, namely input layer and hidden layer, the first layer contains notes connected to the input data. The second layer stores the information and also forwards this information to the next layer. The cell in the hidden layer has a connection to itself, which means the information stored in the cell at time $t-1$ also influences the information stored in the cell at time $t$.

Figure 4: The structure of a simple RNN. It contains three layers i.e. one input layer, one hidden layer and one output layer.

layer as $\vec{o}$. The horizontal direction is the direction of time.

The basic idea of unfolding an RNN in time is to copy a RNN several times and connect it in a chronological order. If the connection is recurrent, then the connection
should be made to the same neuron in the next time step. Figure 5 illustrates the model of unfolding a simple RNN in three time steps.

![Diagram of RNN model](image)

Figure 5: The model of unfolding a simple RNN described in Figure 4 in \( n \) time steps. \( \vec{x}_t \) represents the input layer at time \( t \), \( \vec{s}_t \) representes the hidden states at time \( t \) and \( \vec{o}_t \) represents the output layer at time \( t \).

The finite unfolding technique transforms a neural network with a recurrent connection to a network that makes it easier to compute the gradients. It is also easy for us to write this neural network’s expression:

\[
\vec{s}_t = \sigma(W\vec{x}_t + B\vec{s}_{t-1}),
\]  

(35)

where \( W \) is the weight matrix that transfers input neurons to hidden norons. \( \vec{s}_t \) is the value of the hidden state at time \( t \) and \( B \) is the weight matrix for updating the hidden state information from time \( t - 1 \) to \( t \).

### 3.3.2 Overshooting

Considering that we only have one time series, the task for RNN is to predict the next data point based on the previous data that we have. There are two steps for solving this problem. First, we need to copy the data into two sets and shift the input by one to produce an output. It is illustrated in Figure 6.

The last training step takes \( \vec{x}(t - 2), \vec{x}(t - 1), \vec{x}(t) \) as input and produces \( \vec{x}(t - 1), \vec{x}(t), \vec{x}(t + 1) \) as output. However, it is very difficult to predict the value at time \( t + 2 \) as we do not have information about \( \vec{x}(t + 1) \). According to Formula 35, the input is set to \( \vec{0} \) \( \forall t > T \) so we get:
3.3.3 Dynamical Consistency

Dynamical consistency is kept by introducing the output of the last time step to the input of the current time step. Figure 7 illustrates how it is done through modification of the structure.

By remembering all the parameters in the network and unfolding in time for several steps, the recurrent neural network is able to predict the next data point in the sequence. The relationship between the input and the output is listed as follows:

\[
\tilde{s}_t = \sigma(W\tilde{x}_t + Bs_{t-1}) \\
\tilde{x}_{t+1} = As_t, 
\]

where \( A \) is the weight matrix of internal states to output states.

To sum up, the dynamics of a system can be described as follows.
Inputs: $\vec{x}_{t-2}$ $\vec{x}_{t-1}$ $\vec{x}_t$ $\vec{x}_{t+1}$

Hidden States: $\vec{s}_{t-2}$ $\vec{s}_{t-1}$ $\vec{s}_t$ $\vec{s}_{t+1}$

Outputs: $\vec{x}_{t-1}$ $\vec{x}_t$ $\vec{x}_{t+1}$ $\vec{x}_{t+2}$

Figure 7: The model for predicting data for time series. For the first unfolded structure of a neural network, the data of series at time $t-2$ is fed to the network and we expect data at time $t-1$ to be predicted by the network.

\[ \vec{s}_t = \sigma(W\vec{x}_t + B\vec{s}_{t-1}) \quad \forall t \leq T \]  
\[ \vec{s}_t = \sigma(W\vec{x}_t + B\vec{s}_{t-1}) \quad \forall t > T \]  
\[ o_{t+1} = A\vec{s}_t \quad \forall t \leq T - 1 \]  
\[ \vec{x}_{t+1} = A\vec{s}_t \quad \forall t > T - 1, \]  

Formula 39 shows how the values of internal states are updated at the training stage. Formula 40 shows how the values of the output data are predicted after the training stage. Formula 41 shows how the outputs are generated after the training stage. Formula 42 shows how the outputs are predicted after the training stage. The overall goal of the system is to minimize the difference between the output and the predicted output for all time series. Mathematically, it is defined as:

\[ J = \sum_{i=1}^{T} \vec{o}_i - \vec{y}_i \]  

If we assume each input and output has $N$ data points, then the cost becomes:

\[ J = \sum_{i=1}^{T} \sum_{n=1}^{N} o_{i,n} - y_{i,n} \]  

The main goal of the function is to minimize the cost over function parameters defined in the network. See section 3.6 for more references.
3.4 Universal Approximation

It has been proven that multi-layer feed-forward neural networks are universal approximators by Hornik in 1989 [HSW89]. Similar works was also mentioned by Cybenko and Funahashi in the same year. In this work, Hornik continued the work of Minsky and Papert about two layers network. Minsky and Papert proved that two-layer neural networks are not able to approximate functions that do not belong to a special class. Then, by adding a third layer, the neural network is able to approximate different functions.

Details of the approximation theory of multi-layer neuron network is governed by Stone-Weierstrass theorem [Cot89]. This theory states that every continuous function defined on a closed interval $[a, b]$ can be uniformly approximated by a polynomial function. As a result, it also proves that a neural network with more than one hidden layer can approximate any function.

RNN, as special model of neuron network model also follows Stone-Weierstrass theorem. Herrn Anton Maximilian Schäfer proved that RNN is also a universal approximator [SZ06].

3.5 Learning Long-Term Dependencies

Long-term dependency is an important concept in control. Some states of the system can be crucial to the system. They might influence the future behaviour of the system in many ways. For example, in the T maze example (see Figure 8 for reference), the decision at the turning point is essential for the system. As the behaviour is totally different after the turning point, a long term memory is important in this case.

Traditional RNN has several problems. One famous problem is that it suffers exploding and vanishing gradient problem as when RNN is unfolded in time, it becomes neurally deep. For example, the gradient becomes very small as we back-propagate to the first few layers. Although, there are methods like policy gradient and optimal ordered problem solver that can solve this problem, some specially designed RNN also has this ability.

However, traditional RNNs suffer exploding or vanishing gradient problems. Training of the RNN normally requires unfolding of the network in time. Compared with FFNN, it is naturally deep and as a result, the gradient becomes smaller and smaller.
Figure 8: An example of the T maze problem. State $s$ is the starting state. The agent needs to go to the end of the maze to terminate the process. At the end state 1, it gets 10 reward from the environment and at the end state 2, it gets a punishment from the environment. The agent wants to maximize the reward it gets from the environment, as a consequence, the turning state $t$ is important for it.

Soon after this problem was discovered, several new structures of RNN were developed for keeping long-term memory. Long Short Term Memory (LSTM) [HS97] is one of them.

LSTM is normally treated as a hidden layer in a neural network system. It normally adds some extra components to the network. More specifically, it has input connections, three gates and output connections. A more illustrative example is shown in Figure 17.

The idea of LSTM is relatively straightforward. The memory cell stores information of the sequential data. Gates try to remember when and how much the information in memory cell should be updated. Mathematically, the process is defined by Formulas 66 to 70.
Figure 9: The basic structure of LSTM. In the middle, there is the memory cell which keeps the information of the data sequence. Around it, there are three gates, namely the input gate, the output gate and the forget gate. Each of them gets information from the input and controls the updating rule of the memory cell. On the left side, there is input connection. On right side, there is the output connection of this layer.

\[
\tilde{i}_t = \sigma \left( W_{xi} \vec{x}_t + W_{hi} \vec{h}_{t-1} + W_{ci} \vec{c}_{t-1} + \vec{b}_i \right) \\
\tilde{f}_t = \sigma \left( W_{xf} \vec{x}_t + W_{hf} \vec{h}_{t-1} + W_{cf} \vec{c}_{t-1} + \vec{b}_f \right) \\
\vec{c}_t = \tilde{f}_t \vec{c}_{t-1} + \tilde{i}_t \tanh \left( W_{xc} \vec{x}_t + W_{hc} \vec{h}_{t-1} + \vec{b}_c \right) \\
\vec{o}_t = \sigma \left( W_{xo} \vec{x}_t + W_{ho} \vec{h}_{t-1} + W_{co} \vec{c}_t + \vec{b}_o \right) \\
\vec{h}_t = \vec{o}_t \tanh(\vec{c}_t)
\]
After understanding how the parameters in LSTM are updated, the next step is to determine what kind of optimization process should be applied to the network so that it can be easily optimized.

### 3.6 Training RNN

After getting the data and designing the structure of the network, the next important step is to train the predict network to model the data we have and also to predict next point in the data sequence. The main goal of training is to minimize the cost of the objective function between the expected outcome and real outcome. As a result, predicting future data based on the data we have.

#### 3.6.1 Backpropagation

Backpropagation, shown in Algorithm 3, is an algorithm that uses gradient information of the network to update the parameters of the system. The basic idea of this algorithm is to first use the output of system $\tilde{o}$ and real data $\tilde{y}$ to compute the error $J(\theta) = |\tilde{o} - \tilde{y}|_2$. Then, the algorithm uses this information to calculate $\Delta W_h$ of the hidden layer and to further calculate $\Delta W_i$ of the input layer. Normally, this algorithm also uses a parameter $\alpha$ to decide on the step size of the algorithm, which makes the real update rule of the parameters to be $W_{t+1} = \alpha \cdot \Delta W_t + W_t$.

---

**Algorithm 3: Backpropagation**

Require: $\alpha$: Stepsize

Require: $\delta$: Threshold

Require: $W_{1...n}$: Randomly initialized weights

while $J \leq \delta$ do

\[ \hat{o} = \text{net} \left( \hat{i} \right) \]

\[ J(W_{1...n}) = |\hat{o} - \hat{y}|_2 \]

\[ \nabla W_i = \frac{\partial J(W_i)}{\partial W_i}, \forall i \in \{1, \ldots, n\} \]

\[ W_i = \alpha \cdot \nabla W_i + W_i, \forall i \in \{1, \ldots, n\} \]

end while

return $\theta_t$ (Resulting parameters)

---

The basic gradient descent approach (and its backpropagation algorithm implemen-
(tation) is notorious for slow convergence because the learning rate $\alpha$ must be typically chosen quite small to avoid instability. Many speed-up techniques are described in the literature. As a first order method, dynamic learning rate adaptation schemes is sometimes used in the system. It has complexity $O(TM)$, where $T$ is the number of epochs and $M$ is the number of connections. Sometimes, second-order gradient descent techniques are also used to exploit curvature of the gradient but have the epoch complexity of $O(TM^2)$. These two classes of methods are mainstream in the optimization area.

However, even when using these methods, the system still suffers from local error minima, which means the optimization gets stuck at some local minimum and cannot get out. For this particular problem, there are normally three ways of solving it:

- adding noise
- repeating the entire learning from different initial weight settings
- using task-specific prior information to start from an already plausible set of weights

In the following section, we will briefly introduce two methods that are able to optimize the LSTM in practice. One of them is called Nesterov’s Accelerated Gradient (NAG) [JY09] and another is called Adam [KB14a].

### 3.6.2 Shared Weight Extended Backpropagation

Training algorithm of a recurrent neural network, called shared weight backpropagation, is introduced from feed-forward neural network backpropagation. For simplicity, we first consider a simple feed-forward neural network defined as:

$$y = \sigma(W_h \sigma(W_i \vec{x}))$$

This network only contains three layers, including input layer, hidden layer and output layer. The input layer has $n_i$ input neurons, $h_i$ has hidden neurons and output layer has $n_o$ output neurons. The data we have is a list of input-output pairs. Now, we add one more recurrent connection to the recurrent connection weights $W_t$. The formula describing the one step of the system becomes like Formula 51:

$$y = \sigma(W_h (\sigma(W_i \vec{x}) + \sigma(W_i \vec{s}_{t-1})))$$
When we unfold the network in time in more than one step, the system will become similar to the one illustrated in Figure 10. In the figure, all the connections of the network share the same weights. In this case, we need to decide how to update the weight of each parameter. Otherwise if we update each parameter according to the gradient information, all the updates will eliminate each other.

\[
\begin{align*}
\vec{x}_t - \vec{x}_{t-1} & \quad \vec{x}_t - \vec{x}_{t+1} \\
\vec{s}_t & \quad \vec{s}_{t-1} - \vec{s}_{t+1} \\
\vec{o}_t - \vec{o}_{t-1} & \quad \vec{o}_t - \vec{o}_{t+1} \\
W_i & \quad W_i \quad W_i \\
W_h & \quad W_h \quad W_h \quad W_h
\end{align*}
\]

Figure 10: The RNN network when it is unfolded in time. Each parameter is copied several times. The unfolded network shares all the weights of the original RNN.

**Algorithm 4:** Backpropagation for two layers feed-forward neural network.

**Data:** List of input-output pairs

**Result:** return the network with parameters $W$

1. initialize network weights (often small random values);
2. while training example $ex$ do
   3. prediction = neural-net-output(network, ex);
   4. actual = teacher-output(ex);
   5. compute error (prediction - actual) at the output units;
   6. compute $\Delta W$ for all weights from hidden layer to input layer;
   7. update network weights ;
3. until all examples classified correctly or another stopping criterion satisfied

The key element in this algorithm is the update rule of the system.

Optimizing a neural network is not a simple task. Researchers tried different methods since perceptron was introduced. However, not until the back-propagation algorithm was introduced, did the neuron network models have a role in the machine learning field. Currently, back-propagation is the standard of training neuron network and many optimization algorithms for neural networks are based on that. In
the following section, We will first describe the basic back propagation algorithm and then describe two optimization methods I used in the experiments. One of them is call Nesterov’s Accelerated Gradient (NAG) and another of them is called Adam.

### 3.6.3 Nesterovs Accelerated Gradient

Nesterov’s Accelerated Gradient is sometimes called Nesterov’s momentum. Here momentum refers to the Classic Momentum (CM) method which is a technique to accelerate the standard gradient descent algorithm. Let us recall the original gradient descent algorithm, which updates parameters based on a step size $\alpha$ and gradient information of objective function:

$$\Delta_{\theta} = -\alpha \nabla_{\theta} f(\theta),$$

(52)

The CM method adds a momentum coefficient $\mu$ to the system and uses this parameter to include a momentum-like behaviour in the optimization process. The updating rule for the parameters becomes influenced by the momentum as shown in Formula 53.

$$\tilde{\Delta}_{\theta t} = \mu \tilde{\Delta}_{\theta t-1} - \alpha \nabla_{\theta} f(\theta),$$

(53)

NAG, like CM algorithm, is a first order optimization algorithm. For general smooth (non-strongly) convex functions and deterministic gradient, NAG has a convergence rate of $O\left(\frac{1}{T^2}\right)$, where $T$ means optimization steps. Compared to the convergence rate of $O\left(\frac{1}{T}\right)$ for CM, it is exponentially faster. The updating rule for NAG is shown in Formula 54. In fact, there is not much difference between CM and NAG.

$$\tilde{\Delta}_{\theta t} = \mu \tilde{\Delta}_{\theta t-1} - \alpha \nabla_{\theta} f(\theta + \mu \tilde{\Delta}_{\theta t-1}),$$

(54)

### 3.6.4 Adam

Adam is a first-order gradient-based algorithm invented by Diederik Kingma and Jimmy Ba in 2015 [KB14b]. It combines ideas of RMSProp [TH12] and AdaGrad [DHS11]. Algorithm 5 shows each step of this algorithm.

Adam’s parameters are invariant to rescaling of the gradient. By using statistical information, its step sizes are approximately bounded by hyper-parameters, which
Algorithm 5: $Adam$, $g^2_t$ indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$ and $\lambda = 1 - 10^{-8}$.

Require: $\alpha$: Stepsize

Require: $\beta_1, \beta_2 \in [0, 1)$, $\lambda \in [0, 1)$: Exponential decay rates for the moment estimates

Require: $f(\theta)$: Stochastic objective function with parameters $\theta$

Require: $\theta_0$: Initial parameter vector

$m_0 \leftarrow 0$ (Initialize initial 1st moment vector)

$v_0 \leftarrow 0$ (Initialize initial 2nd moment vector)

$t \leftarrow 0$ (Initialize timestep)

while $\theta_t$ not converged do

$t \leftarrow t + 1$

$\beta_{1,t} \leftarrow \beta_1 \lambda^{t-1}$ (Decay the first moment running average coefficient)

$g_t \leftarrow \nabla\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)

$m_t \leftarrow \beta_{1,t} \cdot m_{t-1} + (1 - \beta_{1,t}) \cdot g_t$ (Update biased first moment estimate)

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g^2_t$ (Update biased second raw moment estimate)

$\hat{m}_t \leftarrow m_t / (1 - \beta_{1,t}^t)$ (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow v_t / (1 - \beta_{2,t}^t)$ (Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)

end while

return $\theta_t$ (Resulting parameters)

enables it to perform better by using step size annealing. For more reference, you can look at the paper by Kingma, Diederik and Ba[KB14b].

4 Prior Art of Combining Deep Neuron Network and RL

Different research groups have tried different methods for combining deep neuron networks with reinforcement learning. The first ideas turned out to be using neural network as a general controller. In the early work applying neuron network to control algorithm, several algorithms using neuron network were proposed including Explanation-Based Neural Network Learning [Thr96] and Neural networks for self-learning control systems [NW90]. These early initiatives provide information about
how one could formulate a framework for neural network to learn a behaviour of
the controller. These structures normally have less input and are relatively less
complicated structure but they offered a way of how to apply neuron network based
algorithm in control theory.

4.1 Control-focused Learning Algorithm

Deep Q Network is a network proposed by Google’s deep learning group. It cre-
ates a direction of combining Convolutional Neural Network with Q learning algo-
rum [MKS$^{+13}$, MKS$^{+15}$]. Despite the fact that it needs a lot of time to train, this
algorithm is actually the first algorithm that can learn vision and strategy together.
In this work, Volodymyr Mnih, etc. provide a frame of using CNN and simple Q
learning algorithm to learn actions of an Atari game (e.g. breakout) so that the
reward for this game is maximized. These algorithms try to build a connection
between reinforcement learning and neural network. On one hand, Control-focused
learning algorithm wants to use neural network to replace the some component in
traditional reinforcement learning. On the other hand, the perception-focused learn-
ing algorithm tries to incorporate learning algorithm for perception.(They only used
simplest form of reinforcement learning in system). Though they made progress,
these two algorithms did not provide a uniform solution to neural based algorithm.
This might be a further research direction.

5 Experiments

In this chapter, we are going to discuss several experiments. First is a cart-pole
balancing experiment and second is Baxter robot learning experiment. Both exper-
iments use PGPE algorithm for learning.

5.1 Cart-pole Balancing

Cart-pole balancing problem is a basic test problem in the control theory. It consid-
ers a physical agent that tries to balance a pole attached to it. Figure 11 illustrates
the problem. The system consists of several states, including the position $s$ and
velocity $\dot{s}$ of the cart, angle $\theta$ and angular velocity $\dot{\theta}$ of the pole. The task involved
in this process is to apply a suitable force to enable the system to balance the pole
so that the angle $\theta$ and the position of the cart $s$ are kept within a range.

Figure 11: The cart-pole balancing system. A force $\vec{F}$ is exerted on the cart $C$ with mass $M$ to balance a pole $l$ that also has length $l$ attached to it. A mass $m$ is attached to the pole $l$. The system has four states including point of the system $s$, velocity of the system $\dot{s}$, angle between pole $l$ and normal $\theta$ and angular acceleration $\dot{\theta}$. When force $\vec{F}$ is exerted on cart $C$, the pole $l$ will also have a force on the point of attachment with cart and the task is to provide forces that can keep the position of the cart $s$ as well as the angle $\theta$ between the pole and the normal within a range of allowed values (e.g. $s \in [-4, 4]$, $\theta \in [-0.2, 0.2]$).

Based on the information we have about the system, we can write the Lagrangian of system:

$$L = \frac{1}{2} M \dot{s}^2 + \frac{1}{2} m v_m^2 - mgl \cos(\theta), \quad (55)$$

where, $v_m$ is the speed of the mass attached to pole $L$. We infer from the system

$$v_m^2 = \left( \frac{d}{dt}(x - l \sin(\theta)) \right)^2 + \left( \frac{d}{dt}(l \cos(\theta)) \right)^2 \quad (56)$$

Then it can be further simplified as

$$v_m^2 = \dot{s}^2 - 2l \dot{x} \dot{\theta} \cos(\theta) + l^2 \dot{\theta}^2 \quad (57)$$

Following the definition of Lagrange, the equations of motion are defined as:

$$\frac{d}{ds} \frac{\partial L}{\partial \dot{s}} - \frac{\partial L}{\partial s} = F \quad (58)$$

$$\frac{d}{dt} \frac{\partial L}{\partial \dot{\theta}} - \frac{\partial L}{\partial \theta} = 0 \quad (59)$$
If we substitute the Lagrangian into the system, we find that the two equations of motion become:

\[(M + m) \ddot{s} - ml \dot{\theta} \cos(\theta) + \frac{1}{2} ml^2 \dot{\theta}^2 - mgl \cos(\theta) = F \] (60)

\[l \ddot{\theta} - g \sin(\theta) = \ddot{s} \cos(\theta) \] (61)

Traditional control algorithms, like proportional-integral-derivative controller (PID) control can solve the problem with a high frequency loop. Although it is an entrance level problem in control theory and also in reinforcement learning, it is also a basic example to test effectiveness of the algorithm.

We used this model for testing the basic performance of the policy gradient algorithm. In this case, the policy algorithm also needs to take into account the states and evaluate the action. During the process, it needs to update its policy parameters from the experience gained from its failures.

Normally, the process’ reward is calculated by the simulator during the training of the actor network. However, in the meantime, we also learn a model that can predict the reward and action according to the previous information we gathered.

5.1.1 System Implementation

The system is implemented using several libraries of Python, including Pybrain, Theano and nntools to provide easy use of the testing process.

Pybrain is a library developed by IDSIA [SBW+10] in order to offer flexible, easy-to-use yet still powerful algorithms for machine learning tasks. It is excellent for reinforcement learning researchers as it separates reinforcement learning into three parts, namely, environment, agent and task. In this way, the library actually builds a framework for further development.

Theano is a GPU-based computational library for implementing deep learning algorithms. It has complete support for different deep learning structures including Convolutional Neuron Networks (CNN), Deep Blief Network (DBN) or Restricted Boltzmann Machine (RBM). It is built itself on scipy, which is a scientific computing library of python and numpy, which is a numerical computation library of python. This enables the library to provide the best computation performance for the system. It also uses Cuda-toolkit to support GPU-based computation, which ensures the nvidia-based system can have the best performance.
ntools is library built on Theano. It provides modularized APIs for different deep learning architectures.

We use these libraries to build a reinforcement learning platform. Following the MDP process, we need to figure out four important elements of the MDP process, namely, \((S, A, T \cdot (\cdot, \cdot), R \cdot (\cdot, \cdot))\). \(S\) is the set of all states of the system. Each state \(s \in S\) contains four physical parameters, including \(s\) position of the cart, \(\dot{s}\) velocity of the cart, \(\theta\) angle of pole, \(\dot{\theta}\) angular velocity of the pole. The action \(a \in A\) is decided by the policy parameters and current states. The transition probability \(T\) is decided by the physical system and the reward function \(R\) is defined as follows:

\[
R(s, \theta, t) = \begin{cases} 
0, & \text{if } |s| \leq 0.05, |\theta| \leq 0.05, t \leq 200 \\
-1, & 2.4 \geq |s| \geq 0.05, 0.7 \geq |\theta| \geq 0.05, t \leq 200 \\
-2 \cdot (200 - t), & \text{otherwise}
\end{cases}
\] (62)

where \(t\) is the time steps in which the system has been running.

### 5.1.2 Experimental Results

The results of the algorithm for solving a cart-pole balancing problem are great. Within 300 trials, it can find the best policy parameters for the system. However, it needs a little bit longer to get the best average rewards. Figure 12 shows the performance of the PGPE algorithm on cart-pole balancing problem. As shown in Figure 12, the x-axis represents number of real world samples needed for training and y-axies represents bar chart of reward of 50 trails. We can see that the average reward increases when more and more real world sampels are used.
Figure 12: The result of the benchmark of the cart-pole balancing problem. The x-axis represents the real world samples needed for training and the y-axis represents the average reward of 50 trials.

5.2 Baxter Robot Learning an Action

Baxter robot is a research robot developed by rethink company (as shown in Figure 5.2). It has two arms, two grippers and an animated face. In this experiment, we will use its arms and grippers to learn to perform a certain wood stacking task. In the following sections, we will discuss how this experiment was implemented.
5.2.1 System Implementation

Each arm of Baxter contains seven degrees of freedom including four roll degrees of freedom $s_0$, $e_0$, $w_0$ and $w_2$, three pitch degrees of freedom $s_1$, $e_1$ and $w_1$. The detail of the information is presented in Figure 5.2. In total, Baxter’s arms have fourteen degrees of freedom, which we consider as states of the robot. They together form one state of the robot i.e. $(\{s_0', s_1', e_0', e_1', w_0', w_1', w_2'\} \cup \{s_0'', s_1'', e_0'', e_1'', w_0'', w_1'', w_2''\}) \in S$. Then, we can apply to each degree of freedom an angular velocity as action, and as a result we have a set of actions $a \in A$. The policy function $\pi(\cdot)$ is a $n$-layer multilayer linear neural network that mathematically is formulated as:

$$\pi(\vec{s}) = (f_1 \circ f_2 \cdots f_n)(\vec{s}),$$

where $f_i(\vec{s})$, $\forall i \in \{1 \cdots n\}$ is defined as $W_i \cdot \vec{s}$ as a single layer linear neural network. The transition function follows the physical law with some noise. The reward function is defined similar to the cart-pole problem. Now, we follow the formulation of PGPE algorithm, using two hidden parameters $\mu, \sigma^2$ for each parameters in the policy function. We also update the algorithm according to the PGPE Algorithm 1 mentioned in Chapter 2. Similarly, we also define the reward function as a step...
Figure 14: The Baxter robot and its corresponding joints. Seven degrees of freedom, including shoulder roll $s_0$, elbow roll $e_0$, wrist roll $w_0$, wrist roll $w_2$, shoulder pitch $s_1$, elbow pitch $e_1$ and wrist pitch $w_1$ are presented in the figure.

function. Here we need to define time steps to be longer, which can be seen in Formula 64.

\[
R(s, \theta, t) = \begin{cases} 
0, & \text{if } |s| \leq 0.05, |\theta| \leq 0.05, t \leq 2000 \\
-1, & 2.4 \geq |s| \geq 0.05, 0.7 \geq |\theta| \geq 0.05, t \leq 2000 \\
-2 \cdot (2000 - t) & \text{otherwise}, 
\end{cases}
\]  

(64)

5.2.2 Experimental Results Using the Baxter Robot

In the real-life experiment, the limit of actions have been set to 0.2 m/s for the actuators of Baxter’s arm. At each iteration we optimize the system for 500 times
and we repeat this process 20 times. We get a similar Figure 5.2 as follows:

Figure 15: Relationship between the reward received and the real world samples needed using PGPE algorithm.

In Figure 15, x-axis represents bar chart of rewards of 50 trials and y-axis shows total number of real world samples. We can observe that although average reward of learning process increased but the variance of the system is very high. This result shows that we need to find a new architecture for learning method of Baxter robot. Current state-of-the-art like LSTM are slow and requires large amount of data. As a consequence, we began a research on optimizing architectures of RNN networks.
6 Research on Accelerating RNN Networks

In this thesis, we also discussed possible neural network structures for reinforcement learning. The multilayer perceptron neural network provided is suitable to be used for reinforcement learning, however more complicated neural networks like RNN are too slow for training. Moreover learning long-term dependences with neural networks is a difficult task not only because it is hard to train in real-time but also because traditional RNN models like LSTM and GRU are not fast enough in training. Currently we are investigating two general RNN network structures, namely Simple Gated Unit (SGU) [GG16] and Deep SGU. Compared to traditional Gated Recurrent Unit (GRU), both structures require fewer parameters and less computation time in sequence classification tasks. Unlike GRU, which requires two gates to control information flow in the network, SGU and DSGU only use one gate to control the flow of information. We show that this difference can accelerate learning speed for tasks that require long dependency information. In addition to SGU and DSGU, we also propose a standard way of representing inner structure of RNN called RNN Conventional Graph (RCG), which is a convenient tool for analysing relationship between input units and hidden units of RNN.

The use of advanced architectures of RNNs, such as Long Short-Term Memory (LSTM) [HS97] and Gated Recurrent Unit (GRU) [CVMG+14a] for learning long dependencies has led to significant improvements in various tasks, such as Machine Translation [BCB15] or Robot Reinforcement Learning [Bak01]. The main idea behind these networks is to use several gates to control the information flow from previous steps to current steps. By employing the gates, any recurrent unit can learn a mapping from one point to another. Hochreiter proposed using two gates, namely an input gate and an output gate in original LSTM [HS97], while Gers added a forget gate to the original LSTM to form current well-known LSTM network [GSC00]. Similarly, in GRU a reset gate and an update gate are used for the same purpose. In this paper, we simplify GRU to contain only one gate, which we name Simple Gated Unit (SGU). Then, we added one layer in order to make SGU deeper to form DSGU. Both models use multiplicative operation to control the flow of information from the previous step to the current step. By doing so, we can approximately reduce one third (for SGU) and one sixth (for DSGU) of parameters needed and accelerate the learning process compared to GRU. The results also reveal that adding layers in RNN’s multiplication gate is an interesting direction of optimizing RNN structure.

In the following sections, we first describe the standard graph for representing de-
tailed inner structure of RNN, i.e. relationships between the input units and recurrent units of each time step. We call this kind of graph RNN Conventional Graph (RCG). Then, we introduce the description of the proposed network SGU and its variant DSGU. Finally, we present preliminary experimental results showing the performance of the proposed networks on IMDB semantic analysis.

6.1 RNN Conventional Graph

RNN is a neural network with recurrent connections. The first step of training a RNN network is to unfold recurrent connections in time, which results in a deep hierarchy consisting of layers of the inner structure of RNN. In most cases, researchers produce their own graphs or use only mathematical equations to represent the structure of an RNN network. However, these representations can be rather complicated and idiosyncratic to each researcher. We designed RCG in order to provide a clear and standard view of the inner structure of RNN.

RCG consists of an input unit, an output unit, default activation functions and gates. It is easy to see how many and what kind of gates an RNN has and the graph can be easily translated into formulas. Normally, RCG takes $\vec{x}_t$ (sometimes also $\vec{x}_1$ to $\vec{x}_{t-1}$) and $\vec{h}_{t-1}$ (sometimes also $\vec{h}_1$ to $\vec{h}_{t-2}$) as inputs and produces $\vec{h}_t$ as an output. An RCG represents the input on the left side and the output on the right side, which enables the graph to show clearly how the information from the left side flows through different structures to the right side.

In the following sections, we use RCG to describe different structures of RNN.

6.1.1 RCG example: Vanilla Recurrent Neural Network

Vanilla Recurrent Neural Network (VRNN) is the simplest form of RNN. It consists of one input node and several hidden nodes. Hidden nodes are recurrent units, which means the current value $\vec{h}_t$ is updated according to the previous unit $\vec{h}_{t-1}$ and the current input $i_t$. Figure 16 shows the relationship between $x_t$, $\vec{h}_{t-1}$ and $\vec{h}_t$ in VRNN. In the form of RCG, VRNN can be represented as:

Mathematically, the updating rule of VRNN is defined as:

$$ \vec{h}_t = \sigma \left( W_{xh}\vec{x} + W_{hh}\vec{h}_{t-1} + \vec{b} \right),$$

(65)

In Figure 16, an arrow $\rightarrow$ represents multiplication with a matrix, an addition node
Figure 16: RCG of Vanilla Recurrent Neural Network. Recurrent input $\vec{h}_{t-1}$ is drawn from the upper side of the graph. Similarly, the input of the current step $x_t$ is drawn from the left side. With an arrow $\rightarrow$ indicating a matrix multiplication operation, the information from two different sources goes into the addition node $\oplus$ in the middle of graph. Followed by a non-linear function $\tanh$, it outputs the current value of hidden units $\vec{h}_t$.

$\oplus$ indicates an addition operation to all the inputs. For example, $\vec{x}_t \rightarrow$ represents $W_{xh}\vec{x}_t$ and $\vec{h}_{t-1} \rightarrow$ represents $W_{hh}\vec{h}_{t-1}$. As a consequence, the whole graph can be directly transformed into Formula 65. The bias vector $\vec{b}$ is ignored in the graph as it can be integrated into the multiplication matrix).

6.2 LSTM and GRU

LSTM and GRU were developed to tackle a major problem suffered by traditional VRNN, namely the exploding and vanishing gradient problem for long-term dependency tasks [PMB12]. They both use gated units to control the information flow through the network. However, LSTM differs from GRU in that LSTM uses three gates to control the information flow of the internal cell unit, while GRU only uses gates to control the information flow from the previous time steps.

6.2.1 LSTM

LSTM contains three gates: an input gate, an output gate and a forget gate – illustrated in Figure 17. At each iteration, the three gates try to remember when and how much the information in the memory cell should be updated. Mathematically, the process is defined by Formulas 66 to 70. Similarly to VRNN, LSTM can also be represented by RCG.

Figure 17 shows the RCG representation of LSTM. In the figure, $\vec{x}_t \rightarrow$ is defined as $W_{xh}\vec{x}_t$, $\vec{c}_{t-1} \rightarrow$ is defined as $W_{ci}\vec{c}_t$, $\vec{h}_{t-1} \rightarrow$ is defined as $W_{hi}\vec{h}_{t-1}$, $\text{var}$ means that the output from the left side is named var and passed to the right side, $\oplus$ means summation over all the inputs, $\otimes$ means multiplication over all the inputs. For symbols
and $\otimes$, the input connections are normally defined as left and up connections, but if there are four connections to the node, then only the right connection is an output connection and the rest of the connections are input connections.

![Diagram of LSTM](image)

Figure 17: LSTM represented in the RCG form. The input $x_t$ is fed in from the left side and recurrent connections are fed from either down or up. The outputs $c_t$ and $h_t$ of this time step are output on the left side.

Mathematically, the relationship between the input and the output of LSTM is defined by a set of the following equations.

$$\tilde{i}_t = \sigma \left( W_{xi} \tilde{x}_t + W_{hi} \tilde{h}_{t-1} + W_{ci} \tilde{c}_{t-1} + \tilde{b}_i \right)$$  (66)

$$\tilde{f}_t = \sigma \left( W_{xf} \tilde{x}_t + W_{hf} \tilde{h}_{t-1} + W_{cf} \tilde{c}_{t-1} + \tilde{b}_f \right)$$  (67)

$$\tilde{c}_t = \tilde{f}_t \tilde{c}_{t-1} + \tilde{i}_t \tanh \left( W_{xc} \tilde{x}_t + W_{hc} \tilde{h}_{t-1} + \tilde{b}_c \right)$$  (68)

$$\tilde{o}_t = \sigma \left( W_{xo} \tilde{x}_t + W_{ho} \tilde{h}_{t-1} + W_{co} \tilde{c}_t + \tilde{b}_o \right)$$  (69)

$$\tilde{h}_t = \tilde{o}_t \tanh(\tilde{c}_t)$$  (70)

Formula 66 describes the update rule of the input gate. It takes the output $\tilde{h}_{t-1}$ of the last time step of the system, the input for the current time step $\tilde{x}_t$, the memory cell value of the last time step $\tilde{c}_{t-1}$ and a bias term $\tilde{b}_i$ into its updating rule. Similarly, Formulas 67 to 70 use these parameters to update their values. Generally, the input gate controls the information flow into the memory cell, the forget gate controls the information flow out of the system and the output gate controls how the information can be translated into the output. These three gates form a path of remembering the long term dependency of the system. A direct relationship between
the RCG representation of LSTM and Formulas 66 to 70 can be easily observed. The first line of Figure 17 shows Formula 66. The second line of Figure 17 shows how Formula 68 calculates $\bar{c}_t$. Similarly, the third and fourth lines of the figure map to Formula 67 and Formula 69, respectively. In the RCG representation of LSTM, three multiplication gates are contained in the graph. It is an important characteristic of LSTM.

6.2.2 GRU

GRU was first designed by Kyunghyun Cho in his paper about Neural Machine Translation [BCB15]. This structure of RNN only contains two gates. The update gate controls the information that flows into memory, while the reset gate controls the information that flows out of memory. Similarly to the LSTM unit, GRU has gating units that modulate the flow of information inside the unit, however, without having a separate memory cell. Figure 18 shows the RCG representation of GRU.

![Figure 18: RCG representation of GRU. The input $\vec{x}_t$ is fed in from the left side. Recurrent connections are fed from either down or up, and output $\vec{h}_t$ is passed to the left. The special gate $\sum \otimes$ in the right corner is defined in Equation 71.](image)

The elements in Figure 18 are similar to the elements in Figure 17. However, the activation $\vec{h}_t$ of GRU at time $t$ is a linear interpolation between the previous activation $\vec{h}_{t-1}$ and the candidate activation $\tilde{h}_t$, which is represented by $\sum \otimes$ in the RCG representation of GRU and defined mathematically as:
\[ \tilde{h}_t = (1 - \tilde{z}_t) \cdot \tilde{h}_{t-1} + \tilde{z}_t \cdot \tilde{h}_t \]  

(71)

where an update gate \( z_t \) decides how much the unit updates its activation, or information from the previous step.

\[ \tilde{z}_t = \sigma_1(W_{hz} \tilde{h}_{t-1} + W_{xz} \tilde{x}_t) \]  

(72)

\[ \tilde{r}_t = \sigma_1(W_{hr} \tilde{h}_{t-1} + W_{xr} \tilde{x}_t) \]  

(73)

\[ \tilde{h}_t = \sigma_2(W_{chx} \tilde{x}_t + W_{chr}(\tilde{r}_t \cdot \tilde{h}_{t-1})) \]  

(74)

\[ \tilde{h}_t = (\tilde{I} - \tilde{z}_t)h_{t-1} + \tilde{z}_t \tilde{h}_t, \]  

(75)

The RCG representation of GRU can be directly translated into Formulas 72 to 75. Formula 72 represents the update gate, Formula 73 represents the reset gate and Formula 75 shows how the output \( h_t \) is calculated.

### 6.3 SGU and DSGU

In this section we describe the proposed Simple Gated Unit (SGU) and Deep Simple Gated Unit (DSGU).

#### 6.3.1 SGU

SGU is a recurrent structure designed for learning long term dependencies. Its aim is to reduce the amount of parameters needed to train and to accelerate the training speed in temporal classification tasks. As we observed earlier, GRU uses two gates to control the information flow from the previous time step to the current time step. However, compared to GRU, SGU uses only one gate to control the information flow in the system, which is simpler and faster in terms of computation time.

Figure 19 shows the structure of SGU. The input is fed into two different function units of the structure. The first line of the graph represents the gate to the recurrent neural network and the second line is the normal recurrent operation. Mathematically, Figure 19 represents the following formulas:
\[
\tilde{x}_t = W_{gh} \tilde{x}_t, \\
\tilde{z}_t = \sigma_1(W_{zh} \tilde{x}_t + \tilde{h}_t), \\
\tilde{h}_t = \sigma_2(\tilde{z}_t \tilde{h}_t), \\
\tilde{h}_t = (1 - \tilde{z}_t) \tilde{h}_t + \tilde{z}_t \tilde{z}_{out}.
\]

Figure 19: SGU in the form of RCG. This structure receives information from the current step and amplifies it by multiplying it with the current hidden states.

Compared to GRU, SGU needs fewer parameters. From Figure 19, we can observe that six weight matrices are needed for GRU, but SGU only needs four weight matrices. Inspired by IRNN, which is a Recurrent Neural Network (RNN) with rectifier as inner activation function [LJH15], the structure uses softplus activation function for input, which intuitively enables the network to learn faster.

### 6.3.2 DSGU

DSGU is also an RNN structure designed for classification tasks. DSGU is designed to tackle a problem associated with SGU – we observed that if SGU is continuously trained, the process might drop dramatically. This is probably due to the shallowness of the gate and the nature of the softmax activation function.

Adding an extra weight matrix to \(z_{out}\) would make controlling the gate with a more complicated structure easier and the network more stable. The structure of DSGU is shown in Figure 20. The only difference compared to SGU is that before \(z_{out}\) one weight matrix is added to the previous output.
Figure 20: The structure of DSGU. Similarly to SGU, the input is fed into two different function units of the structure, namely $\vec{z}$ and $\vec{z}_{out}$.

The first line of the graph represents the gate to the recurrent neural network and the second line is the normal recurrent operation. Mathematically, Figure 20 represents the following formulas:

\[
\begin{align*}
\vec{x}_g &= W_{xh}\vec{x}_t \\
\vec{z}_g &= \sigma_1(W_{zxh}(\vec{x}_g \cdot \vec{h}_{t-1})) \\
\vec{z}_{out} &= \sigma_2(W_{go}(\vec{z}_g \cdot \vec{h}_{t-1})) \\
\vec{z} &= \sigma_3(W_{xz}\vec{x}_t + W_{hz}\vec{h}_{t-1}) \\
\vec{h}_t &= (1 - \vec{z}_t)\vec{h}_{t-1} + \vec{z}_t \cdot \vec{z}_{out},
\end{align*}
\]

6.4 Experimental Results

In this section we report on the performance of SGU and DSGU in three classification tasks.

6.4.1 IMDB Sentiment Classification Task

We use a collection of 50,000 reviews from IMDB, extracted and provided by Stanford Universit [?]. Each movie has no more than 30 reviews and the whole dataset contains an even number of positive and negative reviews. As a consequence, a totally random algorithm yields 50% accuracy.

We tested the prediction accuracy on this dataset with GRU, LSTM, SGU and
DSGU. All the tests were performed using Titan X GPU architecture. We used standard initializations for the LSTM network, including the forget gate. The network uses glorot uniform as the initialization method of the input matrices and orthogonal initialization for the recurrent matrices. See Table 1 for initialization methods and activation functions for all the parameters in LSTM.

<table>
<thead>
<tr>
<th></th>
<th>update gate</th>
<th>reset gate</th>
<th>output gate</th>
<th>cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>glorot uniform</td>
<td>glorot uniform</td>
<td>glorot uniform</td>
<td>None</td>
</tr>
<tr>
<td>U</td>
<td>orthogonal</td>
<td>orthogonal</td>
<td>orthogonal</td>
<td>None</td>
</tr>
<tr>
<td>activation</td>
<td>hard sigmoid</td>
<td>hard sigmoid</td>
<td>hard sigmoid</td>
<td>tanh</td>
</tr>
</tbody>
</table>

Table 1: The initialization method and the activation function of each gate of LSTM in the IMDB sentiment analysis task.

Our implementation of GRU uses the standard structure mentioned above. $W$ is the matrix used for multiplication of $x$, and $U$ is the matrix used for multiplication of the hidden units. Table 2 shows all the configurations of the initialization of different parameters.

<table>
<thead>
<tr>
<th></th>
<th>input gate</th>
<th>forget gate</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>glorot uniform</td>
<td>glorot uniform</td>
</tr>
<tr>
<td>U</td>
<td>orthogonal</td>
<td>orthogonal</td>
</tr>
<tr>
<td>activation</td>
<td>sigmoid</td>
<td>hard sigmoid</td>
</tr>
</tbody>
</table>

Table 2: The initialization method and activation function of each gate of GRU in the IMDB sentiment analysis task.

We ran GRU, SGU and LSTM 50 times on the IMDB sentiment analysis dataset and calculated the mean and variance over all the experiments. The test results comparing SGU, DSGU, GRU and LSTM are shown in Figure 21 and Figure 22. Figure 21 compares the models in terms of the number of iterations, while Figure 22 compares the models in terms of time. Compared with LSTM and GRU, SGU can converge in a very short period (in approximately 2000 iterations or 180 seconds). In this task, we can also observe high variance in the testing phase when learning with LSTM, which makes LSTM less stable in practice. Both figures use the mean values for comparisons of SGU, DSGU, GRU and LSTM. In both cases, SGU and DSGU learn faster than GRU and LSTM.
Figure 21: Comparison of SGU, DSGU, GRU and LSTM in the IMDB sentiment classification task. The y-axis shows validation accuracy of the model, whilst the x-axis represents the number of iterations.

Figure 22: Comparison of SGU, DSGU, GRU and LSTM in the IMDB sentiment classification task in terms of seconds required to perform the task.

### 6.5 MNIST Classification from a Sequence of Pixels

Image classification is a major problem in the field of image processing. MNIST is the simplest and the most well-studied dataset that has been used in many image classification tasks [LBBH98]. In recent studies, pixel-by-pixel MNIST [LJH15] was used to train RNN networks in order to test their ability to classify temporal data. Below, we follow research in Quoc Lee’s paper [LJH15] and compare SGU and DSGU against two models used in the paper [LJH15] i.e. LSTM and IRNN.
In our experiments, we used the rmsprop optimization algorithm and the softmax activation function for LSTM and IRNN in order to match the setting used by Quoc Lee [LJH15]. For IRNN, we used a relatively higher leaning rate of 1e-6 in order to speed up the learning process (In original paper [LJH15], the learning rate is $10^{-8}$). In SGU and DSGU, we used Adam [KB14b] as the optimization algorithm and sigmoid function for optimization.

![Graph showing validation accuracy of DSGU, IRNN, SGU, and LSTM in terms of time for MNIST classification task.](image)

Figure 23: Validation accuracy of DSGU, IRNN and SGU and LSTM in terms of time in the MNIST classification task. Both DSGU and SGU reached a very high accuracy within a short period of time. However, SGU dropped after a short period, which might be due to the fact that it is too simple for learning this task.

The results, presented in Figures 23 and 24, show that DSGU reached the best validation error accuracy (0.978) within a relatively short period of time. (The best result of IRNN in paper [LJH15] is 97% with relatively long training time.) However, SGU failed to increase after around 30 iterations, which indicates hidden units in SGU might not be enough to keep the information in its structure in this particular task. This problem could be fixed by cutting the gradient in the structure, however, in order to provide a more general model and to avoid this problem, we propose using DSGU.
Figure 24: Validation accuracy of DSGU, IRNN and SGU and LSTM in terms of number of iterations in the MNIST classification task. Both DSGU and SGU reached a very high accuracy within a short period of time. However, SGU dropped after short period, which might be due to the fact that it is too simple for learning this task.

6.5.1 Text Generation

Text generation is a specific task designed for testing performance of recurrent neural networks. According to Graves [Gra13b], the recurrent neural network needs to be trained with a large amount of characters from the same distribution, e.g. a particular book. In this experiment, we use a collection of writings by Nietzsche to train our network. In total, this corpus contains 600901 characters and we input one character at a time in order to train the network to find a common pattern in the writing style of Nietzsche.

The structure for learning includes an embedding layer, a corresponding recurrent layer, and an output layer. For this experiment, we vary the recurrent layer and the activation function of the output layer. We tested DSGU, GRU and SGU with the sigmoid activation function in the output layer, while in LSTM, we used both sigmoid and softmax function in the output layer. The optimization algorithm for the models is Adam. We run each configuration 15 times and average the results.
Figure 25 shows the results of the text generation task in terms of the number of iteration. Figure 26 represents the results of the text generation task in terms of time. We can observe that DSGU reached the best accuracy (0.578) the fastest. SGU is also relatively fast. However, the best it can get is less than GRU (0.555 vs 0.556).

![Graph showing validation accuracy of DSGU, GRU, SGU, and LSTM](image)

Figure 25: Validation accuracy of DSGU, GRU, SGU, and LSTM in terms of the number of iterations in the text generation task. Each line is drawn by taking the mean value of 15 runs of each configuration.

### 6.6 Conclusion

In this paper, we proposed two recurrent neural network structures, namely Simple Gated Unit (SGU) and Deep Simple Gated Unit (DSGU). Both structures require fewer parameters than GRU and LSTM.

In experiments, we noticed that both DSGU and SGU are very fast and often more accurate than GRU and LSTM. However, unlike DSGU, SGU seems to sometimes lack the ability to characterize accurately the mapping between two time steps, which indicates that DSGU might be more useful for general applications.

We also found deep multiplication gate to be an intriguing component of RNN. On
one hand, with properly designed multiplication gate, RNN can learn faster than other models but become more fragile due to the fluctuation of data, on the other hand, adding a layer to the multiplication gate can make RNN more stable, while keeping the learning speed.

In the future, we would like to prove the usefulness of deep multiplication gate mathematically, test the performance with a deeper gate as well as perform experiments on more tasks.

Figure 26: Validation accuracy of DSGU, GRU and SGU and LSTM in terms of time in the text generation task. Each line is drawn by taking the mean value of 15 runs of each configuration.
References


Bak02  Bakker, B., Reinforcement Learning with Long Short-Term Memory, 2002.


BCB15  Bahdanau, D., Cho, K. and Bengio, Y., Neural machine translation by jointly learning to align and translate. *ICLR*.


Mil68 Miller, B. L., Finite state continuous time markov decision processes with an infinite planning horizon. *Journal of Mathematical Analysis and Applications*, 22,3(1968), pages 552–569.


