Learning with Artificial Neural Networks

by

Shangtong Zhang

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Statistical Machine Learning

Department of Computing Science

University of Alberta

Abstract

In this thesis, we make two contributions in learning with artificial neural networks. Artificial neural networks have made great success in various challenging domains.

Our first contribution is a new technique named cross-propagation that does cross-validation online. In cross-validation, hold-out training data (i.e., validation set) is used to tune hyper-parameters (e.g., step size) of an algorithm. The key idea of cross-propagation is to use the newly coming training example as a hold-out validation set to update parameters (e.g., weights of a neural network) of an algorithm. We propose three cross-propagation-based algorithms to train neural networks and show the advantage of the three algorithms empirically in both online and off-line setting.

Our second contribution is a systematic evaluation of experience replay, a method that is commonly used in modern deep reinforcement learning systems to stabilize the training of neural networks. Experience replay involves storing transitions into a memory and training the agent with sampled transitions from the memory. In this thesis, we rethink the utility of experience replay. It introduces a new hyper-parameter, the memory size, which is a task-dependent hyper-parameter. We further propose a simple new experience replay method which requires only little extra computation and made the reinforcement learning system more robust to the selection of the memory size compared with the original experience replay.

Preface

A version of Chapter 3 has been accepted for publication and presentation as Vievek Veeriah, Shangtong Zhang, & Richard S. Sutton, Crossprop: Learning Representations by Stochastic Meta-Gradient Descent in Neural Networks. In *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 445-459. Vivek Veeriah and I contributed equally to this work. To be more specific, we developed the cross-propagation algorithm, designed and performed experiments, and wrote the paper together. Richard was the supervisory author and provided insights and feedbacks. The Chapter 3 of this thesis is solely based on my writing.

A version of Chapter 4 has been accepted for presentation as a poster as Shangtong Zhang & Richard S. Sutton, A Deeper Look at Experience Replay. In Deep Reinforcement Learning Symposium at the Conference on Neural Information Processing Systems (NIPS). Long Beach, U.S., 2017. I designed experiments and wrote the paper. Richard was the supervisory author and provided insights and feedbacks. The Chapter 4 of this thesis is solely based on my writing.

To Mom and Dad

Were it to benefit my country I would lay down my life;

What then is risk to me?

– Zemin Jiang, Chinese President, 2009.

Acknowledgements

I would like to thank first my advisor Richard S. Sutton, whose supervision is the most valuable treasure I got in the past two years, introducing me to the fantastic world of reinforcement learning. The most important thing I learned is to have a scientific attitude to problems and be clear about what you are saying.

Further, I would like to thank Vivek Veeriah, who helped me a lot when I was a beginner in research in my first year of graduate study and gave me many useful suggestions when I wrote this thesis. I would also like to thank Kris De Asis, who helped me a lot in my writing, gave insightful feedback on my ideas and taught me to play Rubik's cube. I am also thankful for Yi Wan, Prof Osmar R. Zaiane, Eric Graves, Janey Yu and Tian Tian for their thoughtful discussion and kindly help along the way. I would also like to thank all the members in the RLAI lab for their help during the past two years.

Last but not least, I would like to thank my girlfriend, Yijia Yu, for her company and support during the writing of this thesis.

Contents

1 Learning with Artificial Neural Networks **1**

1.1 Importance of Artificial Neural Networks 1

1.2 Online Cross-validation 2

1.3 An Evaluation of Experience Replay 3

1.4 Outline 4

2 Background and Related Work **5**

2.1 An Example of Back-propagation 5

2.2 Over-fitting in Neural Networks 8

2.3 Elements of Reinforcement Learning 11

2.4 Value Functions 12

2.5 Q-Learning 14

2.6 Experience Replay 16

2.7 Linear Function Approximation with Tile Coding 17

2.8 Nonlinear Function Approximation with Networks 19

2.9 Gradient-based Hyper-parameter Optimization 21

3 The Cross-propagation Algorithms **25**

3.1 Towards Learning Features Stably 25

3.2 Derivation of the Cross-propagation Algorithms 27

3.3 Experimental Results 36

3.3.1 Experiment 1: Online Learning of Related Tasks 39

3.3.2 Experiment 2: Off-line Supervised Learning 44
3.4 Weakness of the Cross-propagation Algorithms 47

4 Evaluation of Experience Replay 49

4.1 Open Questions 49
4.2 Combined Experience Replay 50
4.3 Evaluation Setup 51
4.4 Experiment 1: Tabular Function Representation 58
4.5 Experiment 2: Linear Function Approximation 60
4.6 Experiment 3: Nonlinear Function Approximation 60
4.7 There Is No Universal Rule to Set the Memory Size 64

5 Conclusions and Extensions 69

5.1 Cross-propagation Is A Promising Technique 69
5.2 The Memory Size Is A New Trouble 70
5.3 More Combinations of Cross-propagation and Neural Networks 70
5.4 Cross-propagation in Reinforcement Learning 71

References 73

List of Tables

3.1 Performance gap between cross-propagation and back-propagation in the GEOFF task 47

List of Figures

2.1 The classical reinforcement learning setting 11

3.1 A network fragment to elaborate the issue with back-propagation 26

3.2 The learning curves of cross-propagation and back-propagation for the online GEOFF task 43

3.3 The learning curves of cross-propagation and back-propagation for the online MNIST task 45

4.1 Testbeds for experience replay 57

4.2 Learning curves of agents with tabular function representation in the grid world 61

4.3 Learning curves of agents with linear function representation in the grid world 62

4.4 Learning curves of agents with nonlinear function representation in the grid world 65

4.5 Learning curves of agents with nonlinear function representation in Lunar Lander 66

4.6 Learning curves of agents with nonlinear function representation in Pong 67

Chapter 1

Learning with Artificial Neural Networks

This chapter serves as a brief introduction to the thesis. We first present the importance of neural networks, after which we make two contributions in learning with neural networks. Then we show the outline of the rest of this thesis.

1.1 Importance of Artificial Neural Networks

In machine learning, features are usually used as a proxy of raw inputs to feed a learning algorithm. Manually designing those features for each algorithm was the norm and has enjoyed great success. However, hard-coded features are usually task-dependent and cannot easily scale to large problems. To ease human beings from designing complicated features for each task and each algorithm, neural networks (McCulloch & Pitts ,1943) are used for automatic feature representation learning. Recently, neural networks are shown to be able to handle pretty large and hard problems. And learning feature representations with neural networks in an end-to-end system has enjoyed great success in various challenging domains, for example, defending top human players in the Go game (Silver et al., 2016), outperforming human control in various Atari games (Hessel et al., 2017), classifying or recognizing objects from nat-

ural scene images (He, Gkioxari, Dollar, & Girshick, 2017), and automatically translating text and speeches (Vaswani et al., 2017).

The algorithm to train neural networks behind the great success is stochastic gradient descent. Back-propagation (Rumelhart, Hinton, & Williams, 1985) is an efficient way to implement stochastic gradient descent for neural networks. In this thesis, we use the term back-propagation to refer to a family of algorithms that minimize the current training error via stochastic gradient descent w.r.t. the current weights of a network, where this training error flows backwards through the network layer by layer according to the chain rule.

1.2 Online Cross-validation

Our first contribution is a new technique named cross-propagation that does cross-validation online to update parameters of an algorithm. We propose three cross-propagation-based algorithms to train neural networks and show the merits of the three algorithms empirically in both online and off-line learning.

Cross-validation is a common method in off-line supervised learning to tune hyper-parameters (e.g., step size) of a learning system. In cross-validation, training data is divided into several folds. Some folds are used to train the learning system, and the remaining folds (aka validation set) are used to measure the performance. The error of the learning system on the validation set is a generalization error, which measures how well the learning system can generalize what it has learned from training examples to unseen examples. Hyper-parameters are adjusted according to this generalization error to avoid over-fitting, a phenomenon that a learning system performs well in training examples but fails the unseen examples. In cross-propagation, we do cross-validation online by treating each newly coming training example as a

validation set. And the error on this new training example is interpreted as a generalization error, instead of a training error like back-propagation. We update the current parameters of the learning system via measuring how the parameters at all time steps influence this generalization error.

We propose three cross-propagation-based algorithms to train a single hidden layer network. The three algorithms compute the gradients of the *generalization error* w.r.t. the weights of the network at *all* time steps. In contrast, back-propagation computes the gradients of the *training error* w.r.t. the *current* weights. We show the merits of the three algorithms empirically in both online learning of related tasks and off-line supervised learning.

1.3 An Evaluation of Experience Replay

Our second contribution is a systematic evaluation of experience replay. We rethink the utility of experience replay. It introduces a new task-dependent hyper-parameter, the memory size, which needs careful tuning.

Modern deep reinforcement learning (RL) systems usually use neural networks for function approximation. One common problem introduced by a neural network function approximator is that when the network is trained on recent transitions, it tends to forget what it has learned from earlier transitions. When the neural network experiences previous states again, it fails to produce accurate control or prediction. One possible approach to this issue is to remind the network of earlier transitions frequently. This is what experience replay does. In a learning system with experience replay, the current transition is stored into a memory with a predefined size. And some transitions are sampled from the memory to train the agent. In this thesis, we show that the memory size is a task-dependent hyper-parameter, which needs careful tuning. An improper memory size (e.g., too small or too large) leads to a significant performance drop. However, the defect of a large memory is now hidden by

the complexity of modern deep RL systems, and to our best knowledge, no previous work has pointed this out.

Furthermore, we propose a simple new experience replay method, where we train the agent with not only the sampled transitions but also the transition at the current time step. We name this new method combined experience replay (CER). CER only requires little extra computation while making the learning system more robust to the selection of the memory size compared with the original experience replay.

1.4 Outline

Here we present a brief outline of the rest of this thesis. In Chapter 2, we present some basics of reinforcement learning and neural networks. These basics are necessary to explain our contributions in the following chapters. In Chapter 3, we elaborate our first contribution by showing the derivation of the cross-propagation-based algorithms and some experimental results in both online setting and off-line setting. In Chapter 4, we elaborate our second contribution. We present our evaluation results of the original experience replay and our proposed combined experience replay. Lastly, in Chapter 5, we present some possible directions for future research.

Chapter 2

Background and Related Work

This chapter introduces some basics which are necessary to understand the contributions of this thesis. We start with basics about neural networks, after which we talk about core concepts of reinforcement learning. Finally, we discuss some hyper-parameter optimization techniques, which are related to our contribution.

2.1 An Example of Back-propagation

In this section we present the derivation of back-propagation algorithms for a single hidden layer network in online setting. We start with a scalar regression task. At time step t, the agent receives an observation $\mathbf{x}_t \in \mathbb{R}^n$, the agent tries to predict the target value $y_t^* \in \mathbb{R}$. The prediction y_t is computed by a learnable function $f : \mathbb{R}^n \to \mathbb{R}$,

$$y_t = f(\mathbf{x}_t)$$

We compute the current loss l_t as the squared error,

$$l_t \doteq l(y_t, y_t^*) \doteq \frac{1}{2}(y_t - y_t^*)^2$$

We parameterize f as a single hidden layer network which consists of m hidden units, the incoming weight matrix $\mathbf{U} \in \mathbb{R}^{n \times m}$ and the outgoing weight $\mathbf{w} \in \mathbb{R}^m$. For notational simplicity, we do not treat bias units separately throughout this

thesis. Instead, we assume bias units are included in $\mathbf{U}$ and $\mathbf{w}$. In the rest of this thesis, we use w_j to denote the element indexed by j in $\mathbf{w}$ and u_{ij} to denote the element indexed by (i, j) in $\mathbf{U}$. And we use g to denote the nonlinear function over the hidden units. The prediction at time step t is

$$y_t = \sum_{j=1}^{m} \phi_{j,t} w_{j,t}$$

where $\phi_{j,t}$ is the j-th feature in the hidden layer, in other words,

$$\phi_{j,t} = g(\sum_{i=1}^{n} x_{i,t} u_{ij,t}) \tag{2.1}$$

where $x_{i,t}$ is the i-the element in $\mathbf{x}_t$. In the rest of this thesis, we use $\psi_{j,t}$ to denote $\sum_{i=1}^{n} x_{i,t} u_{ij,t}$.

Back-propagation computes the update of $w_{j,t}$ as

$$\begin{aligned} \frac{\partial l_t}{\partial w_{j,t}} &= \delta_t \frac{\partial y_t}{\partial w_{j,t}} \\ &= \delta_t \frac{\partial \phi_{j,t} w_{j,t}}{\partial w_{j,t}} \\ &= \delta_t \phi_{j,t} \end{aligned} \tag{2.2}$$

where $\delta_t = y_t - y_t^*$. So we have

$$w_{j,t+1} \doteq w_{j,t} - \alpha \delta_t \phi_{j,t} \tag{2.3}$$

The update of $u_{ij,t}$ is

$$\begin{aligned} \frac{\partial l_t}{\partial u_{ij,t}} &= \delta_t \frac{\partial l_t}{\partial \phi_{j,t}} \frac{\partial \phi_{j,t}}{\partial u_{ij,t}} \\ &= \delta_t w_{j,t} \frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} x_{i,t} \end{aligned} \tag{2.4}$$

So we have

$$u_{ij,t+1} \doteq u_{ij,t} - \alpha \delta_t w_{j,t} x_{i,t} \frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \tag{2.5}$$

Note that once the nonlinearity g is determined, $\frac{\partial \phi_{j,t}}{\partial \psi_{j,t}}$ is in a closed form. For instance, if a *sigmoid* function is used, in other words,

$$\phi_{j,t} = \frac{1}{1 + e^{\psi_{j,t}}}$$

then

$$\frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} = (1 - \phi_{j,t})\phi_{j,t}$$

If a *tanh* function is used, in other words,

$$\phi_{j,t} = \frac{e^{2\psi_{j,t}} - 1}{e^{2\psi_{j,t}} + 1}$$

then

$$\frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} = 1 - \phi_{j,t}^2$$

If a *rectified linear* function is used, in other words,

$$\phi_{j,t} = \psi_{j,t}\mathbb{I}_{\psi_{j,t}>0}$$

then

$$\frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} = \mathbb{I}_{\phi_{j,t}>0}$$

where $\mathbb{I}_S$ is an indicator function, it is valued 1 if the statement S is true otherwise 0. To summarize, Equations (2.3) and (2.5) form the back-propagation algorithm for the single hidden layer network in the online scalar regression task. In this thesis, we name this algorithm *Reg-BP*.

Now we consider a classification task of d classes. At time step t, the agent receives an observation $\mathbf{x}_t \in \mathbb{R}^n$. The agent tires to predict the target label $\mathbf{p}_t^* \in \{0,1\}^d$, where $\mathbf{p}_t^*$ is a one-hot vector of length d encoding a class label. We use $x_{i,t}$ and $p_{i,t}^*$ to denote the i-th element of $\mathbf{x}_t$ and $\mathbf{p}_t^*$ respectively. The prediction $\mathbf{p}_t \in [0,1]^d$ is a probability distribution of the d class labels and is computed via a learnable function $f : \mathbb{R}^n \to [0,1]^d$. We use the cross-entropy loss to measure the prediction error l_t at time step t,

$$l_t \doteq l(\mathbf{p}_t, \mathbf{p}_t^*) \doteq -\sum_{k=1}^{d} p_{k,t}^* \log p_{k,t}$$

We parameterize the learnable function f as a neural network, which consists of m hidden units, the incoming weight matrix $\mathbf{U} \in \mathbb{R}^{n \times m}$, and the outgoing

weight matrix $\mathbf{W} \in \mathbb{R}^{m \times d}$. We apply a nonlinear function g over the hidden units and a softmax operation over the output units. The prediction at time step t is

$$\mathbf{p}_t = \text{softmax}(\mathbf{y}_t)$$

in other words,

$$p_{k,t} = \frac{e^{y_{k,t}}}{\sum_{j=1}^{d} e^{y_{j,t}}}$$

where

$$y_{k,t} = \sum_{j=1}^{m} \phi_{j,t} w_{jk,t}$$

where $\phi_{j,t}$ is the j-th feature in the hidden layer as defined before. Note we have

$$\frac{\partial p_{k,t}}{\partial y_{j,t}} = p_{k,t}(\mathbb{I}_{j=k} - p_{j,t}) \tag{2.6}$$

and

$$\delta_{k,t} \doteq \frac{\partial l_t}{\partial y_{k,t}} = p_{k,t} - p^*_{k,t} \tag{2.7}$$

The back-propagation algorithm, referred to as *Cls-BP* in this thesis, for the single hidden layer network in this classification task is

$$w_{jk,t+1} \doteq w_{jk,t} - \alpha \delta_{k,t} \phi_{j,t} \tag{2.8}$$

$$u_{ij,t+1} \doteq u_{ij,t} - \alpha x_{i,t} \frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \sum_{k=1}^{d} \delta_{k,t} w_{jk,t} \tag{2.9}$$

2.2 Over-fitting in Neural Networks

Under certain circumstance, a neural network may reach a reasonable performance level on training data but still perform poorly on unseen examples. This phenomenon is usually referred to as over-fitting.

In typical off-line supervised learning, a neural network is trained on training data with multiple sweeps. We expect the neural network to generalize

the knowledge learned from the training data to unseen test examples. However, sometimes a neural network may only *memorize* the training data rather than *learn* from the training data and perform poorly on unseen test examples. Intuitively, if a neural network is complicated enough, memorizing all the training data may be easier compared with learning knowledge from the training data. Then over-fitting is likely to happen. If a neural network is simple, however, it is impossible to memorize all the training data. So the training algorithm (e.g., stochastic gradient descent) forces the neural network to learn some knowledge from the data to minimize the training loss, and the network is likely to generalize well to unseen data. In practice, a complicated neural network (e.g., a neural network with many layers) usually requires more data to avoid over-fitting than a simple neural network.

Although increasing training examples is an effective approach to address over-fitting, in many tasks, especially supervised learning, it is impractical as obtaining labelled data tends to be expensive. To approach over-fitting without increasing training data, many regularization methods are proposed, where prior knowledge is introduced to constrict the distribution of the weights of a network. The most commonly used regularization methods are ridge regularization (Hoerl & Kennard, 1970) and lasso regularization (Tibshirani, 1996), where an extra penalty is added to the original loss based on the L_2 norm and L_1 norm of the parameters respectively. Although these methods can improve generalization and reduce variance, they often introduce extra bias. So there is a bias-variance trade-off. There are also techniques during the training time to avoid over-fitting. Dropout (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014) is a commonly used technique and has gained great success. When training a neural network with dropout, some units are randomly disabled at each forward pass. A possible hypothesis is that dropout increases the sparsity of the weights and implements model ensemble

implicitly. Besides dropout, some nonlinear activation functions are also be able to increase the sparsity of the weights, for example, Rectified Linear Unit ($ReLU$, Nair & Hinton, 2010).

The cross-validation technique is also used to avoid over-fitting in off-line supervised learning. The training data is divided into several folds. Some random folds are selected to form a training set, and the remaining folds are used as a validation set. Hyper-parameters of an algorithm are tuned according to the performance of the algorithm on the validation set. Although cross-validation has enjoyed great success in off-line learning, it is still not clear how to apply cross-validation in online learning.

Catastrophic forgetting refers to the phenomena that a neural network does perform well on new examples after training on old examples, but fails to preserve the knowledge learned from old examples. So the network performs poorly when it experiences old examples again. Catastrophic forgetting is also another form of over-fitting. A neural network over-fits new examples and forgets what it has learned from old examples.

There are various methods to address catastrophic forgetting. Kirkpatrick et al. (2017) introduced extra loss based on the difference between the current weights and the optimal weights for old tasks to preserve the learned knowledge from old tasks. Li & Hoiem (2017) proposed joint training of old tasks and new tasks. Shin, Lee, Kim, & Kim (2017) exploited Generative Adversarial Nets (GAN, Goodfellow et al., 2014) to generate similar data to old tasks when training a neural network on new tasks. Despite the success of those methods on specific domains, there is still no universal framework to address catastrophic forgetting.

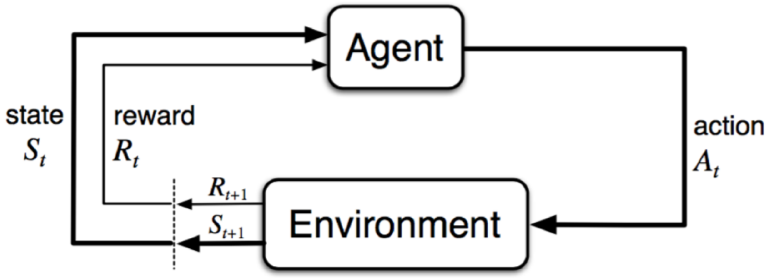


Figure 2.1: The classical reinforcement learning setting: an agent interacts with the environment. This figure is from Sutton & Barto (2018).

2.3 Elements of Reinforcement Learning

In the classical reinforcement learning setting, an agent continually interacts with the environment to achieve a goal, which is defined by the reward signal.

Figure 2.1 shows a paradigm that an agent interacts with the environment. At time step t, the agent is at state S_t. It chooses an action A_t based on its policy π. The environment then gives a reward R_{t+1} and leads the agent to a new state S_{t+1}. The goal of an agent is to maximize the return, which is defined to be the sum of the discounted rewards in the future.

Formally speaking, we present a reinforcement problem by a Markov Decision Process (MDP), which consists of a state space $\mathcal{S}$, an action space $\mathcal{A}$, a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, a state-transition function $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$, and a discount factor $\gamma \in [0, 1]$. The state transition function p defines a probability distribution over next states given the current state and action,

$$p(s' \mid s, a) \doteq \Pr\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\}$$

The reward function r computes the expected immediate reward given the current state and action,

$$r(s, a) \doteq \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a]$$

The state transition function p and the reward function r together form the

dynamics of an MDP. The discount factor γ determines how the agent trades off between short-term rewards and long-term rewards.

At time step t, the agent is at state $S_t \in \mathcal{S}$, it takes an action $A_t \in \mathcal{A}$ according to its policy $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$, which defines a probability distribution over the action space $\mathcal{A}$ at state S_t,

$$\pi(a \mid s) \doteq \Pr\{A_t = a \mid S_t = s\}$$

The environment then leads the agent to a new state S_{t+1} according to $p(\cdot \mid S_t, A_t)$ and gives a reward signal R_{t+1}. The goal for an agent is to maximize the return, which is defined to be the sum of the discounted future rewards. We use G_t to denote the return at time step t,

$$G_t \doteq \sum_{i=t+1}^{T} \gamma^{i-t-1} R_i$$

where T is the time step that the episode ends. We can see here if γ is 0, the agent only cares the immediate reward. However, if γ is 1, the agent treats all the future rewards with same importance.

In a reinforcement learning problem, the state transition function p and the reward function r are usually unknown to the agent. The agent has to learn its policy from interactions with the environment, in other words, the agent has to adjust the policy according to the reward signal.

2.4 Value Functions

In reinforcement learning, we have state value function and action value function. State value function describes the utility of a state. The most powerful method to learn state value function is the Temporal-Difference methods (TD, Sutton, 1988). The fundamental idea of TD methods is bootstrapping. We learn the utility of a state based on our estimation of the utility of its successor.

We use v_π to denote the state value function under policy π. The state value function v_π is defined to be the expected discounted return starting from state s following policy π,

$$v_\pi(s) \doteq \mathbb{E}_\pi\left[\sum_{i=t+1}^{T} \gamma^{i-t-1} R_i \mid S_t = s\right]$$

The value function v_π satisfies the Bellman equation (Bellman, 2013),

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)\big(r(s,a) + \gamma v_\pi(s')\big)$$

We use V to denote our estimation of the state value function v_π. Following the simplest TD method, 1-step TD(0), we update our estimation of the value of S_t as

$$V(S_t) \leftarrow V(S_t) + \alpha\big(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\big)$$

where α is a step size. Here $R_{t+1} + \gamma V(S_{t+1})$ is our estimation of G_t. We learn our estimation of $v_\pi(S_t)$ from another estimation of G_t. This is the key idea of TD learning.

Similar to state value function, we use action value function $q_\pi(s,a)$ to measures the utility of an action a at state s,

$$q_\pi(s,a) \doteq \mathbb{E}_\pi\left[\sum_{i=t+1}^{T} \gamma^{i-t-1} R_i \mid S_t = s, A_t = a\right]$$

The two value functions, v_π and q_π, are connected via the policy π,

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a)$$

We also have the Bellman equation for the action value function q_π,

$$q_\pi(s,a) = \sum_{s'} p(s'|s,a)\big(r(s,a) + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')\big)$$

We use Q to denote our estimation of the action value function q_π. Similar to TD methods, we have 1-step SARSA(0) (Rummery & Niranjan, 1994) to learn the Q function,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big)$$

With tabular function representation, TD methods converge to the optimal state value function v_*, which is defined as

$$v_*(s) \doteq \max_\pi v_\pi(s)$$

SARSA methods converge to the optimal action value function $q_*(s, a)$, which is defined as

$$q_*(s, a) \doteq \max_\pi q_\pi(s, a)$$

We have corresponding Bellman optimality equation (Bellman, 2013),

$$v_*(s) = \max_a \sum_{s'} p(s'|s, a)\big(r(s, a) + \gamma v_*(s')\big)$$

$$q_*(s, a) = \sum_{s'} p(s'|s, a)\big(r(s, a) + \gamma \max_{a'} q_*(s', a')\big)$$

And the optimal state value function is related to the optimal state action value function as

$$v_*(s) = \max_a q_*(s, a)$$

The two different value functions usually serve different purposes. Action value function is usually used for control, as it describes which action is better directly by giving the utility of that action. However, it is often difficult to use state value function for control, as to determine the utility of an action via the state value function, the agent has to know the dynamics (i.e., p and r) of the environment. Having access to the dynamics is often impractical in typical reinforcement learning problems. In general, state value function often serves policy evaluation, a setting where we want to know how good a policy is.

2.5 Q-Learning

The TD methods and SARSA methods are both on-policy methods, where the behavior policy and the target policy are the same. Here we use the term

behavior policy to denote the policy that the agent is following when it chooses actions, while the term *target policy* refers to the policy that the agent wants to learn. In general, we use π to denote the target policy and μ to indicate the behavior policy.

It is common the case that the target policy is different from the behavior policy, where we need off-policy methods. Q-learning (Watkins, 1989) is a commonly used off-policy algorithm. Q-learning updates our estimation Q as

$$Q(S_t) \leftarrow Q(S_t) + \alpha\big(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\big) \tag{2.10}$$

The learned estimation Q always converges to the optimal action value function q_* in tabular case, independent with the behavior policy that is being followed by the agent. In practice, a ϵ-greedy policy is often used as a behavior policy. The agent takes a random action with probability ϵ and takes a greedy action w.r.t. to the current action value with probability $1 - \epsilon$. This algorithm is elaborated in Algorithm 1.

Algorithm 1: Q-learning with with ϵ-greedy exploration.

```
Initialize the action value function Q
while not converged do
    Initialize S
    while S is not terminal do
        Choose A from S using the policy derived from Q (ε-greedy)
        Take action A, observe S' and R
        Q(S, A) ← Q(S, A) + α(R + γ max_a Q(S', a) − Q(S, A))
        S ← S'
    end
end
```

The difference between Q-learning and SARSA is elaborated in the Cliff Grid World example (Sutton & Barto, 1998), where the Q-learning agent learned the optimal policy, while the SARSA agent learned the safe policy. Q-learning is the simplest importance-sampling-free off-policy method. It is an instance of off-policy expected SARSA (Van Seijen, Van Hasselt, White-

son, & Wiering, 2009), which is a special case of the Tree Backup algorithm (Precup, Sutton, & Singh, 2000). There are also various importance sampling based off-policy methods, which are out of the scope of this thesis.

2.6 Experience Replay

Experience replay was first introduced by Lin (1992) and then adapted by Mnih et al. (2015). The key idea of experience replay is to train an agent with previously experienced transitions. Lin (1992) used experience replay in off-line training. After an episode ends, the trajectory of this episode is stored into a memory, and several trajectories are sampled from the memory under certain sampling strategy to train the agent. Mnih et al. (2015) used experience replay in online learning. At each time step, the current transition $S_t, A_t \rightarrow R_{t+1}, S_{t+1}$ is stored into a memory (named *replay buffer*) with a predefined size as an entry. If the memory is full, the oldest entry will be removed. Some transitions are then sampled uniformly from the memory to train the agent. It is important to note that the current transition is not used for training the agent immediately. Algorithm 2 shows the skeleton of the experience replay proposed by Mnih et al. (2015). This experience replay method for online learning is then further used in Deep Deterministic Policy Gradient (DDPG, Lillicrap et al., 2015) and off-policy actor-critic methods (ACER, Wang et al., 2016). In the rest of this thesis, we use the term experience replay to denote the exact experience replay method used by Mnih et al. (2015). We may also refer to this experience replay method as the original experience replay when we need to distinguish it from our proposed new experience replay method.

Instead of a uniform sampling policy, prioritized sampling is another common strategy (Schaul, Quan, Antonoglou, & Silver, 2015). To be more specific, each transition stored in the memory is associated with a priority. The prioritized experience replay (PER) then computes a probability distribution over

all the stored transitions based on the priorities. This probability distribution is used to sample transitions from the memory. There are various methods to determine the priority of a transition. The most successful one is to use the TD error of a transition as the priority. The intuition is that the TD error describes how 'surprising' a transition is. Schaul et al. (2015) also proposed to set the priority of the current transition to the maximal priority value of all the transitions in the memory to increase its opportunity to be replayed. PER significantly speeds up learning compared with the original experience replay. However, PER always introduces some complicated data structures (e.g., a sum-tree), and the required extra computation has a logarithmic growth with the increase of the memory size.

Algorithm 2: The experience replay framework.

```
Initialize the replay buffer
while not converged do
    for each time step in the episode do
        ...
        Store the current transition (S, A, R, S') into the memory
        ...
        Sample a batch of transitions uniformly from the memory
        Train the agent with the sampled transitions
        ...
    end
end
```

2.7 Linear Function Approximation with Tile Coding

The method to represent a value function in a parameterized function form is named function approximation (Sutton, 1996). The simplest function approximation method is linear function approximation, where value function is represented by a linear combination of the features of a state. Formally

speaking, a value function is parameterized by a vector $\mathbf{w} \in \mathbb{R}^d$. And we use

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^{\mathrm{T}}\mathbf{x}(s)$$

and

$$\hat{q}(s, a, \mathbf{w}) \doteq \mathbf{w}^{\mathrm{T}}\mathbf{x}(s, a)$$

to represent our approximation of the true state value function $v_\pi(s)$ and the true state action value function $q_\pi(s, a)$. Here we abuse the symbol $\mathbf{x}$ to denote the feature vector of both a state s and a state action pair (s, a). To be more specific, $\mathbf{x}(s) \doteq \big(x_1(s), \ldots, x_d(s)\big)^{\mathrm{T}}$ where $x_i : \mathcal{S} \to \mathbb{R}$ is a feature function for the state s, and $\mathbf{x}(s, a) \doteq \big(x_1(s, a), \ldots, x_d(s, a)\big)^{\mathrm{T}}$ where $x_i : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a feature function for the state action pair (s, a). $\mathbf{w}^{\mathrm{T}}\mathbf{x}$ represents the inner product of the two vector $\mathbf{w}$ and $\mathbf{x}$, in other words, $\mathbf{w}^{\mathrm{T}}\mathbf{x} \doteq \sum_{i=1}^{d} w_i x_i$.

Similar to TD methods in tabular case, we have semi-gradient TD methods (Sutton & Barto, 2018) for function approximation. Particularly for linear function approximation, we update the weight vector $\mathbf{w}$ as

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\big(R_{t+1} + \gamma\mathbf{w}_t^{\mathrm{T}}\mathbf{x}(S_{t+1}) - \mathbf{w}_t^{\mathrm{T}}\mathbf{x}(S_t)\big)\mathbf{x}(S_t)$$

For linear semi-gradient SARSA, we have

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\big(R_{t+1} + \gamma\mathbf{w}_t^{\mathrm{T}}\mathbf{x}(S_{t+1}, A_{t+1}) - \mathbf{w}_t^{\mathrm{T}}\mathbf{x}(S_t, A_t)\big)\mathbf{x}(S_t, A_t)$$

And for linear semi-gradient Q-learning, we have

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\big(R_{t+1} + \gamma \max_a \mathbf{w}_t^{\mathrm{T}}\mathbf{x}(S_{t+1}, a) - \mathbf{w}_t^{\mathrm{T}}\mathbf{x}(S_t, A_t)\big)\mathbf{x}(S_t, A_t)$$

There are various methods to build a feature function, for example, polynomial basis, Fourier basis (Konidaris, Osentoski, & Thomas, 2011) and coarse coding (Hinton, McClelland, & Rumelhart, 1986). The most powerful one is tile coding, which is a special case of coarse coding and is designed for multi-dimensional continuous state space (e.g., $\mathbb{R}^d$). In tile coding, the whole state

space is partitioned into many groups. We usually use many different partition strategies, resulting in many partitions. Each such partition is called a tiling. Each group in a tiling (partition) is named a tile. In practice, we usually use multiple overlapped tilings. A state (i.e., an n-dimensional point) is represented by active tiles (i.e., tiles that happen to cover that state). A binary vector is used to encode those active tiles.

Tile coding and its variants have made great success in solving challenging reinforcement learning tasks (Stone, Sutton, & Kuhlmann, 2005).

2.8 Nonlinear Function Approximation with Networks

Although linear function approximation has enjoyed great success in reinforcement learning, designing features for linear methods requires non-negligible human effort, especially for raw pixel inputs (Liang, Machado, Talvitie, & Bowling, 2016). Inspired by the success of convolutional neural networks in computer vision (Krizhevsky, Sutskever, & Hinton, 2012), Mnih et al. (2015) proposed the Deep-Q-Network (DQN), an end-to-end learning system that learns the feature and the value function simultaneously. DQN used a deep neural network with multiple convolutional layers to approximate the state-action value function and achieved human-level control on various Atari games with raw pixels as input.

We use $\mathbf{w}$ to represent the vector of the weights of the deep neural network used by DQN. The update that DQN makes is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\big(R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)\big)\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w}_t) \tag{2.11}$$

where $\tilde{q}$ is the target network which will be explained soon. This is similar to the update rule of the semi-gradient Q-learning (Sutton & Barto, 2018).

The function approximator in DQN is a deep convolutional network, which is extremely hard to train. Typically to train a neural network, data with i.i.d. property is necessary. However, in the online data stream of a reinforcement learning system, the data is highly temporally correlated. During the online training, the neural network function approximator tends to over-fit recent transitions, which only covers a small portion of the whole state space. As a result, the network fails to produce accurate control or prediction when it experiences old states again. To reduce the negative influence of the correlation in the online data stream, Mnih et al. (2015) incorporated experience replay into semi-gradient Q-learning. The current transition is stored into the memory directly, and only the uniformly sampled transitions from the memory are used to train the network. In other words, the quadruple (S_t, A_t, R_t, S_{t+1}) in Equation (2.11) is no longer the current transition. Instead, it is the transition sampled from the memory. Mnih et al. (2016) also proposed multiple workers to gain temporally uncorrelated data. The reinforcement learning system starts several agents simultaneously, and each agent interacts with its own environment instance to collect data. Then the network update is based on the data from all the agents. Although this method has gained great success in practice, it is cheating. In the original problem, there is only one environment, however, this method requires multiple independent instances of this environment.

Another significant contribution of DQN is the target network $\tilde{q}$, which allows a stable training target. The target network $\tilde{q}$ is a copy of the learning network $\hat{q}$ and is synced with the learning network periodically.

Due to the success of DQN in Atari games, deep neural networks are now widely used as nonlinear function approximators. And following DQN, many deep reinforcement learning systems use experience replay to stabilize the training of the neural network function approximator (Lillicrap et al., 2015;

Andrychowicz et al., 2017).

2.9 Gradient-based Hyper-parameter Optimization

Back-propagation applies gradient descent to optimize parameters of a learning system (e.g., weights of a neural network). There are also methods optimizing hyper-parameters (e.g., a step size) by gradient descent.

We first present some online gradient-based hyper-parameter optimization methods. Sutton (1992a) proposed the Incremental Delta-Bar-Delta (IDBD) algorithm, which is a stochastic meta-gradient descent algorithm and aimed to solve the problem of step size selection. To be more specific, in IDBD each weight of a network is associated with its own step size, and the system optimizes the step size via stochastic gradient descent based on the received error signal.

We consider the same online learning setting as Section 2.1. At time step t, the agent receives an observation $\mathbf{x}_t \in \mathbb{R}^n$. The agent tries to predict the target value $y_t^* \in \mathbb{R}$. The prediction y_t is computed by a learnable function $f : \mathbb{R}^n \to \mathbb{R}$,

$$y_t = f(\mathbf{x}_t)$$

We compute the current loss l_t as the squared error,

$$l_t \doteq l(y_t, y_t^*) \doteq \frac{1}{2}(y_t - y_t^*)^2$$

We then parameterize the learnable function $f : \mathbb{R}^n \to \mathbb{R}$ via a vector $\mathbf{w} \in \mathbb{R}^n$. We use w_i to denote the i-th element of $\mathbf{w}$, and the prediction for the example $\mathbf{x}_t$ is

$$y_t = \sum_{i=1}^{n} w_{i,t} x_{i,t}$$

In a traditional back-propagation algorithm (c.f. *Reg-BP* in Section 2.1), we update w_i as

$$w_{i,t+1} \doteq w_{i,t} - \alpha \delta_t x_{i,t}$$

where α is a step size and $\delta_t = y_t - y_t^*$ as defined in Section 2.1. However, in IDBD, each weight w_i is associated with its own learning rate α_i, and the update rule for w_i is

$$w_{i,t+1} \doteq w_{i,t} - \alpha_{i,t+1} \delta_t x_{i,t} \tag{2.12}$$

Here we use $t+1$ rather than t to index α_i to indicate that the update for α_i occurs before the update for w_i. In IDBD, each learning rate α_i has the form

$$\alpha_{i,t} = e^{\beta_{i,t}}$$

The incremental update for $\beta_{i,t}$ is $\frac{\partial l_t}{\partial \beta_i}$. This partial derivative measures the sensitivity of the loss at the current time step l_t to a small change of β_i at all time steps. Similar techniques are used by Williams & Zipser (1989) for the analysis of recurrent neural networks. We further expand $\frac{\partial l_t}{\partial \beta_i}$ as

$$\frac{\partial l_t}{\partial \beta_i} = \delta_t \frac{\partial \delta_t}{\partial \beta_i} \tag{2.13}$$

$$= \delta_t \frac{\partial y_t}{\partial \beta_i} \tag{2.14}$$

$$= \delta_t \sum_j \frac{\partial y_t}{\partial w_{j,t}} \frac{\partial w_{j,t}}{\partial \beta_i} \tag{2.15}$$

(The primary influence of β_i on the loss is through w_i)

$$\tag{2.16}$$

$$\approx \delta_t \frac{\partial y_t}{\partial w_{i,t}} \frac{\partial w_{i,t}}{\beta_i} \tag{2.17}$$

$$= \delta_t x_{i,t} h_{i,t} \tag{2.18}$$

where we define

$$h_{i,t} \doteq \frac{\partial w_{i,t}}{\partial \beta_i}$$

Finally, the update rule for $\beta_{i,t}$ is

$$\beta_{i,t+1} \doteq \beta_{i,t} - \theta\delta_t x_{i,t} h_{i,t} \tag{2.19}$$

where θ is a meta learning rate. The update for $h_{i,t}$ can also be done incrementally,

$$\begin{aligned}
h_{i,t+1} &= \frac{\partial w_{i,t+1}}{\partial \beta_i} && (2.20)\\
&= \frac{\partial (w_{i,t} - \alpha_{i,t+1}\delta_t x_{i,t})}{\partial \beta_i} && \text{Plug in Equation (2.12)}\\
& && (2.21)\\
&= h_{i,t} - x_{i,t}\frac{\partial e^{\beta_{i,t+1}}\delta_t}{\partial \beta_i} && (2.22)\\
&= h_{i,t} - x_{i,t}\frac{\partial e^{\beta_{i,t+1}}}{\partial \beta_i}\delta_t - x_{i,t}e^{\beta_{i,t+1}}\frac{\partial \delta_t}{\partial \beta_i} && \text{Product rule of calculus}\\
& && (2.23)\\
&= h_{i,t} - x_{i,t}\frac{\partial e^{\beta_{i,t+1}}}{\partial \beta_{i,t+1}}\delta_t - x_{i,t}e^{\beta_{i,t+1}}\frac{\partial \delta_t}{\partial \beta_i} && (2.24)\\
&= h_{i,t} - x_{i,t}e^{\beta_{i,t+1}}\delta_t - x_{i,t}e^{\beta_{i,t+1}}\frac{\partial \delta_t}{\partial \beta_i} && (2.25)\\
&= h_{i,t} - x_{i,t}\alpha_{i,t+1}\delta_t - x_{i,t}\alpha_{i,t+1}\frac{\partial y_t}{\partial \beta_i} && (2.26)\\
&= h_{i,t} - x_{i,t}\alpha_{i,t+1}\delta_t - x_{i,t}\alpha_{i,t+1}\sum_{j=1}^{n}\frac{\partial y_t}{\partial w_{j,t}}\frac{\partial w_{j,t}}{\partial \beta_i} && (2.27)
\end{aligned}$$

(The primary influence of β_i on the prediction y_t is through $w_{i,t}$)

$$\begin{aligned}
&\approx h_{i,t} - x_{i,t}\alpha_{i,t+1}\delta_t - x_{i,t}\alpha_{i,t+1}\frac{\partial y_t}{\partial w_{i,t}}\frac{\partial w_{i,t}}{\partial \beta_i} && (2.28)\\
&= h_{i,t} - x_{i,t}\alpha_{i,t+1}\delta_t - x_{i,t}\alpha_{i,t+1}x_{i,t}h_{i,t} && (2.29)\\
&= h_{i,t}(1 - \alpha_{i,t+1}x_{i,t}^2) - \alpha_{i,t+1}\delta_t x_{i,t} && (2.30)
\end{aligned}$$

After adding a positive bounding operation, we obtain the update rule for $h_{i,t}$,

$$h_{i,t+1} = h_{i,t}[1 - \alpha_{i,t+1}x_{i,t}^2]^+ - \alpha_{i,t+1}\delta_t x_{i,t} \tag{2.31}$$

where $[x]^+ \doteq x\mathbb{I}_{x>0}$. Equations (2.12), (2.19) and (2.31) together form the IDBD algorithm.

Inspired by IDBD, Sutton (1992b) further proposed K1 and K2 algorithms for online step size adaptation, and Schraudolph (1999) generalized K1 and IDBD via the Stochastic Meta Gradient (SMD) algorithm.

Besides IDBD and its extensions, there is another line of works that do gradient-based hyper-parameter optimization in off-line setting. This line of works shares the similar idea with IDBD but mainly focuses on off-line learning. This line started from Bengio (2000), where the gradient of a model selection criterion (e.g., the error of the trained model on a validation set) is used to update hyper-parameters. And Do, Foo, & Ng (2008) used a similar technique to learn weights of regularizer terms for a model. Later on, Domke (2012) applied the similar idea to the training of energy models and showed improvements in image labelling and denoising. More recently, Maclaurin, Duvenaud, & Adams (2015) applied gradient-based hyper-parameter optimization in optimizing various huge amount of hyper-parameters, for example, neural network weight initialization and training data augmentation procedure. Luketina, Berglund, Greff, & Raiko (2016) proposed a scalable gradient-based approach to optimize both parameters and hyper-parameters simultaneously. Pedregosa (2016) proposed an approximate gradient in gradient-based hyper-parameter optimization to speed up training. And Franceschi, Donini, Frasconi, & Pontil (2017) proposed a real-time approach for gradient-based hyper-parameter optimization. More recently, Franceschi, Frasconi, Salzo, & Pontil (2018) unified gradient-based hyper-parameter optimization and meta learning with bilevel programming.

The proposed cross-propagation technique in this thesis is related to the gradient-based hyper-parameter optimization algorithms in this section, and we will detail the similarity and difference in Section 3.1.

Chapter 3

The Cross-propagation Algorithms

This chapter elaborates the first contribution of this thesis in detail. We first show a possible defect of back-propagation. Then we present the three cross-propagation-based algorithms for training a neural network. And we show the merits of the three algorithms empirically in both online and off-line setting.

3.1 Towards Learning Features Stably

In a neural network, we usually interpret the output of the second last layer as features of the input. We now present a possible defect of back-propagation. Back-propagation makes the smallest change to the weights of a neural network to minimize the training error. This update tends to change the most used feature more than the less used features. Although back-propagation has enjoyed great success in various domains, this update may be a possible defect of back-propagation. We clarify this possible defect by a small example introduced by Sutton (1986). As shown in Figure 3.1, we consider a simple network with 3 hidden units A, B and C. They represent 3 learned features. D is the output unit. According to the weights of the connections AD, BD and CD, C is the most used feature as it has the largest outgoing weight 10. Back-propagation updates the weight of CE 10 times more than the weight of BE. Although this

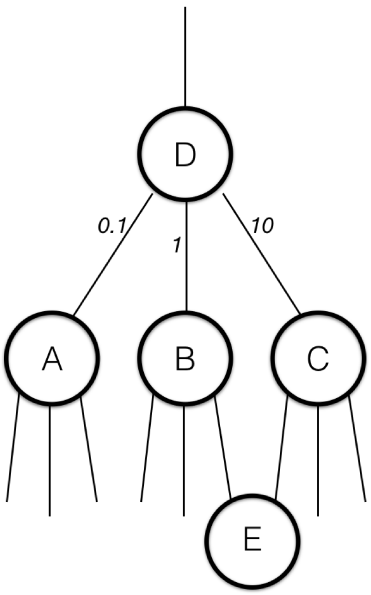


Figure 3.1: A network fragment, numbers indicate weight values.

update is the smallest change to minimize the current training error, updating the most used feature most may be a possible defect. The most used feature may contain much knowledge learned from previous training examples. Updating this feature much may destroy that knowledge, and the network may need to relearn this feature again in the future. As a result, the learning of features may become unstable. This update may be a cause of catastrophic forgetting. In this thesis, we propose three cross-propagation-based algorithms to learn features stably.

We first introduce the cross-propagation technique. At the current time step, we receive a new training example and get the prediction loss. Many algorithms (e.g., back-propagation and its variants) consider this loss to be a *training loss* and update the parameters based on how the current parameters influence this loss. However, in cross-propagation, we treat this new training example as a hold-out validation set and all previous training examples as the training set. We interpret the current loss as a *generalization loss* or *validation loss*. And we update the current parameters based on how all previous

parameters influence the current loss. In this way, we achieve cross-validation online. This technique is inspired by the IDBD algorithm, where the current hyper-parameters (e.g., step size) are updated based on how their history values influence the current loss. Cross-propagation is also related to the off-line gradient-based hyper-parameter optimization methods discussed in Section 2.9. However, there are two main differences between cross-propagation and those works. First, those works focused on off-line learning, and there has not been a successful application of those works in online learning. Second, those works use generalization loss to tune hyper-parameters, while cross-propagation uses generalization loss to tune parameters.

In this thesis, we propose to do online cross-validation to update the feature layer (i.e., the first layer) of a single hidden layer network, resulting in three cross-propagation-based algorithms. To be more specific, we update the feature layer by measuring how its previous weights influence the current loss. In this way, the update for the feature layer at the current time step is likely to compromise with all previous examples, and the frequently used features are more likely to be well preserved than back-propagation.

3.2 Derivation of the Cross-propagation Algorithms

In this section, we first show the derivation of two cross-propagation-based algorithms for scalar regression tasks, after which we present a cross-propagation-based algorithm for classification tasks.

To derive the cross-propagation algorithms for scalar regression tasks, we use the same online scalar regression task and corresponding function parameterization as described in Section 2.1. At time step t, the agent receives an observation $\mathbf{x}_t \in \mathbb{R}^n$. It tries to predict the target value $y^*_t \in \mathbb{R}$. Its prediction

y_t is computed by a learnable function $f : \mathbb{R}^n \to \mathbb{R}$,

$$y_t = f(\mathbf{x}_t)$$

We compute the current loss l_t as the squared error,

$$l_t \doteq l(y_t, y_t^*) \doteq \frac{1}{2}(y_t - y_t^*)^2$$

We parameterize f as a single hidden layer network which consists of m hidden units, the incoming weight matrix $\mathbf{U} \in \mathbb{R}^{n \times m}$ and the outgoing weight $\mathbf{w} \in \mathbb{R}^m$. We use $\phi_{j,t}$ to indicate the j-th feature (i.e., the value of the j-th hidden unit) and $\psi_{j,t}$ to indicate the value of the j-th hidden unit before the non-linear function g at time t as in Section 2.1. We also use δ_t to indicate the difference between the prediction and the target. To be more specific, we have

$$y_t = \sum_{j=1}^{m} \phi_{j,t} w_{j,t} \tag{3.1}$$

$$\phi_{j,t} = g(\psi_{j,t}) \tag{3.2}$$

$$\psi_{j,t} = \sum_{i=1}^{n} x_{i,t} u_{ij,t} \tag{3.3}$$

$$\delta_t = y_t - y_t^* \tag{3.4}$$

The update rule for the outgoing weight $\mathbf{w}$ is the same as back-propagation (i.e., Equation (2.3)). When updating $u_{ij,t}$, however, we use $\frac{\partial l_t}{\partial u_{ij}}$ instead of $\frac{\partial l_t}{\partial u_{ij,t}}$. This partial derivative with respect to u_{ij} measures the sensitivity of the loss at current time step l_t to a small change of u_{ij} at all time steps. Similar techniques are used in Williams & Zipser (1989) and Sutton (1992a). We further expand $\frac{\partial l_t}{\partial u_{ij}}$ as

$$\frac{\partial l_t}{\partial u_{ij}} = \delta_t \frac{\partial y_t}{\partial u_{ij}} \tag{3.5}$$

$$\frac{\partial y_t}{\partial u_{ij}} = \sum_k \frac{\partial y_t}{\partial w_{k,t}} \frac{\partial w_{k,t}}{\partial u_{ij}} \tag{3.6}$$

(The primary effect of the input weight u_{ij} will be through the corresponding output weight $w_{j,t}$)

$$\approx \frac{\partial y_t}{\partial w_{j,t}} \frac{\partial w_{j,t}}{\partial u_{ij}} \tag{3.7}$$

$$= \phi_{j,t} h_{ij,t} \tag{3.8}$$

where we define

$$h_{ij,t} \doteq \frac{\partial w_{j,t}}{\partial u_{ij}}$$

According to Equations (3.5) and (3.8), the update rule for $u_{ij,t}$ is

$$u_{ij,t+1} \doteq u_{ij,t} - \alpha \delta_t \phi_{j,t} h_{ij,t} \tag{3.9}$$

And we update $h_{ij,t}$ incrementally as

$$h_{ij,t+1} = \frac{\partial w_{j,t+1}}{\partial u_{ij}} \tag{3.10}$$

$$= \frac{\partial (w_{j,t} - \alpha \delta_t \phi_{j,t})}{\partial u_{ij}} \qquad \text{Equation (2.3)} \tag{3.11}$$

$$= h_{ij,t} - \alpha \frac{\partial \delta_t \phi_{j,t}}{\partial u_{ij}} \tag{3.12}$$

$$= h_{ij,t} - \alpha \phi_{j,t} \frac{\partial \delta_t}{\partial u_{ij}} - \alpha \delta_t \frac{\partial \phi_{j,t}}{\partial u_{ij}} \qquad \text{Product rule of calculus} \tag{3.13}$$

$$= h_{ij,t} - \alpha \phi_{j,t} \frac{\partial y_t}{\partial u_{ij}} - \alpha \delta_t \frac{\partial \phi_{j,t}}{\partial u_{ij}} \tag{3.14}$$

$$= h_{ij,t} - \alpha \phi_{j,t} \phi_{j,t} h_{ij,t} - \alpha \delta_t \frac{\partial \phi_{j,t}}{\partial u_{ij}} \qquad \text{Plug in Equation (3.8)} \tag{3.15}$$

$$= h_{ij,t} - \alpha \phi_{j,t} \phi_{j,t} h_{ij,t} - \alpha \delta_t \frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \frac{\partial \psi_{j,t}}{\partial u_{ij}} \tag{3.16}$$

(We take approximation by only considering the influence of a samll change of u_{ij} at the time step t rather than all time steps.)

$$\approx (1 - \alpha \phi_{j,t}^2) h_{ij,t} - \alpha \delta_t \frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \frac{\partial \psi_{j,t}}{\partial u_{ij,t}} \tag{3.17}$$

$$= (1 - \alpha \phi_{j,t}^2) h_{ij,t} - \alpha \delta_t x_{i,t} \frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \tag{3.18}$$

Note that as mentioned in Section 2.1, once the nonlinearity over the hidden units g is determined, $\frac{\partial \phi_{j,t}}{\partial \psi_{j,t}}$ is in a closed form.

Equations (2.3), (3.9) and (3.18) form the cross-propagation algorithm for regression, named *Reg-CP*. To summarize,

$$w_{j,t+1} = w_{j,t} - \alpha\delta_t\phi_{j,t}$$

$$u_{ij,t+1} = u_{ij,t} - \alpha\delta_t\phi_{j,t}h_{ij,t}$$

$$h_{ij,t+1} = (1 - \alpha\phi_{j,t}^2)h_{ij,t} - \alpha\delta_t x_{i,t}\frac{\partial\phi_{j,t}}{\partial\psi_{j,t}}$$

Reg-CP is further elaborated in Algorithm 3 (with a *sigmoid* function as the nonlinearity).

Algorithm 3: Reg-CP with *sigmoid*

input: α: step size
n: input dimension
m: amount of hidden units
Initialize h_{ij} to 0
Initialize w_j and u_{ij} as desired where $i = 1, \ldots, n$ and $j = 1, \ldots, m$
for *each new training example* $(\boldsymbol{x}_t, y_t^*)$ **do**
  $\phi_{j,t} \leftarrow sigmoid(\sum_i u_{ij,t}x_{i,t})$
  $y_t \leftarrow \sum_j w_{j,t}\phi_{j,t}$
  $\delta_t \leftarrow y_t - y_t^*$
  for $i = 1, \ldots, n$ **do**
    for $j = 1, \ldots, m$ **do**
      $u_{ij,t+1} \leftarrow u_{ij,t} - \alpha\delta_t\phi_{j,t}h_{ij,t}$
      $h_{ij,t+1} \leftarrow (1 - \alpha\phi_{j,t}^2)h_{ij,t} - \alpha\delta_t(1 - \phi_{j,t})\phi_{j,t}x_{i,t}$
    end
    $w_{j,t+1} \leftarrow w_{j,t} - \alpha\delta_t\phi_{j,t}$
  end
end

In *Reg-CP*, we use an additional memory $\mathbf{H} = \{h_{ij}\}_{i=1,\ldots,n\ j=1,\ldots,m}$ to do gradient descent at all time steps to minimize the generalization loss at the current time step. The update at time step t has to compromise with all previous training examples. As a result, the network is less likely to over-fit recent examples and more likely to learn features in a stable manner than back-propagation. We have to note that here we use an $n \times m$ matrix $\mathbf{H}$, which is computationally inefficient as it involves $n \times m$ updates per time

step. We also need to note that *Reg-CP* is derived under a scalar regression task, where our neural network only has one output unit. The extension to a vector regression task, where we need multiple output units (e.g., k output units), is straightforward. However, then the additional memory will become a 3D tensor of size $n \times m \times k$. To decrease the extra computation, we derived another cross-propagation algorithm which needs only m extra updates per time step for our scalar regression task.

To derive this new cross-propagation algorithm for regression, we first expand the back-propagation update $\frac{\partial l_t}{\partial u_{ij,t}}$ for $u_{ij,t}$ as

$$\frac{\partial l_t}{\partial u_{ij,t}} = \frac{\partial l_t}{\partial \phi_{j,t}} \frac{\partial \phi_{j,t}}{\partial u_{ij,t}} \tag{3.19}$$

We propose to do cross-validation at the feature level directly. We replace the first gradient term $\frac{\partial l_t}{\partial \phi_{j,t}}$ in Equation (3.19) by $\frac{\partial l_t}{\partial \phi_j}$. This new gradient term measures the sensitivity of the current loss l_t to a small change of ϕ_j at all time steps. Now our update for $u_{ij,t}$ is

$$\Delta u_{ij,t} \doteq \frac{\partial l_t}{\partial \phi_j} \frac{\partial \phi_{j,t}}{\partial u_{ij,t}} \tag{3.20}$$

$$= \delta_t \frac{\partial y_t}{\partial \phi_j} x_{i,t} \frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \tag{3.21}$$

$$= \delta_t x_{i,t} \frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \sum_k \frac{\partial y_t}{\partial w_{k,t}} \frac{\partial w_{k,t}}{\partial \phi_j} \tag{3.22}$$

(The primary influence of the j-th feature on the output y_t is through $w_{j,t}$)

$$\approx \delta_t x_{i,t} \frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \frac{\partial y_t}{\partial w_{j,t}} \frac{\partial w_{j,t}}{\partial \phi_j} \tag{3.23}$$

$$= \delta_t x_{i,t} \phi_{j,t} h_{j,t} \frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \tag{3.24}$$

where we define

$$h_{j,t} \doteq \frac{\partial w_{j,t}}{\partial \phi_j}$$

Finally, we have our update rule for $u_{ij,t}$ as

$$u_{ij,t+1} \doteq u_{ij,t} - \alpha \delta_t \phi_{j,t} h_{j,t} x_{i,t} \frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \tag{3.25}$$

We update $h_{j,t}$ incrementally as

$$h_{j,t+1} = \frac{\partial w_{j,t+1}}{\partial \phi_j} \tag{3.26}$$

$$= \frac{\partial (w_{j,t} - \alpha \delta_t \phi_{j,t})}{\partial \phi_j} \qquad \text{Equation (2.3)} \tag{3.27}$$

$$= h_{j,t} - \alpha \frac{\partial \delta_t \phi_{j,t}}{\partial \phi_j} \tag{3.28}$$

$$= h_{j,t} - \alpha \phi_{j,t} \frac{\partial \delta_t}{\partial \phi_j} - \alpha \delta_t \frac{\partial \phi_{j,t}}{\partial \phi_j} \qquad \text{Product rule of calculus} \tag{3.29}$$

$$= h_{j,t} - \alpha \phi_{j,t} \frac{\partial y_t}{\partial \phi_j} - \alpha \delta_t \frac{\partial \phi_{j,t}}{\partial \phi_j} \tag{3.30}$$

$$= h_{j,t} - \alpha \phi_{j,t} \sum_k \frac{\partial y_t}{\partial w_{k,t}} \frac{\partial w_{k,t}}{\partial \phi_j} - \alpha \delta_t \frac{\partial \phi_{j,t}}{\partial \phi_j} \tag{3.31}$$

(The primary influence of the j-th feature ϕ_j on the loss should be through the weight w_j)

$$\approx h_{ij,t} - \alpha \phi_{j,t} \frac{\partial y_t}{\partial w_{j,t}} \frac{\partial w_{j,t}}{\partial \phi_j} - \alpha \delta_t \frac{\partial \phi_{j,t}}{\partial \phi_j} \tag{3.32}$$

(We take approximation by only considering the influence of a small change of the j-th feature at the current time step rather than all the time steps.)

$$\approx h_{j,t} - \alpha \phi_{j,t}^2 h_{j,t} - \alpha \delta_t \frac{\partial \phi_{j,t}}{\partial \phi_{j,t}} \tag{3.33}$$

$$= (1 - \alpha \phi_{j,t}^2) h_{j,t} - \alpha \delta_t \tag{3.34}$$

Now we have our update rule for $h_{j,t}$,

$$h_{j,t+1} = (1 - \alpha \phi_{j,t}^2) h_{j,t} - \alpha \delta_t \tag{3.35}$$

Equations (2.3), (3.25) and (3.35) together form anther cross-propagation algorithm for regression, named *Reg-CP-Alt*. To summarize,

$$\begin{aligned} w_{j,t+1} &= w_{j,t} - \alpha \delta_t \phi_{j,t} \\ u_{ij,t+1} &= u_{ij,t} - \alpha \delta_t \phi_{j,t} h_{j,t} x_{i,t} \frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \\ h_{j,t+1} &= (1 - \alpha \phi_{j,t}^2) h_{j,t} - \alpha \delta_t \end{aligned}$$

Reg-CP-Alt is further elaborated in Algorithm 4 (with a *sigmoid* function as the nonlinearity).

Algorithm 4: Reg-CP-Alt with *sigmoid*

input: α: step size
n: input dimension
m: amount of hidden units
Initialize h_j to 0
Initialize w_j and u_{ij} as desired where $i = 1, \ldots, n$ and $j = 1, \ldots, m$
for *each new training example* $(\boldsymbol{x}_t, y_t^*)$ **do**
  $\phi_{j,t} \leftarrow sigmoid(\sum_i u_{ij,t} x_{i,t})$
  $y_t \leftarrow \sum_j w_{j,t} \phi_{j,t}$
  $\delta_t \leftarrow y_t - y_t^*$
  for $i = 1, \ldots, n$ **do**
    for $j = 1, \ldots, m$ **do**
      $u_{ij,t+1} \leftarrow u_{ij,t} - \alpha \delta_t \phi_{j,t}^2 h_{j,t} (1 - \phi_{j,t}) x_{i,t}$
    end
    $w_{j,t+1} \leftarrow w_{j,t} - \alpha \delta_t \phi_{j,t}$
  end
  for $j = 1, \ldots, m$ **do**
    $h_{j,t+1} \leftarrow (1 - \alpha \phi_{j,t}^2) h_{j,t} - \alpha \delta_t$
  end
end

Here *Reg-CP-Alt* is derived under a scalar regression task where we only have one output unit. We need m extra updates per time step. The extension of *Reg-CP-Alt* to multiple output units (e.g., k output units) is straightforward. However, we then need $m \times k$ extra updates per time step. *Reg-CP-Alt* reduces the required memory and computation compared with *Reg-CP*, and our preliminary experiments showed that *Reg-CP-Alt* almost always had similar learning curves as *Reg-CP*.

In the rest of this section, we derive the cross-propagation algorithm for classification tasks similar to *Reg-CP-Alt*. We consider the same online classification task and corresponding function parameterization as described in Section 2.1. At time step t, the agent receives an observation $\mathbf{x}_t \in \mathbb{R}^n$. It tires to predict the target label $\mathbf{p}_t^* \in \{0, 1\}^d$, where $\mathbf{p}_t^*$ is a one-hot vector of length d encoding a class label. We use $x_{i,t}$ and $p_{i,t}^*$ to denote the i-th element of $\mathbf{x}_t$ and $\mathbf{p}_t^*$ respectively. Our prediction $\mathbf{p}_t \in [0, 1]^d$ is a probability distribution of

the d class labels and is computed via a learnable function $f : \mathbb{R}^n \to [0,1]^d$. We use cross-entropy loss to measure the prediction error l_t at time step t,

$$l_t \doteq l(\mathbf{p}_t, \mathbf{p}_t^*) \doteq -\sum_{k=1}^{d} p_{k,t}^* \log p_{k,t}$$

We parameterize the learnable function f as a neural network, which consists of m hidden units, the incoming weight matrix $\mathbf{U} \in \mathbb{R}^{n \times m}$, the outgoing weight matrix $\mathbf{W} \in \mathbb{R}^{m \times d}$. We apply a nonlinear function g over the hidden units and a softmax operation over the output units. Our prediction at time step t is

$$\mathbf{p}_t = \text{softmax}(\mathbf{y}_t)$$

in other words,

$$p_{k,t} = \frac{e^{y_{k,t}}}{\sum_{j=1}^{d} e^{y_{j,t}}}$$

where

$$y_{k,t} = \sum_{j=1}^{m} \phi_{j,t} w_{jk,t}$$

where $\phi_{j,t}$ is the j-th feature in the hidden layer as defined in Equation (3.2). We still use $\psi_{j,t}$ to denote the value of the j-th hidden unit before applying the non-linear function g as defined in Equation (3.3).

We update $w_{jk,t}$ in the same way as *Cls-BP*, in other words,

$$w_{jk,t+1} \doteq w_{jk,t} - \alpha \delta_{k,t} \phi_{j,t} \tag{3.36}$$

To update $u_{ij,t}$, we first expand the back-propagation update $\frac{\partial l_t}{\partial u_{ij,t}}$ as

$$\frac{\partial l_t}{\partial u_{ij,t}} = \frac{\partial l_t}{\partial \phi_{j,t}} \frac{\partial \phi_{j,t}}{\partial u_{ij,t}} \tag{3.37}$$

Similar to *Reg-CP-Alt*, we replace $\frac{\partial l_t}{\partial \phi_{j,t}}$ by $\frac{\partial l_t}{\partial \phi_j}$ to do cross-validation at feature level. This new gradient term measures the sensitivity of the current loss l_t to

a small change of ϕ_j at all time steps. Now our update for $u_{ij,t}$ is

$$\Delta u_{ij,t} \doteq \frac{\partial l_t}{\partial \phi_j}\frac{\partial \phi_{j,t}}{\partial u_{ij,t}} \tag{3.38}$$

$$= \frac{\partial \phi_{j,t}}{\partial u_{ij,t}} \sum_{k=1}^{d} \frac{\partial l_t}{\partial y_{k,t}}\frac{\partial y_{k,t}}{\partial \phi_j} \tag{3.39}$$

$$= x_{i,t}\frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \sum_{k=1}^{d} \delta_{k,t}\frac{\partial y_{k,t}}{\partial \phi_j} \qquad \text{Plug in Equation (2.7)} \tag{3.40}$$

$$= x_{i,t}\frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \sum_{k=1}^{d} \delta_{k,t} \sum_{s=1}^{m} \frac{\partial y_{k,t}}{\partial w_{sk,t}}\frac{\partial w_{sk,t}}{\partial \phi_j} \tag{3.41}$$

(The primary influence of the j-the feature on the $y_{k,t}$ is through $w_{jk,t}$)

$$\approx x_{i,t}\frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \sum_{k=1}^{d} \delta_{k,t}\frac{\partial y_{k,t}}{\partial w_{jk,t}}\frac{\partial w_{jk,t}}{\partial \phi_j} \tag{3.42}$$

$$= x_{i,t}\frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \sum_{k=1}^{d} \delta_{k,t}\phi_{j,t}h_{jk,t} \tag{3.43}$$

where we define

$$h_{jk,t} \doteq \frac{\partial w_{jk,t}}{\partial \phi_j}$$

Finally, we have our update rule for $u_{ij,t}$ as

$$u_{ij,t+1} \doteq u_{ij,t} - \alpha\phi_{j,t}x_{i,t}\frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \sum_{k=1}^{d} \delta_{k,t}h_{jk,t} \tag{3.44}$$

We update $h_{jk,t}$ incrementally as

$$h_{jk,t+1} = \frac{\partial w_{jk,t+1}}{\partial \phi_j} \tag{3.45}$$

$$= \frac{\partial (w_{jk,t} - \alpha\delta_{k,t}\phi_{j,t})}{\partial \phi_j} \tag{3.46}$$

$$= h_{jk,t} - \alpha\frac{\partial \delta_{k,t}}{\partial \phi_j}\phi_{j,t} - \alpha\delta_{k,t}\frac{\partial \phi_{j,t}}{\partial \phi_j} \tag{3.47}$$

$$= h_{jk,t} - \alpha\frac{\partial p_{k,t}}{\partial \phi_j}\phi_{j,t} - \alpha\delta_{k,t}\frac{\partial \phi_{j,t}}{\partial \phi_j} \tag{3.48}$$

$$= h_{jk,t} - \alpha\sum_{s=1}^{d}\frac{\partial p_{k,t}}{\partial y_{s,t}}\frac{\partial y_{s,t}}{\partial \phi_j}\phi_{j,t} - \alpha\delta_{k,t}\frac{\partial \phi_{j,t}}{\partial \phi_j} \tag{3.49}$$

$$= h_{jk,t} - \alpha\sum_{s=1}^{d} p_{k,t}(\mathbb{I}_{k=s} - p_{s,t})\frac{\partial y_{s,t}}{\partial \phi_j}\phi_{j,t} - \alpha\delta_{k,t}\frac{\partial \phi_{j,t}}{\partial \phi_j} \qquad \text{Equation (2.6)} \tag{3.50}$$

(We take approximation by only considering the influence of a small change of the j-th feature at the current time step rather than all the time steps.)

$$\approx h_{jk,t} - \alpha \sum_{s=1}^{d} p_{k,t}(\mathbb{I}_{k=s} - p_{s,t}) \frac{\partial y_{s,t}}{\partial \phi_j} \phi_{j,t} - \alpha \delta_{k,t} \frac{\partial \phi_{j,t}}{\partial \phi_{j,t}} \tag{3.51}$$

(The primary influence of ϕ_j on the $y_{s,t}$ is mainly through $w_{js,t}$)

$$\approx h_{jk,t} - \alpha \sum_{s=1}^{d} p_{k,t}(\mathbb{I}_{k=s} - p_{s,t}) \frac{\partial y_{s,t}}{\partial w_{js,t}} \frac{\partial w_{js,t}}{\partial \phi_j} \phi_{j,t} - \alpha \delta_{k,t} \tag{3.52}$$

$$= h_{jk,t} - \alpha \sum_{s=1}^{d} p_{k,t}(\mathbb{I}_{k=s} - p_{s,t}) \phi_{j,t} h_{js,t} \phi_{j,t} - \alpha \delta_{k,t} \tag{3.53}$$

$$= h_{jk,t} - \alpha \phi_{j,t}^2 \sum_{s=1}^{d} p_{k,t}(\mathbb{I}_{k=s} - p_{s,t}) h_{js,t} - \alpha \delta_{k,t} \tag{3.54}$$

Finally, the update rule for $h_{jk,t}$ is

$$h_{jk,t+1} = h_{jk,t} - \alpha \phi_{j,t}^2 \sum_{s=1}^{d} p_{k,t}(\mathbb{I}_{k=s} - p_{s,t}) h_{js,t} - \alpha \delta_{k,t} \tag{3.55}$$

Equations (3.36), (3.44) and (3.55) together form the cross-propagation algorithm for classification, named *Cls-CP*. To summarize,

$$\begin{aligned}
w_{jk,t+1} &= w_{jk,t} - \alpha \delta_{k,t} \phi_{j,t} \\
u_{ij,t+1} &= u_{ij,t} - \alpha \phi_{j,t} x_{i,t} \frac{\partial \phi_{j,t}}{\partial \psi_{j,t}} \sum_{k=1}^{d} \delta_{k,t} h_{jk,t} \\
h_{jk,t+1} &= h_{jk,t} - \alpha \phi_{j,t}^2 \sum_{s=1}^{d} p_{k,t}(\mathbb{I}_{k=s} - p_{s,t}) h_{js,t} - \alpha \delta_{k,t}
\end{aligned}$$

Cls-CP is further elaborated in Algorithm 5 (with a *sigmoid* function as the nonlinearity).

3.3 Experimental Results

In this section, we present some experimental results of the three cross-propagation algorithms in both online setting and off-line setting. We start with an introduction to our testbeds.

We consider two tasks as our testbeds: the GEneric Online Feature Finding (GEOFF) task and the MNIST dataset. GEOFF task was first introduced by

Algorithm 5: Cls-CP with *sigmoid*

input: α: step size
n: input dimension
m: number of hidden units
d: number of labels

Initialize h_{jk} to 0

Initialize w_{ij} and u_{ij} as desired where $i = 1, \ldots, n$, $j = 1, \ldots, m$ and $k = 1, \ldots, d$

for *each new training example* $(\boldsymbol{x}_t, \boldsymbol{p}_t^*)$ **do**

$\phi_{j,t} \leftarrow sigmoid(\sum_i u_{ij,t} x_{i,t})$

$y_{k,t} \leftarrow \sum_j w_{jk,t} \phi_{j,t}$

$\boldsymbol{p}_t = softmax(\boldsymbol{y}_t)$

$\boldsymbol{\delta}_t \leftarrow \boldsymbol{p}_t - \boldsymbol{p}_t^*$

for $i = 1, \ldots, n$ **do**

for $j = 1, \ldots, m$ **do**

$u_{ij,t+1} \leftarrow u_{ij,t} - \alpha \phi_{j,t}^2 (1 - \phi_{j,t}) x_{i,t} \sum_{k=1}^{d} \delta_{k,t} h_{jk,t}$

end

end

for $j = 1, \ldots, m$ **do**

for $k = 1, \ldots, d$ **do**

$w_{jk,t+1} \leftarrow w_{jk,t} - \alpha \delta_{k,t} \phi_{j,t}$

$h_{jk,t+1} \leftarrow h_{jk,t} - \alpha \phi_{j,t}^2 \sum_{s=1}^{d} p_{k,t} (\mathbb{I}_{k=s} - p_{s,t}) h_{js,t} - \alpha \delta_{k,t}$

end

end

end

Sutton (2014) as a generic, synthetic, feature-finding testbed for evaluating different representation learning algorithms. The primary advantage of this testbed is that infinitely many supervised-learning training examples can be generated without any experimenter bias. One GEOFF task consists of a single hidden layer neural network (named the GEOFF target network) with n input units, m hidden units and one output units. Each input example $\mathbf{x}_t \in \{0,1\}^n$ is an n-dimensional binary input vector where each element in the vector can take a value of 0 or 1. The incoming weights of the GEOFF target network is $\mathbf{U}^* \in \{-1,+1\}^{n\times m}$, and the outgoing weights of the GEOFF target network is $\mathbf{w}^* \in \{-1,0,+1\}^m$. Each element u^*_{ij} and w^*_j is chosen randomly and remains fixed after initialization. The hidden layer of the GEOFF target network consists of m Linear Threshold Units (LTUs) and is denoted as $\boldsymbol{\phi}^* \in \{0,1\}^m$. This particular form of network is adapted from Sutton & Whitehead (1993). Each feature $\phi^*_{j,t}$ is computed as $\phi^*_{j,t} \doteq \mathbb{I}_{\psi^*_{j,t}>\theta_j}$, where θ_j is a threshold parameter and $\psi^*_{j,t} \doteq \sum_{i=1}^n x_{i,t}u^*_{ij}$. Each unit ϕ^*_j has an input pattern that maximizes ψ^*_j. Sutton & Whitehead (1993) named this pattern the prototype of the feature ϕ^*_j. We set θ_j in such a way that the j-th feature ϕ^*_j is valued 1 only when at least β proportion of the input bits matches the prototype of the feature. This can be achieved by setting the threshold as $\theta_j \doteq n\beta - S_j$, where S_j is the number of input weights with a value of -1 connected to the j-th feature (Sutton & Whitehead, 1993). For each input vector $\mathbf{x}_t$, the GEOFF target network is used to produce a scalar target output $y^*_t = \sum_{i=1}^m \phi^*_{i,t}w^*_i + \epsilon_t$, where $\epsilon_t \sim \mathcal{N}(0,1)$ is a random Gaussian noise. For our experiments, we always set $n = 20$, $m = 1000$ and $\beta = 0.6$ for the GEOFF target network.

The MNIST dataset of handwritten digits was introduced by LeCun (1998). Though the MNIST dataset is old, it is still viewed as a standard supervised learning benchmark task for testing out new learning algorithms (Sironi, Tekin, Rigamonti, Lepetit, & Fua, 2015; Papernot et al., 2016). The dataset consists

of grayscale images each with 28×28 dimensions. These images are obtained from handwritten digits, and their corresponding labels denote the supervised learning target for a given image. The objective of a learning system in an MNIST task is to learn a mapping function that maps each of these images to a label.

3.3.1 Experiment 1: Online Learning of Related Tasks

We now present our experimental results in online setting. The experiments were designed from a continual learning perspective where a learning system experiences examples generated by a sequence of related tasks, and learning one task helps learn other similar tasks. In this section, we use capitals A, B and C to index our tasks.

We first show the results of the GEOFF task. For our experiments, we first created a GEOFF target network. Then we generated a dataset by feeding 5,000 randomly generated input vectors to the GEOFF target network. The 5,000 input vectors and corresponding targets computed by the GEOFF target network formed Task A. Then we produced a new GEOFF target network by randomly choosing and regenerating 50% of the outgoing weights $\mathbf{w}^*$ of the GEOFF target network used in Task A. This new GEOFF target network was used to compute targets for another 5,000 randomly generated input vectors. These new 5,000 examples formed Task B. Similarly, Task C was produced by regenerating 50% of the outgoing weights of the GEOFF target network used in Task B. It is important to point out here that all these tasks share the same feature representation (i.e., the weight $\mathbf{U}^*$ remains fixed throughout). We presented our evaluated learning system with a sequence of the three tasks, {Task A, Task B, Task C}, in an online manner. To be more specific, the $5000 \times 3 = 15000$ examples in the three tasks were presented to the learning system one by one. Each example was seen by the learning system only once

and then discarded.

The learning network of our evaluated learning system was a single hidden layer neural network with a single output unit. It had 20 input units and 1,000 hidden units with a *sigmoid* activation function. Note that all the GEOFF target networks also have 1,000 features. So the learning system can solve all the tasks with one feature representation. The squared error was used for learning the parameters of this network. The weights of the learning network were initialized via a Gaussian distribution $\mathcal{N}(0, 1.0)$. We used 4 algorithms to train the network: *Reg-BP* (defined in Section 2.1), *Reg-BP with fixed features* (i.e., the incoming weight $\mathbf{U}$ remains fixed during the training, only $\mathbf{w}$ is updated), *Reg-CP* (Algorithm 3) and *Reg-CP-Alt* (Algorithm 4). We selected the best step size from $\{1.0, 0.5, 0.1, \ldots, 10^{-4}\}$ for each algorithm based on the mean error of the last 1,000 examples in Task C.

Figure 3.2(a) shows the training progression. In the first stage (Task A), *Reg-BP with fixed features* learned fastest. It is within expectation as the learning network had 1,000 hidden units, so it might already contain some good features immediately after the initialization. Learning the outgoing weights $\mathbf{w}$ may be enough to achieve a reasonable performance level. *Reg-BP* also learned faster than the two cross-propagation algorithms, as *Reg-BP* was directly minimizing the mean squared error while the cross-propagation algorithms did cross-validation. This phenomenon is similar to what we usually observe in off-line supervised learning. If we want to achieve better performance at a validation set, we may have to sacrifice the performance at the training set. When we switched to Task B and Task C, the learning speed of the two cross-propagation algorithms caught up with *Reg-BP*. And *Reg-CP* was even slightly faster than *Reg-BP*. We hypothesize that this is because cross-validation helped the cross-propagation algorithms avoid the over-fitting of features. The cross-propagation algorithms learned the features in a more

stable way than back-propagation. And features learned from Task A helped the learning of Task B and Task C more in the cross-propagation algorithms than *Reg-BP*.

To verify this hypothesis, we plot the weight change of the learning network during the training. To be more specific, after processing the t-th training example, we compute the L_2 distance between the current weights $\mathbf{U}_t$ and initial weights $\mathbf{U}_0$, then we normalize this L_2 distance by the learning rate and the number of elements in the weight matrix. We plot this normalized L_2 distance of the incoming weights $\mathbf{U}$ against the training examples in Figure 3.2(b). We show the change of the outgoing weights $\mathbf{w}$ in the same manner in Figure 3.2(c). In the plots, the curves show the normalized absolute weight change, and the slope shows the rate of weight changing. From Figure 3.2(b), we can see at the very beginning of the learning, *Reg-BP* changed the feature layer (i.e., $\mathbf{U}$) rapidly, while the two cross-propagation algorithms were learning features in a stable way (i.e., the slope is almost constant). The behavior of the two cross-propagation algorithms seems to make more sense than *Reg-BP* in our online learning of related tasks because as indicated by the learning curve of *Reg-BP with fixed features*, there might be some good features immediately after initialization. So it may be better for an algorithm to take advantage of the initialization rather than change the feature layer rapidly. From Figure 3.2(c), we can see that there was nothing special for *Reg-BP* at task switch, while the curves for the two cross-propagation algorithms show a clear pattern at task switch. This phenomena also seems to make sense in our online learning of related tasks. Because all the tasks can be solved by one feature representation, the most important thing to do at task switch may be to relearn how to combine the features (i.e., to adjust the second layer).

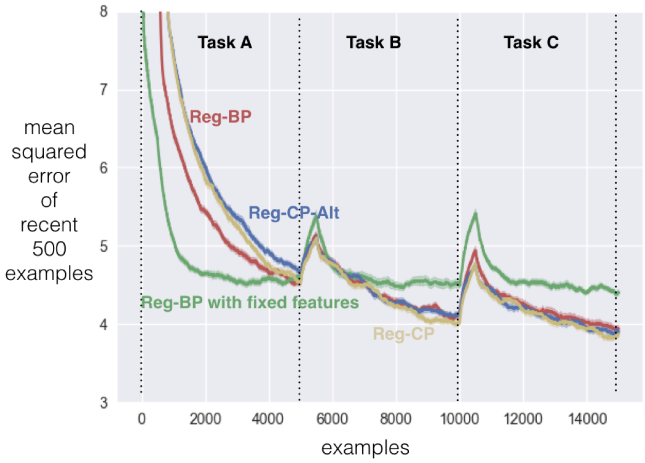
Although a good algorithm should learn features in a stable way in our continual learning experiments, learning feature stably is not sufficient to guaran-

tee a good performance. There are many trivial ways to achieve a stable learning of features (e.g., freezing the feature layer like *Reg-BP with fixed features* or two-time scale learning), but the cross-propagation algorithms achieved this in a non-trivial way.

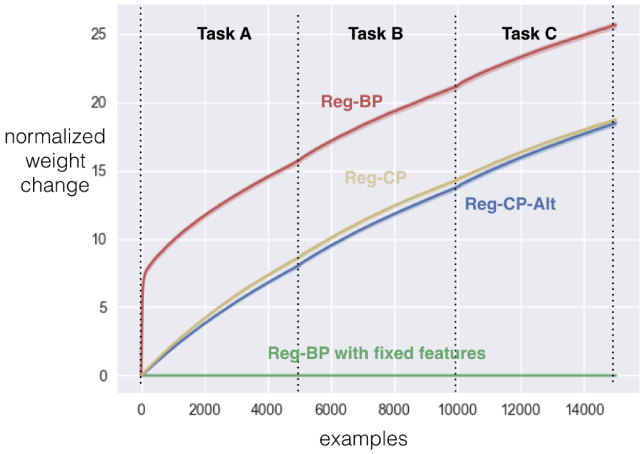
It is also interesting to see that *Reg-CP* outperformed *Reg-CP-Alt* in terms of the learning speed. We hypothesize there is a trade-off between space and performance. *Reg-CP* does cross-validation at the weight level. It measures how a single weight u_{ij} influences the loss. While *Reg-CP-Alt* does cross-validation at the feature level, it measures how a single feature ϕ_j, which is influenced by a set of input weights $\{u_{1j}, \ldots, u_{nj}\}$, influences the loss. This is the reason why *Reg-CP-Alt* requires less memory. However, this did hurt the performance according to the empirical results.

Similar to our online GEOFF tasks, we performed experiments on the MNIST dataset. We built three tasks {Task A, Task B, Task C} upon the MNIST dataset. In each task, the label for the training images was shifted by one. For example, Task A used the standard MNIST training images and their labels. Task B used the same training examples as Task A, but the labels got shifted by one. Similarly, for Task C the labels for the training examples got further shifted by one from labels of Task B. In this way, the three tasks share a common feature representation, the only thing to learn when switching tasks is the way to combine the features. Each task consisted of 5,000 images and corresponding labels. We presented the learning system with a sequence of the three tasks, {Task A, Task B, Task C}, in an online manner. We used 3 algorithms: *Cls-BP* (defined in Section 2.1), *Cls-BP with fixed features*(i.e., the incoming weight $\mathbf{U}$ remains fixed during training, only $\mathbf{w}$ is updated) and *Cls-CP* (Algorithm 5), and the step size for each algorithm was selected in the same manner as the online GEOFF experiments.

The learning network was a single hidden layer neural network with 784



(a) learning curves



(b) change of U

(c) change of W

Figure 3.2: Learning curves for the online GEOFF task. Figure (a) plots the mean squared error of recent 500 examples during training. Figures (b) and (c) show the normalized L_2 norm of the weight change of $\mathbf{U}$ and $\mathbf{w}$ during training. All the curves are averaged over 30 independent runs where each run uses a different initial target network and different random input vectors. Standard errors are plotted as the shadow.

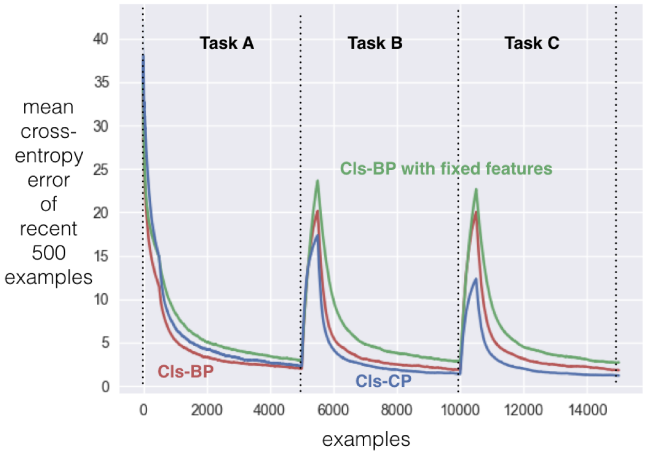
input units, 1,024 hidden units and 10 output units. The hidden units used a *sigmoid* activation function, and the output units used a softmax activation function. Cross-entropy error was used for training the network. The weights of the learning networks were initialized via a Gaussian distribution $\mathcal{N}(0, 1.0)$.

Figure 3.3(a) shows the training progression. The trend is similar to what we observed in the online GEOFF tasks. In the first stage (Task A), *Cls-BP with fixed features* learned fastest, and *Cls-BP* outperformed *Cls-CP*. However, when we switched to Task B and Task C, *Cls-CP* learned faster than *Cls-BP*. We also plot the change of $\mathbf{U}$ and $\mathbf{W}$ in the same manner as mentioned before in Figures 3.3 (b) and (c). From the slope, we can see *Cls-CP* learned the features in a more stable way and changed the last layer (i.e., the layer in charge of how to combine features) faster than *Cls-BP*.

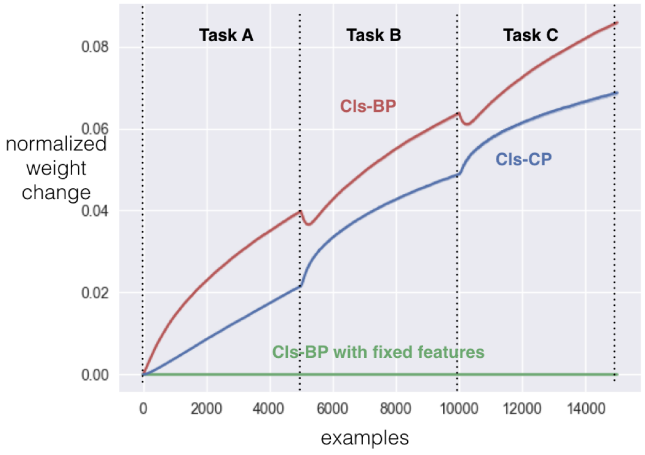
3.3.2 Experiment 2: Off-line Supervised Learning

Although we derived the three cross-propagation-based algorithms in online setting, they can also be used in off-line setting. To be more specific, in off-line setting the network is trained on a training set for multiple sweeps, and we evaluate the network on an unseen test set when training is done. As discussed in Section 2.2, if the network has more parameters, it is more complicated and usually needs more data for training. Otherwise, the network may fall into over-fitting. In the rest of this section, we present some experimental results, revealing how the number of training examples and the network complexity (i.e., the number of parameters) influence the asymptotic performance of the network for different training algorithms.

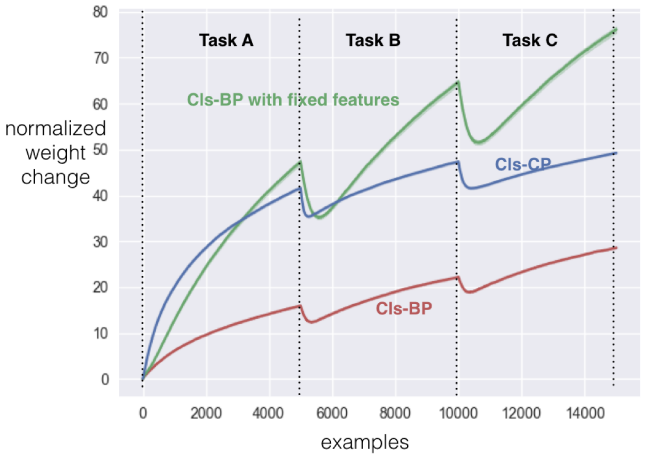
Our testbed is still the GEOFF task. Different from the online setting, we only used one GEOFF target network in offline experiments. We generated different amounts of training examples (3000, 6000, 15000 and 24000) and 500 test examples respectively from this GEOFF target network. We used



(a) learning curves



(b) change of U



(c) change of W

Figure 3.3: Learning curves for the online MNIST task. Figure (a) plots the mean cross entropy error of recent 500 examples during training. Figures (b) and (c) show the normalized L_2 norm of the weight change of $\mathbf{U}$ and $\mathbf{W}$ during training. All the curves are averaged over 30 independent runs, and standard errors are plotted as the shadow.

three single $ReLU$ hidden layer networks with a different number of hidden units (100, 500 and 900). We used *Reg-CP-Alt* (Algorithm 4) and *Reg-BP* to train the networks in a supervised learning manner. To be more specific, we trained the network with the training set for 200 sweeps. Then we evaluated its performance on the corresponding unseen test dataset. We iterated all the $4 \times 3 \times 2 = 24$ combinations of the number of training examples, the number of hidden units and the training algorithms. For each combination, we selected the best step size from $\{2^{-16}, \ldots, 2^{-1}\}$ according to the mean prediction error of the last 5 sweeps during training. All the results were averaged over 30 independent runs.

We use $error_{cp}$ and $error_{bp}$ to denote the asymptotic test error (i.e., the mean prediction error on the test set after 200 sweeps on the training set) of *Reg-CP-Alt* and *Reg-BP* respectively. Table 3.1 shows the value of $error_{cp} - error_{bp}$. A positive number indicates that *Reg-BP* performed better than *Reg-CP-Alt*, and a negative number indicates that *Reg-CP-Alt* performed better than *Reg-BP*.

From the first row, we can see that with 100 hidden units and 3,000 training examples, *Reg-BP* performed better than *Reg-CP-Alt*. However, when we increased the number of hidden units to 500, *Reg-CP-Alt* outperformed *Reg-BP*. When we kept increasing the number of hidden units to 900, the difference got larger. When we increase the number of hidden units, we increase the network complexity. So we need more data to train the network. However, the number of training examples is fixed, so the network tends to over-fit the training data as we increase its complexity, where *Reg-CP-Alt* started to outperform *Reg-BP*. This trend also occurs in the other three rows.

From the second column, we can see that with 500 hidden units and 3,000 training examples, *Reg-CP-Alt* performed better than *Reg-BP*. When we increased the number of training examples while keeping the network complexity

hidden units / training examples	100	500	900
3000	3.65	-2.44	-6.47
6000	3.96	-1.83	-4.77
15000	5.08	0.02	-1.86
24000	5.36	0.33	-0.77

Table 3.1: Performance gap between cross-propagation and back-propagation in various combinations in the GEOFF task.

(i.e., the number of hidden units) fixed, *Reg-BP* started to outperform *Reg-CP-Alt*. With more training data, over-fitting is less likely to happen, and *Reg-BP* performed better than *Reg-CP-Alt*. Other two columns also show similar trends.

It is an interesting observation that when training data was insufficient compared with network complexity, where over-fitting is likely to happen, the cross-propagation-based algorithm performed better than the back-propagation algorithm. However, when we had sufficient training data, the back-propagation algorithm outperformed the cross-propagation-based algorithm.

3.4 Weakness of the Cross-propagation Algorithms

We applied the cross-propagation technique to train the feature layer of a single hidden layer neural network and showed its merit empirically in both online and off-line setting. However, the merit comes at a cost.

The most critical flaw of our three cross-propagation-based algorithms is that they require more memory than corresponding back-propagation algorithms. And although a naive extension of our three cross-propagation-based algorithms to deeper networks is straightforward, the required memory will then increase exponentially according to the increase of the depth of the network. Until now we do not have an efficient way to extend our cross-

propagation-based algorithms to modern deep network architectures.

We should also remind the reader that the results in off-line setting are sensitive to various factors. We did not see the same phenomena with a *tanh* activation function nor on the MNIST dataset. So the exact relationship behind cross-propagation and over-fitting remains unclear.

Chapter 4

Evaluation of Experience Replay

This chapter elaborates the second contribution of this thesis in detail. We first show some open questions about experience replay, after which we introduce our new experience replay method. Finally, we present our systematic evaluation of experience replay under various function representations.

4.1 Open Questions

In this section, we describe some open questions about experience replay. In modern deep RL systems, experience replay is mainly used to stabilize the training of neural network function approximators. Experience replay also improves data efficiency (Lin, 1992; Wang et al., 2016), which is often a desirable property as many RL algorithms are pretty hungry for data. Experience replay itself was proposed in the early age of reinforcement learning when tabular methods and linear function approximation dominated the field, but experience replay did not draw much attention until the success of DQN, after which experience replay become an essential component in many deep RL algorithms (Lillicrap et al., 2015; Andrychowicz et al., 2017). There may be some defect that prevents experience replay from being widely used in the pre-deep-RL era. However, to our best knowledge, no previous work has pointed out what is wrong with experience replay.

Experience replay introduces a new hyper-parameter, the memory size. As far as we know, no previous work has systematically studied how this new hyper-parameter influences the performance. The community seems to have a default value for the memory size, 10^6. For instance, Mnih et al. (2015) set the memory size for DQN to 10^6 for various Atari games (Bellemare, Naddaf, Veness, & Bowling, 2013), after which Lillicrap et al. (2015) also set the memory size for Deep Deterministic Policy Gradient (DDPG) to 10^6 to address various Mujoco tasks (Todorov, Erez, & Tassa, 2012). And Andrychowicz et al. (2017) set the memory size to 10^6 in their Hindsight Experience Replay (HER) for a physical robot arm, and Tassa et al. (2018) used a memory of size 10^6 to solve the tasks in DeepMind Control Suite. In the aforementioned papers, the tasks vary from simulation environments to real-world robots, and the function approximators vary from shallow fully-connected networks to deep convolutional networks. However, they all used the same memory size. Liu & Zou (2017) did a theoretical study on how the memory size influences the performance. However, their analytical study only applies to an ordinary differential equation model, and their experiments did not handle the episode end by timeout properly.

In this thesis, we did a systematic evaluation of experience replay, especially the memory size.

4.2 Combined Experience Replay

Before we present our evaluation, we first introduce a new experience replay method. In the original experience replay, the transition at the current step is not used for training the agent immediately. Only sampled transitions from the memory are used to train the agent at each time step. In this thesis, we propose to train the agent with both the current transition and the sampled transitions. We name our proposed new experience replay method *combined*

experience replay (CER). We show that a learning system with CER is more robust to the selection of the memory size compared with the one with the original experience replay.

CER is similar to a component of prioritized experience replay (PER, Schaul et al., 2015). In PER, Schaul et al. (2015) gave the current transition the largest priority. However, PER is still a stochastic replay method, which means giving the current transition the largest priority does not guarantee the current transition will be replayed immediately. And it is important to note that PER and CER are aimed to solve different problems. To be more specific, CER is designed to make a learning system less sensitive to the selection of the memory size, while PER is designed to replay transitions in the memory more efficiently. If the memory size is set properly, we do not expect CER can further improve performance. However, PER is always expected to improve the performance. Although there is a similar part to CER in PER (giving the largest priority to the current transition), PER never shows how that part interacts with the memory size and whether that part itself makes a significant contribution to the whole learning system. Furthermore, with the increase of the memory size, the required computation of PER increases logarithmically, while that of CER remains constant. In addition, PER often introduces some complicated data structures (e.g., a sum-tree), which may need much engineer effort. CER, however, requires little extra engineer effort.

4.3 Evaluation Setup

Experience replay itself is not a complete learning algorithm. Experience replay has to be combined with some other off-policy algorithms to form a learning system. Following Mnih et al. (2015), we consider the combination of Q-learning and experience replay in our evaluation.

We compared three learning systems, and each learning system had three

kinds of function representations (i.e., tabular function representation, linear function representation, and non-linear function representation). The three learning systems are Q-Learning with the current transition (referred to as Online-Q, Algorithms 6, 7, and 8), Q-Learning with the original experience replay (referred to as Buffer-Q, Algorithms 9, 10, and 11) and Q-Learning with CER (referred to as Combined-Q, Algorithms 12, 13, and 14). Online-Q is exactly the primitive Q-Learning, where the current transition at every time step is used to update the value function immediately. Buffer-Q refers to DQN-like Q-Learning, where the current transition is not used to update the value function immediately. Instead, the current transition is stored into the memory, and only sampled transitions from the memory are used for updating the value function. Combined-Q uses both the current transition and the transitions from the memory to update the value function at every time step. For each of the 9 algorithms, we varied the memory size (if the algorithm has a memory) in a large range to investigate how the memory size influences the performance. When varying the memory size of an algorithm, all other hyper-parameters remained fixed.

Algorithm 6: Online-Q with tabular function representation

Initialize the value function Q
while *not converged* **do**
  Get the initial state S
  while S *is not the terminal state* **do**
    Select an action A according to an ϵ-greedy policy derived from Q
    Execute the action A, get the reward R and the next state S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big(R + \gamma \max_a Q(S',a) - Q(S,A)\big)$
    $S \leftarrow S'$
  end
end

We used three tasks to evaluate the aforementioned algorithms: a grid world, the Lunar Lander and the Atari game Pong. Figure 4.1 elaborates the

Algorithm 7: Online-Q with linear function representation

Input: a state-action feature function $\mathbf{x}$
Initialize the weights $\mathbf{w}$
while *not converged* **do**
  Get the initial state S
  while S *is not the terminal state* **do**
    Select an action A according to an ϵ-greedy policy derived from $\mathbf{w}$
    Execute the action A, get the reward R and the next state S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big(R + \gamma \max_a \mathbf{w}^{\mathrm{T}}\mathbf{x}(S', a) - \mathbf{w}^{\mathrm{T}}\mathbf{x}(S, A)\big)\mathbf{x}(S, A)$
    $S \leftarrow S'$
  end
end

Algorithm 8: Online-Q with non-linear function representation

Initialize the weights $\mathbf{w}$
while *not converged* **do**
  Get the initial state S
  while S *is not the terminal state* **do**
    Select an action A according to an ϵ-greedy policy derived from $\mathbf{w}$ and $\hat{q}$
    Execute the action A, get the reward R and the next state S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big(R + \gamma \max_a \hat{q}(S', a, \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
  end
end

Algorithm 9: Buffer-Q with tabular function representation

Initialize the value function Q
Initialize the replay buffer
while *not converged* **do**
  Get the initial state S
  while S *is not the terminal state* **do**
    Select an action A according to a ϵ-greedy policy derived from Q
    Execute the action A, get the reward R and the next state S'
    Store the online transition (S, A, R, S') into the memory
    Sample a batch of transitions $\mathcal{B}$ uniformly from the memory
    for *each transition* (s, a, r, s') *in* $\mathcal{B}$ **do**
      $Q(s, a) \leftarrow Q(s, a) + \alpha\big(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\big)$
    end
    $S \leftarrow S'$
  end
end

Algorithm 10: Buffer-Q with linear function representation

Input: a state-action feature function $\mathbf{x}$
Initialize the weights $\mathbf{w}$
Initialize the replay buffer
while *not converged* **do**
  Get the initial state S
  while S *is not the terminal state* **do**
    Select an action A according to a ϵ-greedy policy derived from $\mathbf{w}$
    Execute the action A, get the reward R and the next state S'
    Store the online transition (S, A, R, S') into the memory
    Sample a batch of transitions $\mathcal{B}$ uniformly from the memory
    for *each transition* (s, a, r, s') *in* $\mathcal{B}$ **do**
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big(r + \gamma \max_{a'} \mathbf{w}^{\mathrm{T}}\mathbf{x}(s', a') - \mathbf{w}^{\mathrm{T}}\mathbf{x}(s, a)\big)\mathbf{x}(s, a)$
    end
    $S \leftarrow S'$
  end
end

Algorithm 11: Buffer-Q with non-linear function representation

Initialize the weights $\mathbf{w}$
Initialize the replay buffer
while *not converged* **do**
  Get the initial state S
  while S *is not the terminal state* **do**
    Select an action A according to a ϵ-greedy policy derived from $\hat{q}$
    Execute the action A, get the reward R and the next state S'
    Store the online transition (S, A, R, S') into the memory
    Sample a batch of transitions $\mathcal{B}$ uniformly from the memory
    for *each transition* (s, a, r, s') *in* $\mathcal{B}$ **do**
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big(r + \gamma \max_{a'} \tilde{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{q}(s, a, \mathbf{w})$
    end
    $S \leftarrow S'$
  end
end

Algorithm 12: Combined-Q with tabular function representation

Initialize the value function Q
Initialize the replay buffer
while *not converged* **do**
  Get the initial state S
  while S *is not the terminal state* **do**
    Select an action A according to a ϵ-greedy policy derived from Q
    Execute the action A, get the reward R and the next state S'
    Sample a batch of transitions $\mathcal{B}$ uniformly from the memory
    Store the online transition (S, A, R, S') into the memory
    Add the online transition (S, A, R, S') into $\mathcal{B}$
    for *each transition* (s, a, r, s') *in* $\mathcal{B}$ **do**
      $Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big)$
    end
    $S \leftarrow S'$
  end
end

Algorithm 13: Combined-Q with linear function representation

Input: a state-action feature function $\mathbf{x}$
Initialize the weights $\mathbf{w}$
Initialize the replay buffer
while *not converged* **do**
  Get the initial state S
  while S *is not the terminal state* **do**
    Select an action A according to a ϵ-greedy policy derived from $\mathbf{w}$
    Execute the action A, get the reward R and the next state S'
    Sample a batch of transitions $\mathcal{B}$ uniformly from the memory
    Store the online transition (S, A, R, S') into the memory
    Add the online transition (S, A, R, S') into $\mathcal{B}$
    for *each transition* (s, a, r, s') *in* $\mathcal{B}$ **do**
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big(r + \gamma \max_{a'} \mathbf{w}^{\mathrm{T}}\mathbf{x}(s',a') - \mathbf{w}^{\mathrm{T}}\mathbf{x}(s,a)\big)\mathbf{x}(s,a)$
    end
    $S \leftarrow S'$
  end
end

Algorithm 14: Buffer-Q with non-linear function representation

Initialize the weights $\mathbf{w}$
Initialize the replay buffer
while *not converged* **do**
  Get the initial state S
  while S *is not the terminal state* **do**
    Select an action A according to a ϵ-greedy policy derived from $\hat{q}$
    Execute the action A, get the reward R and the next state S'
    Sample a batch of transitions $\mathcal{B}$ uniformly from the memory
    Store the online transition (S, A, R, S') into the memory
    Add the online transition (S, A, R, S') into $\mathcal{B}$
    for *each transition* (s, a, r, s') *in* $\mathcal{B}$ **do**
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big(r + \gamma \max_{a'} \tilde{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{q}(s, a, \mathbf{w})$
    end
    $S \leftarrow S'$
  end
end

tasks.

Our first task was a 20×20 grid world. The agent is placed at the same location at the beginning of each episode (S in Figure 4.1(a)), and the location of the *goal* is fixed (G in Figure 4.1(a)). There are four possible actions {*Left*, *Right*, *Up*, *Down*}, and the reward is -1 at every time step, implying the agent should learn to reach the *goal* as soon as possible. Some fixed walls (black blocks in Figure 4.1(a)) are placed in the grid world, and if the agent bumps into the wall, the agent will remain in the same position.

Our second task was the Lunar Lander task from Box2D. The state space is $\mathbb{R}^8$ with value of each dimension unbounded. Lunar Lander has four discrete actions. Solving the Lunar Lander task needs careful exploration. Negative rewards are continually given during landing, so an algorithm can easily get trapped in a local minima, where the agent avoids negative rewards by doing nothing after certain steps until the episode ends.

The last task was the Atari game Pong. It is important to note that our evaluation is aimed to study experience replay, especially the memory size. We

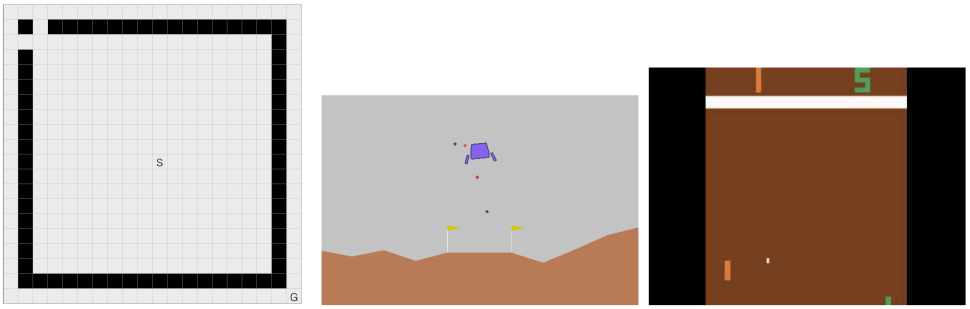


Figure 4.1: From left to right: the grid world, Lunar Lander, Pong

are not going to study how experience replay interacts with a deep convolutional network. To this end, it is better to use an accurate state representation of the game rather than learn the state representation during an end-to-end training. We, therefore, used the ram of the game as the state rather than the raw pixels. A state is then a vector in $\{0, \ldots, 255\}^{128}$. We normalized each element of this vector into $[0, 1]$ by dividing 255. The game Pong has six discrete actions.

To do experiments efficiently, we used the timeout mechanism during our evaluation. In other words, an episode will ends automatically after certain time steps. Timeout is necessary in practice. Otherwise, an episode can be arbitrarily long. However, we have to note that the timeout mechanism makes the environment non-stationary. To mitigate the negative influence of the timeout mechanism on our experimental results, we selected a timeout large enough for each task, so that an episode rarely ends due to timeout. We set the timeout to 5,000, 1,000 and 10,000 time steps for the grid world, Lunar Lander and the game Pong respectively. Furthermore, we used the partial-episode-bootstrap (PEB) technique introduced by Pardo, Tavakoli, Levdik, & Kormushev (2017). To be more specific, an episode may end with a transition $(S_t, A_t, R_{t+1}, S_{t+1})$ even if S_{t+1} is not a terminal state due to the timeout mechanism. In PEB, when we update the value function according to this transition, we still bootstrap from the value estimation of S_{t+1} computed by

the value function, rather than use 0 like some RL systems (e.g., Mnih et al., 2015). Pardo et al. (2017) showed PEB significantly mitigates the negative influence of the timeout mechanism.

Different mini-batch size has different computation complexity. Throughout our evaluation, we did not vary the batch size and used a mini-batch of fixed size 10 for all the tasks. In other words, we sampled 10 transitions from the memory at each time step. For Buffer-Q, we only sampled 9 transitions, and the mini-batch consisted of the sampled 9 transitions and the current transition. The behavior policy was always a ϵ-greedy policy with $\epsilon = 0.1$. We plot the online training progression for each experiment, in other words, we plot the episode return against the number of training episodes.

4.4 Experiment 1: Tabular Function Representation

In tabular methods, the value function Q is represented by a look-up table. Among the three tasks, only the grid world is compatible with tabular methods. We studied Algorithms 6, 9 and 12 with different memory size in this part. In our experiments, the initial values for all state-action pairs were set to 0, which is an optimistic initialization (Sutton, 1996) to encourage exploration. The discount factor was 1.0, and the step size was 0.1.

Figures 4.2 (a - c) show the training progression of different algorithms with different memory size for the grid world task. From Figure 4.2(a), the Online-Q agent solved the task in about $1,000$ episodes. From Figure 4.2(b), although all the Buffer-Q agents with different memory size achieved a reasonable performance level, it is interesting to see that the agent with the smallest memory performed the best in terms of both learning speed and final performance. When we increased the memory size from 10^2 to 10^5, the learning speed kept decreasing. When we kept increasing the memory size to 10^6, the

learning speed caught up but was still slower than the memory size 10^2. We did not keep increasing the memory size to a larger value than 10^6 as in all of our experiments the total training steps were less than 10^6. Things are different in Figure 4.2(c), where all the Combined-Q agents with different memory size learned the optimal solution at a similar speed. When we zoom in, we can find the agents with larger memory learned faster than the agents with smaller memory. This observation is contrary to what we observed with the Buffer-Q agents. From Figure 4.2(b), we can learn that in the original experience replay, a large memory hurt the performance. And from Figure 4.2(c), it is clear that CER made the agents less sensitive to the selection of the memory size.

Q-learning with a tabular function representation is guaranteed to converge under any data distribution only if each state-action pair is visited infinitely many times (together with some other weak conditions). However, the data distribution does influence the convergence rate. In the original experience replay, if a large memory is used, a transition is more likely to take effect later compared with a small memory. We use a simple example to show this. Assume we have a memory of size m, and we randomly sample 1 transition from the memory per time step. We assume the memory is full at the current time step and a new transition comes. We then remove the oldest transition in the memory and add this new transition into the memory. The probability that this newly added transition is replayed within next k ($k <= m$) time steps is

$$1 - (1 - \frac{1}{m})^k$$

This probability is monotonically decreasing as m increases. So with a larger memory, a transition is likely to take effect later. If that transition happens to be important (e.g., a transition to the goal state, a transition with a large reward or a transition to a well-learned state), this delay will further influence the data collection of the agent in the future. As a result, the overall learning

speed slows down. This explains the phenomenon shown in Figure 4.2(b) that when we increased the memory size from 10^2 to 10^5 the learning slowed down. Note with the memory size 10^6, the memory never got full, and all transitions were well preserved. It was a special case. In CER, all the transitions take effect immediately. As a result, the agents became less sensitive to the selection of the memory size.

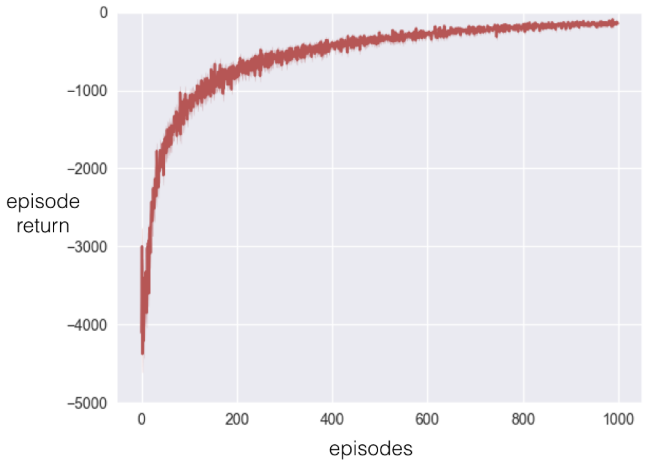
4.5 Experiment 2: Linear Function Approximation

We studied Algorithms 7, 10 and 13 with different memory size in this part. We used linear function approximation with tile coding. Among our three tasks, only the grid world is compatible with tile coding. In our experiments, tile coding was done via the tile coding software [1] with 8 tilings. We set the initial weight parameters to 0 to encourage exploration. The discount factor was 1.0, and the step size was $0.1/8 = 0.125$. The results are summarized in Figure 4.3. Figure 4.3(b) shows that a large memory hurt the learning speed of the Buffer-Q agents. Comparing Figure 4.3(b) and Figure 4.3(c), it is clear that CER sped up learning significantly, especially for agents with a large memory. The results are similar to what we observed in agents with tabular function representation.

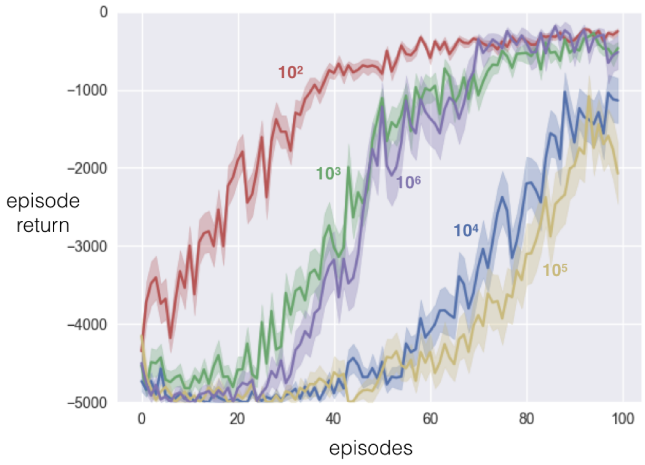
4.6 Experiment 3: Nonlinear Function Approximation

We studied Algorithms 8, 11 and 14 with different memory size in this part. We used a single hidden layer network as our nonlinear function approximator. We used the $ReLU$ nonlinearity over the hidden units, and the output units were linear to produce the state-action value. With a neural network as function

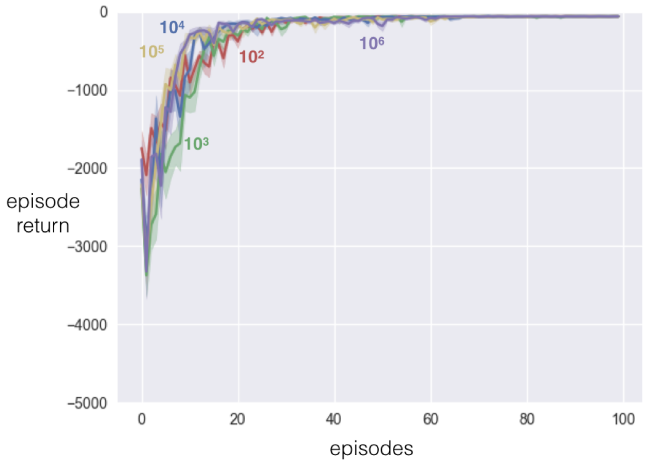
[1] http://incompleteideas.net/sutton/tiles/tiles3.html



(a) Online-Q (Algorithm 6)

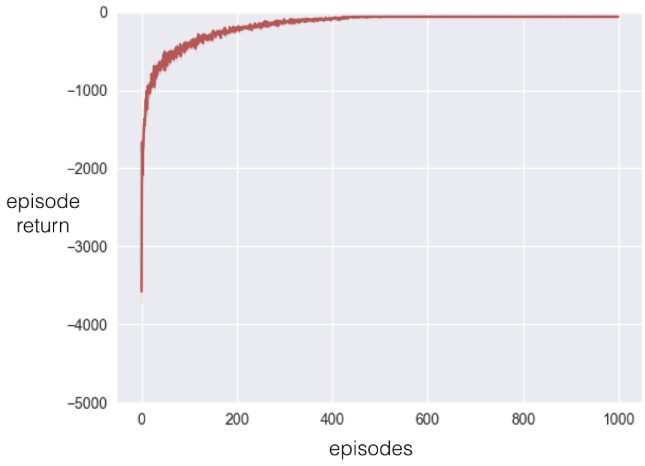


(b) Buffer-Q (Algorithm 9)

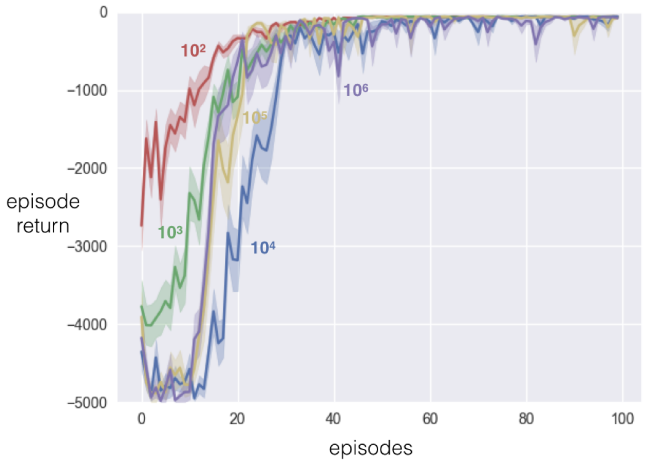


(c) Combined-Q (Algorithm 12)

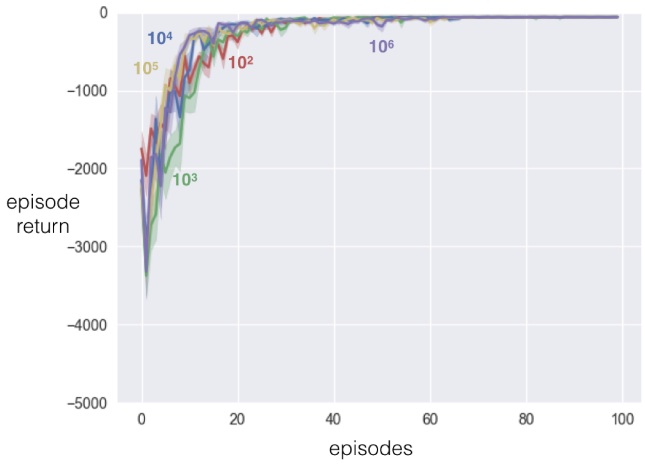
Figure 4.2: Learning curves of agents with tabular function representation in the grid world. Numbers indicate the memory size, and all other hyperparameters except the memory size were the same. The results are averaged over 30 independent runs, and standard errors are plotted as the shadow.



(a) Online-Q (Algorithm 7)



(b) Buffer-Q (Algorithm 10)



(c) Combined-Q (Algorithm 13)

Figure 4.3: Learning curves of agents with linear function representation in the grid world. Numbers indicate the memory size, and all other hyperparameters except the memory size were the same. The results are averaged over 30 independent runs, and standard errors are plotted as the shadow.

approximator, Buffer-Q and Combined-Q are similar to DQN, and we used a target network to allow stable update targets following DQN. Our preliminary experiments showed that some other techniques used in DQN (e.g., random exploration at the beginning stage and a linearly decayed exploration rate) did not increase the performance in our tasks.

In the grid world task, we used 50 hidden units for the hidden layer, and for the other tasks, we used 100 hidden units. In the grid world task, we used a one-hot vector of length $20 \times 20 = 400$ to encode the current position of the agent. We used an RMSProp optimizer (Tieleman & Hinton, 2012) for all the tasks, while the initial step size varied from task to task. To be more specific, we used 0.01, 0.0005 and 0.0025 as the initial step size for the grid world, Lunar Lander and the game Pong respectively. All the aforementioned hyper-parameters were empirically tuned to achieve good performance.

Figure 4.4 shows the learning progression of the agents with various memory size in the grid world task. We observed that the online Q agent, the Buffer-Q agent with a memory size 100 and the Combined-Q agent with a memory size 100 failed to reach a reasonable performance level. This phenomenon is expected, as in those cases the network tended to over-fit recent transitions thus forgot what it had learned from previous transitions. From Figure 4.4(a), the Buffer-Q agent with a memory size 10^4 learned fastest among all the Buffer-Q agents. This is a medium memory size rather than the smallest memory size as we observed in Buffer-Q agents with tabular and linear function representation. We hypothesize that there is a trade-off in the nonlinear function approximation case. With a small memory, important transitions are likely to take effect early. However, sampled transitions from the memory tend to be highly temporally correlated, while training a neural network often needs i.i.d. data. With a large memory, sampled transitions tend to be more uncorrelated. However, important transitions are usually delayed to take effect.

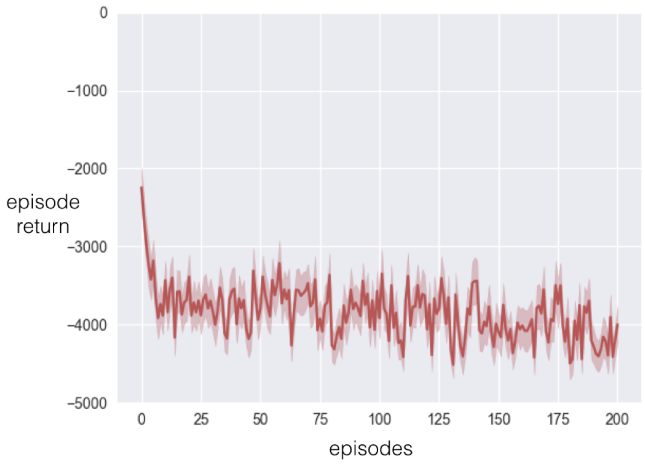
In Figure 4.4(b), the Buffer-Q agents with a huge memory (e.g., 10^5 and 10^6) failed to find the optimal solution. However, in Figure 4.4(c), corresponding Combined-Q agents did find a reasonable solution. And comparing Figure 4.4 (b) and (c), it is clear that CER significantly sped up learning, especially for agents with a large memory.

Figure 4.5 shows the learning progression of the agents with various memory size in Lunar Lander. Different from the grid world task, the Online-Q agent, the Buffer-Q agent with a memory size 100 and the Combined-Q agent with a memory size 100 achieved a reasonable performance level. The Online-Q agent achieved almost the best performance among all the agents. This suggests that in this task the neural network function approximator may be less likely to over-fit recent transitions compared with the grid world. From Figure 4.5(b), the Buffer-Q agent with a medium memory (10^3) achieved better performance than all the other Buffer-Q agents. With a large memory (10^5 or 10^6), the Buffer-Q agent failed to solve the task. Comparing Figure 4.5 (b) and (c), we can see that CER improved the performance of the agents with a large memory (e.g., $10^3, 10^5$, and 10^6). One interesting observation is that some Buffer-Q agents and Combined-Q agents had a performance drop after certain time steps. We found this drop occurred even with a decreased initial step size.

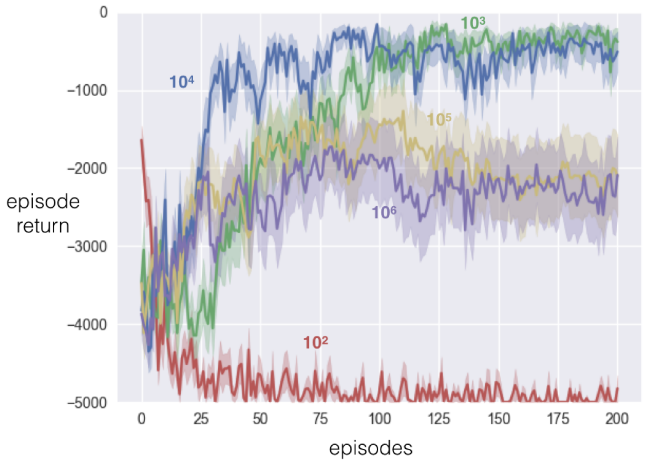
Figure 4.6 shows the learning progression of the agents with various memory size in the game Pong. We observed similar phenomena as the grid world task. However, in this task, CER did not provide a performance improvement.

4.7 There Is No Universal Rule to Set the Memory Size

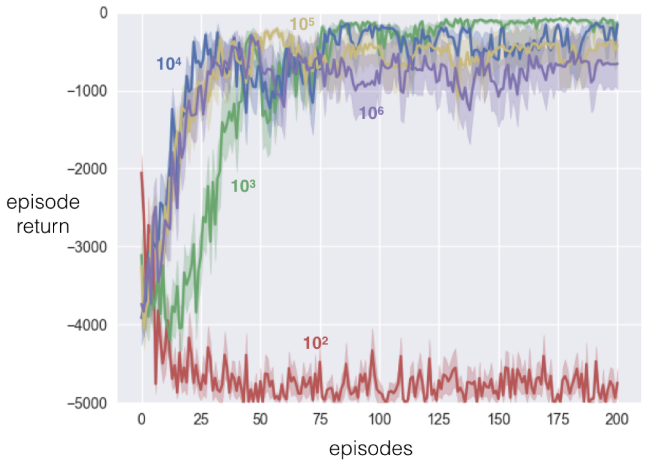
Although experience replay does help stabilize the training of a neural network function approximator and until now we do not have a better approach



(a) Online-Q (Algorithm 8)

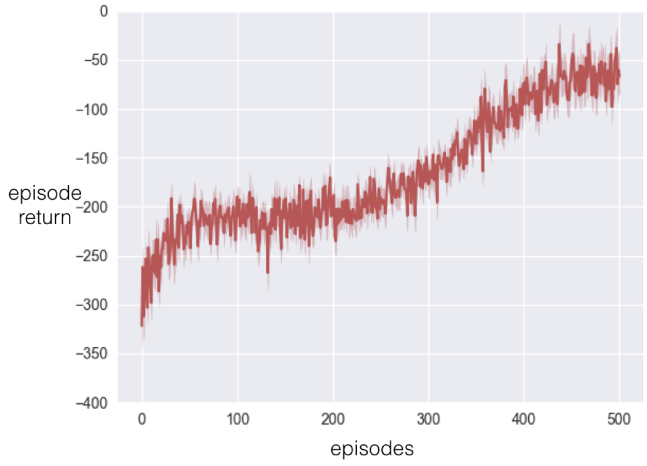


(b) Buffer-Q (Algorithm 11)

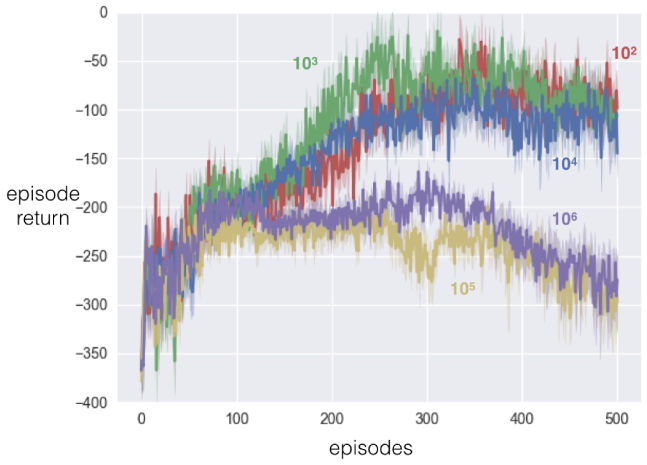


(c) Combined-Q (Algorithm 14)

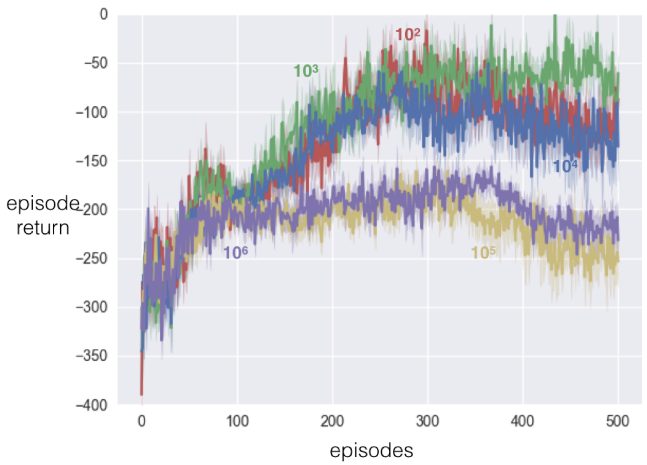
Figure 4.4: Learning curves of agents with nonlinear function representation in the grid world. Numbers indicate the memory size, and all other hyperparameters except the memory size were the same. The results are averaged over 30 independent runs, and standard errors are plotted as the shadow.



(a) Online-Q (Algorithm 8)

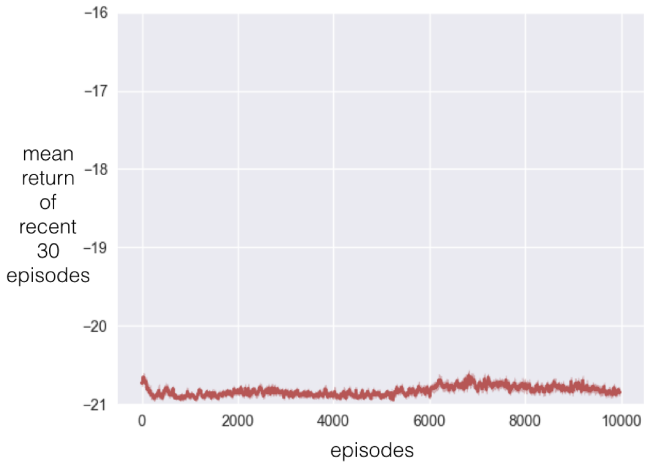


(b) Buffer-Q (Algorithm 11)

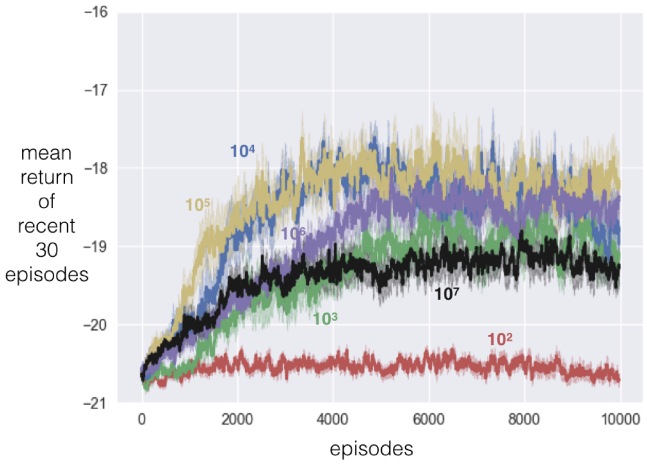


(c) Combined-Q (Algorithm 14)

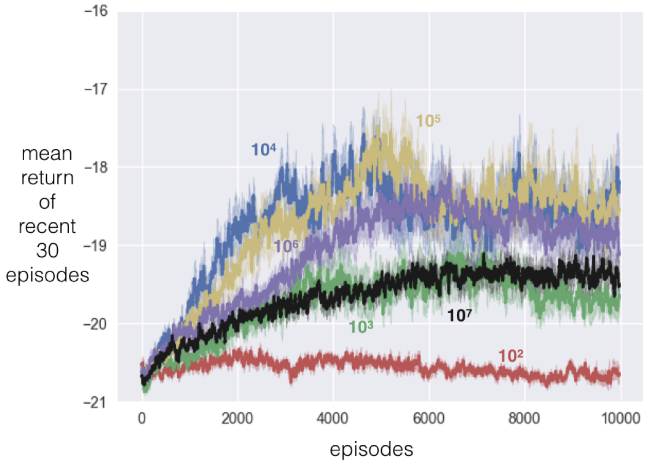
Figure 4.5: Learning curves of agents with nonlinear function representation in Lunar Lander. Numbers indicate the memory size, and all other hyperparameters except the memory size were the same. The results are averaged over 30 independent runs, and standard errors are plotted as the shadow.



(a) Online-Q (Algorithm 8)



(b) Buffer-Q (Algorithm 11)



(c) Combined-Q (Algorithm 14)

Figure 4.6: Learning curves of agents with nonlinear function representation in Pong. Numbers indicate the memory size, and all other hyper-parameters except the memory size were the same. The results are averaged over 10 independent runs, and standard errors are plotted as the shadow. It is expected that the agent did not solve the game Pong, as it is difficult to approximate the state-value function with a single-hidden-layer network.

than experience replay, we have to note that experience replay introduces a new hyper-parameter, the memory size. It is time to rethink the utility of experience replay. There is no universal rule to set the memory size. First, the memory size is task-dependent. The 'default' value, 10^6, never worked best in our experiments. Second, the memory size is not orthogonal with function representation. With tabular and linear function representation, a small memory worked best. With non-linear function representation, a medium memory worked best.

We proposed CER that made the learning system less sensitive to the selection of the memory size compared with the original experience replay. However, CER is only a workaround. We should focus on developing more principled solutions to the instability issue of the online training for a neural network function approximator.

Chapter 5

Conclusions and Extensions

This chapter serves as an end of this thesis. We briefly discuss our contributions, followed by possible directions for future research.

5.1 Cross-propagation Is A Promising Technique

In this thesis, we proposed the cross-propagation technique that does cross-validation online to update parameters of an algorithm. The key idea is to use the newly coming training data as a hold-out validation set and treat the loss on this training data as a generalization loss. We designed three cross-propagation-based algorithms to train a single hidden layer network. We demonstrated the merits of the three cross-propagation-based algorithms in both online and off-line learning. In online learning setting, the cross-propagation-based algorithms learned features in a more stable way than back-propagation. In off-line learning setting, the cross-propagation-based algorithms performed better than back-propagation when training data was not sufficient compared with the network complexity, where over-fitting was likely to happen.

5.2 The Memory Size Is A New Trouble

In this thesis, we presented a systematic study of experience replay, especially the memory size. We showed that the memory size is a task-dependent hyper-parameter and is not orthogonal with function representation. There is no universal rule to set the memory size. The importance of the memory size has been under-estimated by the community for a long time, and experience replay is widely used as a standard technique to obtain temporally uncorrelated transitions. We proposed the Combined Experience Replay (CER). CER made the learning system more robust to the selection of the memory size compared with the original experience replay.

5.3 More Combinations of Cross-propagation and Neural Networks

In this thesis, we applied the cross-propagation technique to train the feature layer of a single hidden layer network. Future work may involve a deeper combination of cross-propagation and neural networks, for example, updating all the layers of a neural network with cross-propagation, applying cross-propagation to convolutional layers, combining recurrent neural networks with cross-propagation. We expect that a deeper combination of cross-propagation and neural networks will help address the over-fitting issue of neural networks. However, those extensions are non-trivial. The most challenging part is to reduce the required memory for cross-propagation algorithms. As discussed in Section 3.4, the memory requirement under a naive extension will increase exponentially with the increase of the depth of a network. In this thesis, we proposed two cross-propagation-based algorithms that apply cross-propagation in the feature level directly, resulting in a reduced memory requirement. However, new techniques are necessary to reduce the memory requirement further

and apply cross-propagation to deeper architectures.

5.4 Cross-propagation in Reinforcement Learning

Cross-validation is an effective method to avoid over-fitting and is widely used in off-line supervised learning, where the dataset is predefined. However, reinforcement learning systems hardly benefit from cross-validation. The main reason is that in reinforcement learning problems, we do not have a predefined dataset. The agent interacts with the environment in an *online* manner and adjusts its policy according to the reward signal.

Over-fitting is likely to happen in a reinforcement learning system when a neural network is used as a non-linear function approximator. Because with a neural network function approximator, an update to current state value estimation may influence the value estimation of other states too much. This issue is also known as over-generalization. When a neural network function approximator is only trained on a small portion of the whole state space, due to the over-generalization issue, the network may focus too much on this small portion of states, and the learned value estimation for other states gets interfered. When the neural network experiences other states again, it cannot produce an accurate prediction. This phenomenon usually happens in the online training of a neural network function approximator, where states for updating the network mainly come from recent transitions. Due to the temporal correlation of the recent transitions, those states are likely to cover only a small portion of the whole state space.

Experience replay is widely used to provide temporally uncorrelated transitions to update a neural network function approximator. In this thesis, we show experience replay introduces a new task-dependent hyper-parameter, the memory size, and there is no universal rule to set this new hyper-parameter.

Cross-propagation does cross-validation online. This property is a perfect match with a reinforcement learning system. Future work could involve combining cross-propagation technique with reinforcement learning systems, e.g., using cross-propagation-based algorithms to update a neural network function approximator. This combination is promising in addressing the aforementioned over-generalization issue. And we expect with this combination, a reinforcement learning system will no longer need experience replay.

References

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., ... Zaremba, W. (2017). Hindsight experience replay. In *Advances in Neural Information Processing Systems 30*, pp. 5048–5058.

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. In *Proceedings of the Third International Conference on Representations Learning*.

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research, 47*:253–279.

Bellman, R. (2013). Dynamic programming. Courier Corporation.

Bengio, Y. (2000) Gradient-based optimization of hyperparameters. *Neural Computation, 12*(8):1889–1900.

Do, C. B., Foo, C. S., & Ng, A. Y. (2008). Efficient multiple hyperparameter learning for log-linear models. In *Advances in Neural Information Processing Systems 19*, pp. 377–384.

Domke, J. (2012) Generic methods for optimization-based modeling. In *Proceedings of the 2012 International Conference on Artificial Intelligence and Statistics*, pp. 318–326.

Franceschi, L., Donini, M., Frasconi, P., & Pontil, M. (2017). Forward and reverse gradient-based hyperparameter optimization. In *Proceedings of the Thirty-Fourth International Conference on Machine Learning*, pp. 1165–1173.

Franceschi, L., Frasconi, P., Salzo, S., & Pontil, M. (2018). Bilevel programming for hyperparameter optimization and meta-learning. In *Proceedings of the Thirty-Fifth International Conference on Machine Learning*, pp. 1563–1572.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... Bengio, Y. (2014). Generative adversarial nets. In *Advances in Neural Information Processing Systems 27*, pp. 2672–2680.

He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask R-Cnn. In *Proceedings of the 2017 IEEE International Conference of Compute Vision*, pp. 2980–2988.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*.

Hinton, G. E., McClelland, J. L., & Rumelhart, D. E. (1986). Distributed representations. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, 1*(3), 77–109.

Hoerl, A. E., & Kennard, R. W. (1970). Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics, 12*(1), 55–67.

Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. *Neural Networks, 1*(4), 295–307.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. In *Proceedings of the Third International Conference on Representation Learning*.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., ... Hassabis, D. (2017). Overcoming catastrophic forgetting in neural networks. In *Proceedings of the National Academy of Sciences, 114*(13), 3521–3526.

Klopf, A., & Gose, E. (1969). An evolutionary pattern recognition network. *IEEE Transactions on Systems Science and Cybernetics, 5*(3), 247–250.

Konidaris, G., Osentoski, S., & Thomas, P. S. (2011). Value function approximation in reinforcement learning using the Fourier basis. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pp. 1097–1105.

Levine, S., Finn, C., Darrell, T., & Abbeel, P. (2016). End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research, 17*(1), pp. 1334–1373.

Li, Z., & Hoiem, D. (2017). Learning without forgetting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. doi:10.1109/TPAMI.2017.2773081

Liang, Y., Machado, M. C., Talvitie, E., & Bowling, M. (2016). State of the art control of Atari games using shallow reinforcement learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pp. 485–493.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... Wierstra, D. (2015). Continuous control with deep reinforcement learning. In *Proceedings of the Fourth International Conference on Representation Learning*.

Lin, L. J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning, 8*(3-4), pp. 293–321.

Liu, R., & Zou, J. (2017). The effects of memory replay in reinforcement learning. ArXiv:1710.06574.

Luketina, J., Berglund, M., Greff, K., & Raiko, T. (2016). Scalable gradient-based tuning of continuous regularization hyperparameters. In *Proceedings of the Thirty-Third International Conference on Machine Learning*, pp. 2952–2960.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics, 5*(4), 115–133.

Maclaurin, D., Duvenaud, D. K., & Adams, R. P. (2015). Gradient-based hyperparameter optimization through reversible learning. In *Proceedings of the Thirty-Second International Conference on Machine Learning*, pp. 2113–2122.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *Proceedings of the Thirty-Third International Conference on Machine Learning*, pp. 1928–1937.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Petersen, S. (2015). Human-level control through deep reinforcement learning. *Nature, 518*(7540), 529.

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the Twenty-Seventh International Conference on Machine Learning*, pp. 807–814.

Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., Celik, Z. B., & Swami, A. (2016). The limitations of deep learning in adversarial settings. In *Proceedings of the 2016 IEEE European Symposium on Security and Privacy*, pp. 372–387.

Pardo, F., Tavakoli, A., Levdik, V., & Kormushev, P. (2018). Time limits in reinforcement learning. In *Proceedings of the Thirty-Fifth International Conference on Machine Learning*, pp. 4042–4051.

Pedregosa, F. (2016). Hyperparameter optimization with approximate gradient. In *Proceedings of the Thirty-Third International Conference on Machine Learning*, pp. 737–746.

Precup, D., Sutton, R. S., & Singh, S. P. (2000). Eligibility traces for off-policy policy evaluation. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pp. 759–766.

Rigamonti, R., Sironi, A., Lepetit, V., & Fua, P. (2013). Learning separable filters. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2754–2761.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985). Learning internal representations by error propagation. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, 1*(318), 318–362

Rummery, G. A., & Niranjan, M. (1994). On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166. University of Cambridge, Department of Engineering.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. In *Proceedings of the Third International Conference on*

Learning Representations.

Schraudolph, N. N. (1999). Local gain adaptation in stochastic gradient descent. In *Proceedings of the Ninth International Conference on Artificial Neural Networks*, pp. 569–574.

Shin, H., Lee, J. K., Kim, J., & Kim, J. (2017). Continual learning with deep generative replay. In *Advances in Neural Information Processing Systems 30*, pp. 2994–3003.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... Dieleman, S. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature, 529*(7587), 484–489.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research, 15*(1), 1929–1958.

Stone, P., Sutton, R. S., & Kuhlmann, G. (2005). Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior, 13*(3), 165–188.

Sutton, R. S. (1986). Two problems with backpropagation and other steepest-descent learning procedures for networks. In *Proceedings of Eightth Annual Conference of the Cognitive Science Society*.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine learning, 3*(1), 9–44.

Sutton, R. S. (1992a). Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *Proceedings of the Tenth AAAI Conference on Artificial Intelligence*, pp. 171–176.

Sutton, R. S. (1992b). Gain adaptation beats least squares. In *Proceedings of the Seventh Yale Workshop on Adaptive and Learning Systems*, pp. 161–166.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 9*, pp. 1038–1044.

Sutton, R. S. (2014). Myths of representation learning. Lecture in the Second International Conference on Representation Learning.

Sutton, R. S., & Barto, A. G. (1998). Reinforcement learning: An introduction. Cambridge: MIT press.

Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction (2nd edition). Cambridge: MIT press.

Sutton, R. S., & Whitehead, S. D. (1993). Online learning with random representations. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 314–321.

Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017). Inception-v4,

inception-resnet and the impact of residual connections on learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*.

Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Casas, D. D. L., ... Lillicrap, T. (2018). DeepMind control suite. ArXiv:1801.00690.

Tibshirani, R. (1996). Regression shrinkage and selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 267–288.

Tieleman, T., & Hinton, G. (2017). RMSProp: Divide the gradient by a running average of its recent magnitude. Lecture 6.5 in Neural Networks for Machine Learning, Coursera.

Todorov, E., Erez, T., & Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *Proceedings of the 2012 IEEE International Conference on Intelligent Robots and Systems*, pp. 5026–5033.

Van Seijen, H., Van Hasselt, H., Whiteson, S., & Wiering, M. (2009). A theoretical and empirical analysis of expected Sarsa. In *Proceedings of the 2009 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pp. 177–184.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems 30*, pp. 6000–6010.

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., & de Freitas, N. (2016). Sample efficient actor-critic with experience replay. In *Proceedings of the Fifth International Conference on Representation Learning*.

Watkins, C. J. C. H. (1989). Learning from delayed rewards. Doctoral dissertation, King's College, Cambridge.

Williams, R. J., & Zipser, D. (1989). Experimental analysis of the real-time recurrent learning algorithm. *Connection Science, 1*(1), 87–111.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., ... Klingner, J. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. ArXiv:1609.08144.

LeCun, Y. (1998). The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist