MUTUAL REINFORCEMENT LEARNING

by

Cameron Reid

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science

Department of Computer and Information Science

Indianapolis, Indiana

May 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

This thesis is dedicated to my mother, who threatened violence against me if I didn't go to college (but in a fun way!) and instilled in me the stubbornness which made this possible, and to Alexa, who helped me believe that I could and should put in the work, with special thanks to Professor Mukhopadhyay, who has been inexplicably generous with his time and support before and after we began to work together.

ACKNOWLEDGMENTS

The research reported here was supported by the National Science Foundation under grant number 1930606.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Mutual learning is an emerging field in intelligent systems which takes inspiration from naturally intelligent agents and attempts to explore how agents can communicate and cooperate to share information and learn more quickly. While agents in many biological systems have little trouble learning from one another, it is not immediately obvious how artificial agents would achieve similar learning. In this thesis, I explore how agents learn to interact with complex systems. I further explore how these complex learning agents may be able to transfer knowledge to one another to improve their learning performance when they are learning together and have the power of communication. While significant research has been done to explore the problem of knowledge transfer, the existing literature is concerned either with supervised learning tasks or relatively simple discrete reinforcement learning. The work presented here is, to my knowledge, the first which admits continuous state spaces and deep reinforcement learning techniques. The first contribution of this thesis, presented in Chapter 2, is a modified version of deep Q-learning which demonstrates improved learning performance due to the addition of a mutual learning term which penalizes disagreement between mutually learning agents. The second contribution, in Chapter 3, is a presentation work which describes effective communication of agents which use fundamentally different knowledge representations and systems of learning (model-free deep Q learning and model-based adaptive dynamic programming), and I discuss how the agents can mathematically negotiate their trust in one another to achieve superior learning performance. I conclude with a discussion of the promise shown by this area of research and a discussion of problems which I believe are exciting directions for future research.

1. BACKGROUND

Reinforcement learning is an area of machine learning which involves an agent learning a policy $\boldsymbol{\pi}$ to operate within an environment in order to maximize a long-term discounted reward. There are many algorithms which attempt to solve this problem [1] which allow various assumptions about the learning agent and the environment. For example, the learning agent may require a continuous or discrete action space, and the environment may provide continuous or discrete observation values (or some of each). Furthermore, the agent may be assumed to have an omniscient view of the environment, or the agent may only receive a subset of the information in the environment; i.e., the environment may be *fully observed* or *partially observed*. All of these factors influence the choice of reinforcement learning algorithm, and some of these factors rule out some options completely.

While there exists a large variety of learning models and algorithms [1], [2] for reinforcement learning depending on the assumptions about the learner or the environment, there are two broad approaches to solving this problem. Model-based reinforcement learning attempts to learn models of the dynamics of the system F and use that information to derive an optimal policy, while model-free (also known as direct) methods attempt to learn about the reward function or the policy without the use of a model.

1.1 Model-free Learning

Model-free methods often rely on estimation of the state value function $V_{\boldsymbol{\pi}}^* : S \to \mathcal{R}$ of a policy $\boldsymbol{\pi}$, where $V_{\boldsymbol{\pi}}^*$ maps a state to the discounted reward attainable starting in that state and following the policy $\boldsymbol{\pi}$. In order to achieve this, we leverage the Bellman optimality equation (Eq. 1.1) where A_s is the set of actions that can be taken in state s, $R(s, a)$ is a function that maps a state/action pair to a reward in $\mathcal{R}$, $T(s, a)$ is a transition function which maps state/action pairs to the next state, and γ is a value close to 1 called the "discount rate" which ensures that the value function is bounded.

$$V_{\boldsymbol{\pi}}^*(s) = R(s, \boldsymbol{\pi}(s)) + \gamma V_{\boldsymbol{\pi}}^*(T(s, \boldsymbol{\pi}(s))) \tag{1.1}$$

We can use this function to optimize an approximation to V^*_π, say $\hat{v}$, by minimizing the temporal difference error:

$$\hat{v}(s) - (r + \gamma \hat{v}(s')) \tag{1.2}$$

Note that these equations are specific to the policy being followed. This kind of learning is referred to as *on-policy* learning, because the learning process is dependent upon the policy being followed, and only transitions from that policy are relevant to the learning process. In other words, learning must involve iteratively collecting a number of transitions using the current policy, optimizing the estimate of the value function based on those transitions, and then discarding them. As discussed below in Section 1.3, this has important consequences for algorithms which might try to use deep learning to approximate the value function.

Another class of algorithms, called *off-policy* algorithms, are able to learn about the problem using transitions generated from any policy. *Q-learning* [3] is a prototypical example of such an algorithm. As the name implies, Q-learning attempts to learn a *Q-function* $Q(s, a)$ which maps a given state/action combination to its long-term discounted reward. The algorithm works by minimizing the loss in 1.3, where a_t is the action taken at time t, r_t is the reward obtained transitioning from state s to s' at time t, $a \in A$ are the set of actions, and γ is the same discount factor mentioned above, which controls how "forward-looking" the algorithm will be, and is generally necessary to ensure that the Q function is bounded.

$$\mathcal{L} = Q(s_t, a_t) - (r_t + \gamma \text{max}_A(Q(s', a_{t+1}))) \tag{1.3}$$

Now note that these value-based learning methods do not actually tell us what the policy for an agent should be. An obvious solution would be to simply choose the action that maximizes the value of the resulting state, say $\max_{a \in A} Q(s, a)$. This policy would be called the greedy policy with respect to Q. However, it is often important to balance *exploration* (taking actions which may be sub-optimal in order to collect transitions from previously-unseen areas of the problem) and *exploitation* (using the knowledge gained to achieve optimal control). This balance is typically found by using what's referred to as an

ϵ-greedy policy: given some value ϵ, take the greedy action with probability $1-\epsilon$, or choose a random action with probability ϵ. Then, ϵ is typically decreased to 0 as training progresses.

1.2 Model-based Learning

In contrast to model-free methods, model-based methods attempt to learn the dynamics of the environment (i.e., the state transition and reward functions) and develop an optimal policy from there.

One such method is known as Adaptive Dynamic Programming, or ADP [4]. This algorithm assumes a discrete state and action space and attempts to develop a model of the transition and reward functions, and then use dynamic programming to generate an estimate of the Q value of each action in each possible discrete state. In other words, we may count each encountered $\langle \mathbf{s}, \mathbf{a}, \mathbf{r}, \mathbf{s}' \rangle$ tuple in a set $\mathcal{X}$. Transition probabilities from state s to state s' given action a are then computed as the fraction of times s' was seen after taking action a in state s More formally the probability of seeing state s' after taking action a in state s is given in Eq. 1.4

$$P(s'|s,a) = \frac{|\mathcal{X}_{\mathbf{s}=s,\mathbf{a}=a,\mathbf{s}'=s'}|}{|\mathcal{X}_{\mathbf{s}=s,\mathbf{a}=a}|} \tag{1.4}$$

where $|*|$ denotes cardinality and subscripts denote which elements of the set are under consideration.

Then, given transition probabilities P as computed above, we use dynamic programming [5] to find the optimal long-term value of each state by recursively applying Eq. 1.5, where the hat operator indicates an estimate of the true function. Because the agent develops a model of the problem dynamics, optimal control can be achieved by selecting the action which is expected to lead to the state with the highest long-term value.

$$V(s) = \max_{a \in A} \left(\hat{R}(s,a) + \gamma V(\hat{T}(s,a)) \right) \tag{1.5}$$

This algorithm is able to derive a reasonable estimate of the Q function in significantly fewer steps than model-free methods; however, the assumption of a discrete state space makes the algorithm intractable for problems with higher-dimensional state spaces, and the dynamic programming algorithm involves significantly more computation than e.g. Q-learning. This is a common finding in the literature – model based methods learn a problem effectively in a small number of steps, but require significantly more computation.

1.3 Deep Reinforcement Learning

Deep reinforcement learning refers to a branch of reinforcement learning which uses deep learning techniques to estimate the value function, the Q function, or the environment dynamics. While deep learning had always shown powerful potential to represent complex domains, the nature of reinforcement learning introduced some major difficulties. The combination of function approximation, bootstrapping, and off-policy learning were described in [1] as "the deadly triad" because they can lead to an algorithm which diverges quickly. Furthermore, deep learning via gradient descent methods generally assumes that training samples will be independent and identically distributed, and reinforcement learning algorithms (which learn on experience tuples generated by the environment in order) violate this assumption.

The seminal paper in deep reinforcement learning is presented in [6]. In this work, the authors present a Q-learning algorithm which uses a convolutional neural network to achieve superhuman performance in several games in the Atari domain. Q-learning with a function approximator (such as a neural network) can easily become unstable due to the combination of off-policy learning, function approximation, and *bootstrapping* (that is, using the value of a state to estimate the value of a previous state). These three factors comprise the so-called "deadly triad" in reinforcement learning, and the three are all present in Q-learning with a function approximator.

To deal with these issues, the authors of [6] discuss several techniques.

1. *Experience replay* [7], in which a buffer of previous experience tuples is stored and then drawn from at random for minibatch updates. This is important; one reason that the deadly triad is so harmful is that sequential states are significantly correlated. In a function approximation setting, updating the value for a particular state also updates the value for all

states, especially those similar to the state being updated. This can lead to rapid divergence in state value estimates and unstable training. However, drawing experience tuples from a replay buffer decorrelates updates and can help stabilize training.

2. The authors also used a *target network*, which is a clone of the Q network which is used to generate targets for the Q-learning update and whose parameters are periodically overwritten by the current parameters of the Q network. Similar to the above, this helps stabilize training by limiting the impact of bootstrapping, as updates to the Q network are not included in the update target until the parameters of the target network are updated.

3. Finally, the authors also indicate that clamping the error value to the range (-1, 1) also stabilizes training.

All of the above have become standard practice in the literature for deep reinforcement learning. In [8], the authors explore why these techniques are important specifically in the context of deep Q learning, and note that *overestimation bias* – that is, the unavoidable tendency of Q learning to overestimate the value of the discounted future state, induced by the max operator in 1.3 – is a significant factor which is ameliorated by computing updates from a different network (such as a target network).

In addition to Q learning, a number of deep *policy gradient* algorithms have been proposed which attempt to learn a policy directly. A typical algorithm in this area, known as *deep deterministic policy gradient* or *DDPG*, was presented in [9]. This is what is known as an *actor-critic* algorithm, because it involves two functions – an actor $\boldsymbol{\pi} : S \to A$ which maps a state to an action, and a critic $Q : S \times A \to \mathcal{R}$, which (as usual) maps a state-action pair to its associated long-term value. Learning a policy directly is very important in problems which involve continuous components in the action space because choosing the optimal action within a continuous state is intractable for Q-learning.. Because of this limitation, much of the state-of-the-art in reinforcement learning for practical, real-world systems and obotics involves variants of these deep reinforcement learning algorithms [10]–[12].

1.4 Transfer Reinforcement Learning

Transfer learning [13] describes to the problem of using some knowledge that was learned by one agent on one problem to achieve faster learning on a new problem, faster learning for a new agent on the same problem, or faster learning for a new agent on a new problem. These techniques are motivated by the ability of natural agents to apply previously learned knowledge to a new problem, or to observe the behavior of other agents to solve new problems, and many techniques are inspired by that.

1.4.1 Student/Teacher Advising

Student/Teacher advising in a reinforcement learning context, refers to an interaction of learning agents wherein one agent (the teacher) has learned about the environment and communicates with the other agent (the student) to guide the student towards taking the best action and learning about the environment more quickly. The general framework is established in [14]. In student/teacher advising, we make no assumptions about the internal knowledge representation of the learning agents. In general, however, we must assume that the teacher has some level of expertise with regards to the problem being solved. Because communication between agents may be expensive or difficult, a student/teacher advising approach usually considers a fixed *budget* of communication. That is, the teacher may only guide the student at a fixed number of steps, and choosing which steps to intervene in becomes an important problem. Reference [14] proposes a few strategies, given a budget of N advice steps: *early advising*, wherein the teacher provides guidance in the first N learning steps; *importance advising*, wherein some importance function (for example, the one shown in Eq. 1.6 below) is included, and advice is only provided if the importance exceeds some threshold.

$$I(s) = \max_a Q(s,a) - \min_a Q(s,a) \tag{1.6}$$

The authors also explore more advanced advising techniques, such as *mistake correcting* (the teacher overrides actions that it deems "bad"), or *predictive advising* (the teacher also attempts to learn a model of the student's policy and preemptively correct bad actions by the student.

This general approach was expanded upon in [15] to relax the constraint that the teacher must have strong knowledge of the problem. The authors propose a number of methods which allow the agents to evaluate how confident they are about their knowledge of a given state, and ask for or provide advice accordingly. Key to this is the idea of *confidence* – how accurate the agent "believes" its knowledge of a state to be. The authors propose formalizing this idea of confidence in two ways: "should I ask for advice about this state?" and "should I provide advice for this state", denoted P_{ask} and P_{give} in Eqs. 1.7 and 1.8 below, where Υ and Ψ are functions from the state space to the unit interval and indicate the "asking" and "giving" confidence of the agent in that state, and v_a and v_g are tunable scaling parameters.

$$P_{ask}(s, \Upsilon) = (1 + v_a)^{-\Upsilon(s)} \tag{1.7}$$

$$P_{ask}(s, \Psi) = 1 - (1 + v_g)^{-\Psi(s)} \tag{1.8}$$

In their analysis, the authors propose possible functions for Ψ and Υ which cause the confidence of the agent to grow with the number of visits it has made to the state. These are shown in Eqs. 1.9 and 1.10 below, where $n(s)$ is the number of times the agent has visited state s

$$\Upsilon(s) = \sqrt{n(s)} \tag{1.9}$$

$$\Psi(s) = \log_2((n(s)) \tag{1.10}$$

With these, the authors were able to achieve improved performance on a fairly complex task. However, this method relies on a discrete state space, which is often an assumption that cannot be met in real-world problems.

1.4.2 Model Distillation

Related to the problem of transfer learning is the problem of *model distillation*, where we wish to take a large, powerful network (or an ensemble of networks) and compress it into a smaller network with the same level of performance to allow it to run on a cheaper or otherwise less-powerful hardware. A survey of the current state-of-the-art can be found [16]. This is a broad area that admits many approaches.

One common approach, initially proposed several years ago by [17], is to train a small neural network to mimic the output of large, complex model (potentially an ensemble of hundreds or thousands of models). The authors present results which show little to no degradation of performance between the complex model and the distilled model, despite distilled models which are several orders of magnitude smaller. The key intuition to this approach is that the limiting factor for many neural networks lies in the optimization process rather than the representational power of the network itself, and training based on the knowledge of a "teacher" network helps to avoid those issues. Reference [18] expand and formalize on this process and explore new ways in which the distilled network can be trained against the teacher network.

1.5 Mutual Learning

Mutual learning attempts to answer the question: "How do agents which are learning the same, or similar, problems communicate to improve learning?" Initial quantitative definition of this problem was presented in [19], which notes that mutual learning can happen between humans and machines, humans and humans, or machines and machines, and notes several scenarios in which mutual learning could be applied. Of particular interest to this thesis are the situations in which agents mutually learn to choose the optimal actions. The authors explore this problem in detail with simple bandit problems, where agents learn the optimal action according to a simple algorithm, and provide several interesting experiments to drive discussion about how these agents could share knowledge. The authors continue their exploration specifically in the area of reinforcement learning in [20] and establish a number of factors that must be considered when considering mutual learning in a system:

- **Communication** – how often, and in what way, will the mutually learning agents communicate?
- **Learning algorithm and knowledge representation** – do the agents use the same, or different, learning algorithms? Do they share the same knowledge representation? If they are different, how should they translate one another's knowledge?
- **Trust** – how much credence does each agent lend to the information provided by the other agents? Note that this is not only a question of malicious or benevolent mutual agents, but also simply that some agents may have weak knowledge in certain areas of the problem.
- **Goal congruence** – are both agents attempting to achieve the same goal?

An interesting development in mutual learning for deep learning situations was presented in [21]. Here, the authors present a mutual learning algorithm for image classification inspired by model distillation. The presented algorithm trains a number of neural networks simultaneously, and modifies the loss function of the classification task to add the Kullback-Leibler divergence among these networks. In the case of two mutually learning networks, then, the loss function becomes that shown in equation 1.11, where p_1 and p_2 are the distributions produced by each of the networks.

$$L = L_{C_k} + D_{\mathrm{KL}}(p_2||p_1) \tag{1.11}$$

Interestingly, authors show that mutual learning in this way produces better results than training the networks by themselves, and that small networks trained mutually with one another can produce better results than distilling a small network from a large, static "teacher" network.

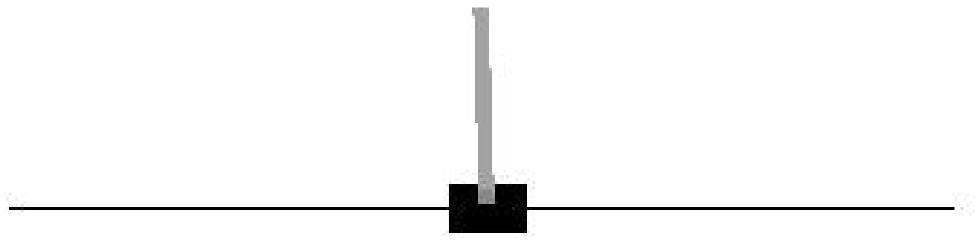


Figure 1.1. Sample state of the cart-pole problem

1.6 Canonical Problems in Reinforcement Learning

In the field of reinforcement learning, there are many classic benchmark problems which attempt to simplify the control problem. Many of these problems are helpfully implemented in the *OpenAI Gym* [22], which has been a boon to reinforcement learning research.

Among these is the *cart-pole* problem, described in [23]. In this problem, the agent is in control of a mobile cart, with a freely spinning pole attached to the top. The goal in this environment is to keep the pole balanced above the cart for as long as possible. The episode ends if cart or the pole move too far from center, and the agent gets 1 point for every step that the episode continues. For reference, Figure 1.1 shows a sample state of the cart-pole problem as implemented in [22].

Additionally, the so-called *Atari domain* describes a number of Atari games on which a seminal paper in deep reinforcement learning is based ([6]). Typically, the problem is formulated as taking the visual screen from the game as a matrix of pixel values and learning to estimate the Q value with a convolutional neural network. While model-based learning in this domain is challenging due to the difficult nature of predicting future screens based on previous screens and actions, attempts have been made on this front with promising results [24], [25].

1.7 Tools for Reinforcement Learning

As deep learning and reinforcement learning become popular and profitable within the industry, the open source toolset is expanding rapidly.

PyTorch [26] is a popular library for deep learning which provides automatic differentiation, supports most typical deep learning architectures, and implements most of the important optimization functionality needed to train deep neural networks. The deep learning work behind the experiments in this paper is implemented in PyTorch 1.3.1.

NumPy [27] is another very popular library for efficient scientific computing in Python based on vectorization and array programming. Significant portions of the work presented in this paper were implemented using NumPy for efficient computation.

2. MUTUAL Q-LEARNING

Given the promise shown in deep mutual learning and the direction of research in student/teacher and transfer learning, I wanted to explore the application of a similar technique to reinforcement learning. In contrast to the work in student/teacher learning, I do not implement transfer learning in the form of action advice; rather, I take inspiration from the mutual learning literature to augment the loss function to force the mutually learning agents into agreement. The work presented in this chapter was published in [28].

2.1 Methods

For this work, I compare two learning agents which are operating in the cart-pole environment. Both use an ϵ-greedy policy and the same simple network architecture: four inputs receive the state from the problem and apply a linear transformation with a leaky rectifier, followed by a densely-connected hidden layer with 8 leaky rectifier units, followed by a final output layer with 2 linear outputs, each of which returns the corresponding action value for the given state.

Training is done using experience replay at each step of the simulation, with 100 experience tuples $\langle s, a, r, s' \rangle$ sampled from an effectively infinite memory buffer. The tuples are divided into batches of 32 and I use Adam [29] to optimize the parameters. No learning takes place until 100 experience tuples have been collected. Our ϵ-greedy policy starts with $\epsilon = 1.0$ and decays by a factor of 0.999 after each step where learning takes place, with a minimum value of 0.01. Because this task is episodic and relatively short, I use a discount factor $\gamma = 1.0$; that is, no discounting. Implementation of all deep learning functionality, including gradient computation, Adam optimization, and layer functionality is provided by PyTorch 1.3.1 with default initialization and hyperparameters.

The agents learn for 5000 simulation steps. To measure learning progress, learning is paused every 100 steps and 5 evaluations are performed using a policy which is purely greedy according to the learned Q function and the total reward obtained is recorded. This generates a learning curve which shows how the agent performs as learning progresses.

I perform this experiment using two agents using different loss functions. In the standard setup, we use the canonical mean squared error loss as shown in Eq. 2.1.

$$\mathcal{L} = \frac{1}{N}\sum_{\mathrm{i}=0}^{N}((\hat{v}(s'_{\mathrm{i}}) + r_{\mathrm{i}}) - \hat{Q}(s_{\mathrm{i}}, a_{\mathrm{i}}))^2 \tag{2.1}$$

In the novel mutual learning setup, I optimize m Q functions with an additional mutual learning term added to the loss function, so that the error includes not only the temporal difference error but also the error between pairs of the m functions, as shown for $\hat{Q}_{\mathrm{i}}$ in Eq. 2.2.

$$\begin{aligned}\mathcal{L}_{\mathrm{M}} = \frac{1}{N}\sum_{\mathrm{j}=0}^{N}((\hat{v}(s'_{\mathrm{j}}) + r_{\mathrm{j}}) - \hat{Q}_{\mathrm{i}}(s_{\mathrm{j}}, a_{\mathrm{j}}))^2 \\ + \frac{C}{m-1}\sum_{k=0}^{m}\mathcal{D}(\hat{Q}_{\mathrm{i}}, \hat{Q}_k)\end{aligned} \tag{2.2}$$

Here, C is a configurable parameter which controls the importance of the mutual learning error term and $\mathcal{D}(\hat{Q}_{\mathrm{i}}, \hat{Q}_{\mathrm{j}}$ is the mean squared error between a pair of Q functions as shown in Eq. 2.3. I normalize the mutual learning error by the number of mutually learning agents so that it doesn't overwhelm the temporal difference error.

$$\mathcal{D}(\hat{Q}_a, \hat{Q}_b) = \frac{1}{N}\sum_{\mathrm{i}=0}^{N}(\hat{Q}_a(s_{\mathrm{i}}, a_{\mathrm{i}}) - \hat{Q}_b(s_{\mathrm{i}}, a_{\mathrm{i}}))^2 \tag{2.3}$$

The policy of the mutual learning implementation is ϵ-greedy with respect to the primary Q approximation, i.e., $\hat{Q}_1$. The other Q estimators do not influence the policy except in that they contribute to the optimization of $\hat{Q}_1$.

2.1.1 Notes on Convergence

In order to achieve convenient mathematical properties for convergence, note that it may be useful to decay the importance of the mutual learning term to zero as training progresses, such that the mutual Q-learning algorithm reduces to standard Q-learning.

When this is desired, I propose replacing C in Eq. 2.2 by a function $\omega\colon \mathbb{Z} \to \mathbb{R}^+$, with the requirement that $\lim_{\mathrm{i}\to\infty} \omega(\mathrm{i}) = 0$. For the simulations, I use a mirrored sigmoid function as shown in Eq. 2.4, parameterized by μ and σ.

This function is convenient for a number of reasons. It gives sufficient time for mutual learning to benefit learning before tapering down to 0; it is simply parameterized to adjust the rate of decay; and it is bound between 0 and 1 to allow simple tuning of the weight of the mutual learning term.

$$\omega(\mathrm{i};\mu,\sigma) = \frac{-1}{1+\mathrm{e}^{\frac{-x+\mu}{\sigma}}} + 1 \tag{2.4}$$

2.2 Simulation Results

2.2.1 Impact of Mutual Learning

First, I present results for a baseline implementation of the mutual reinforcement learning algorithm using two mutually learning Q estimators and constant $C = 1$. Figure 2.1 shows the impact of the additional Q function estimator and using a mutual learning approach.

Performance is significantly improved in the mutual learning case, with the mutual learning implementation achieving an average evaluation score of approximately 218 by the final step of the learning simulation, compared to an average of 194 for the standard Q learning case. Notably, both approaches learn at about the same rate until approximately halfway through training, at which point the standard Q learning approach slows down much more quickly than the mutual Q learning approach; after 2500 steps, the mutual learning implementation always performs better. See Figure 2.2 for details.

2.2.2 Varying the Weight of the Mutual Learning Term

I also present results from a number of simulations to demonstrate the impact of the weight of the mutual learning term. Figure 2.3 shows these results.

$C = 1$ appears to improve performance; however $C = 0.5$ appears to have a minimal impact over traditional Q learning. Furthermore, $C = 2.0$ appears to hinder learning early

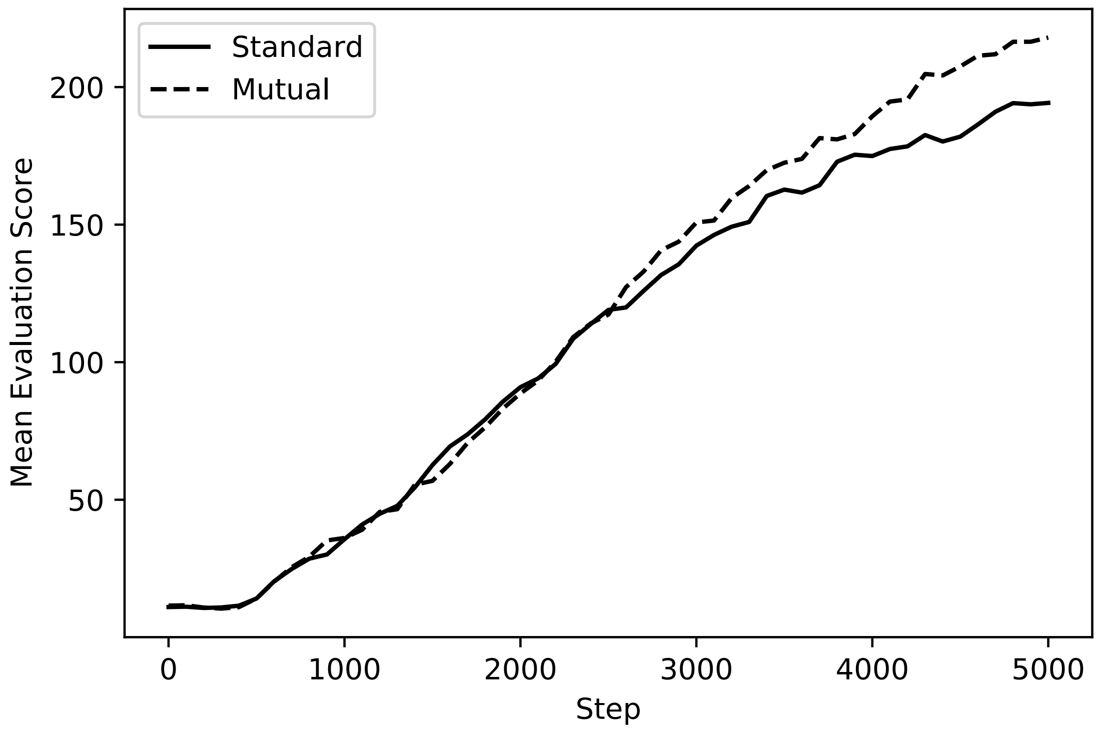
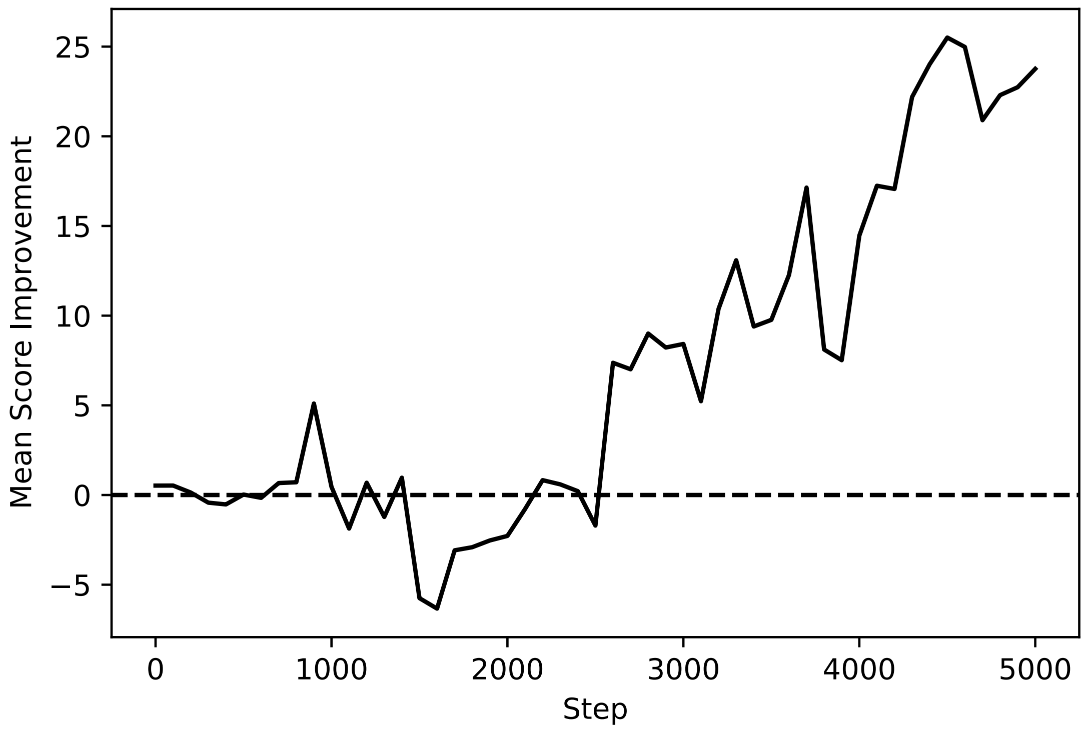


Figure 2.1. Comparison of standard Q learning and mutual Q learning with 2 estimators



[H]

Figure 2.2. Difference in evaluation scores of mutual vs. standard implementation.

in training. Presumably, this is because the mutual learning loss overwhelms the TD error and drives the functions toward similar results despite the fact that they are both poor approximations of the true Q function.

These results indicate that a value of $C = 1.0$ is optimal; thus, I use this value through the rest of the simulations.

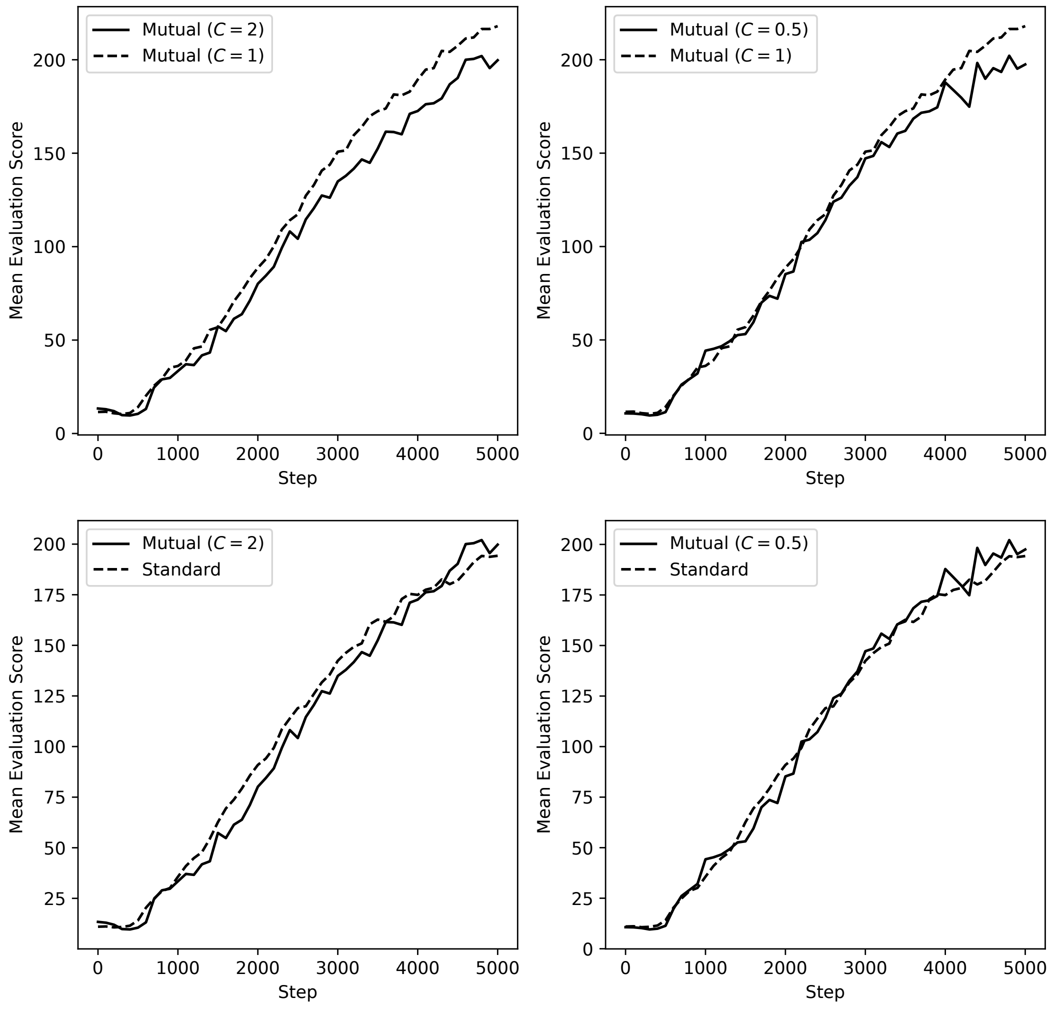


Figure 2.3. Impact of varying the weight of the mutual learning term

2.2.3 Increasing the Number of Mutually Learning Q Estimators

Figure 2.4 shows the impact of increasing the number of mutually learning estimators beyond 2. With 10 Q-functions trained in conjunction as described in this paper, a significant improvement can be seen over both mutual learning with 2 estimators and over traditional Q learning.

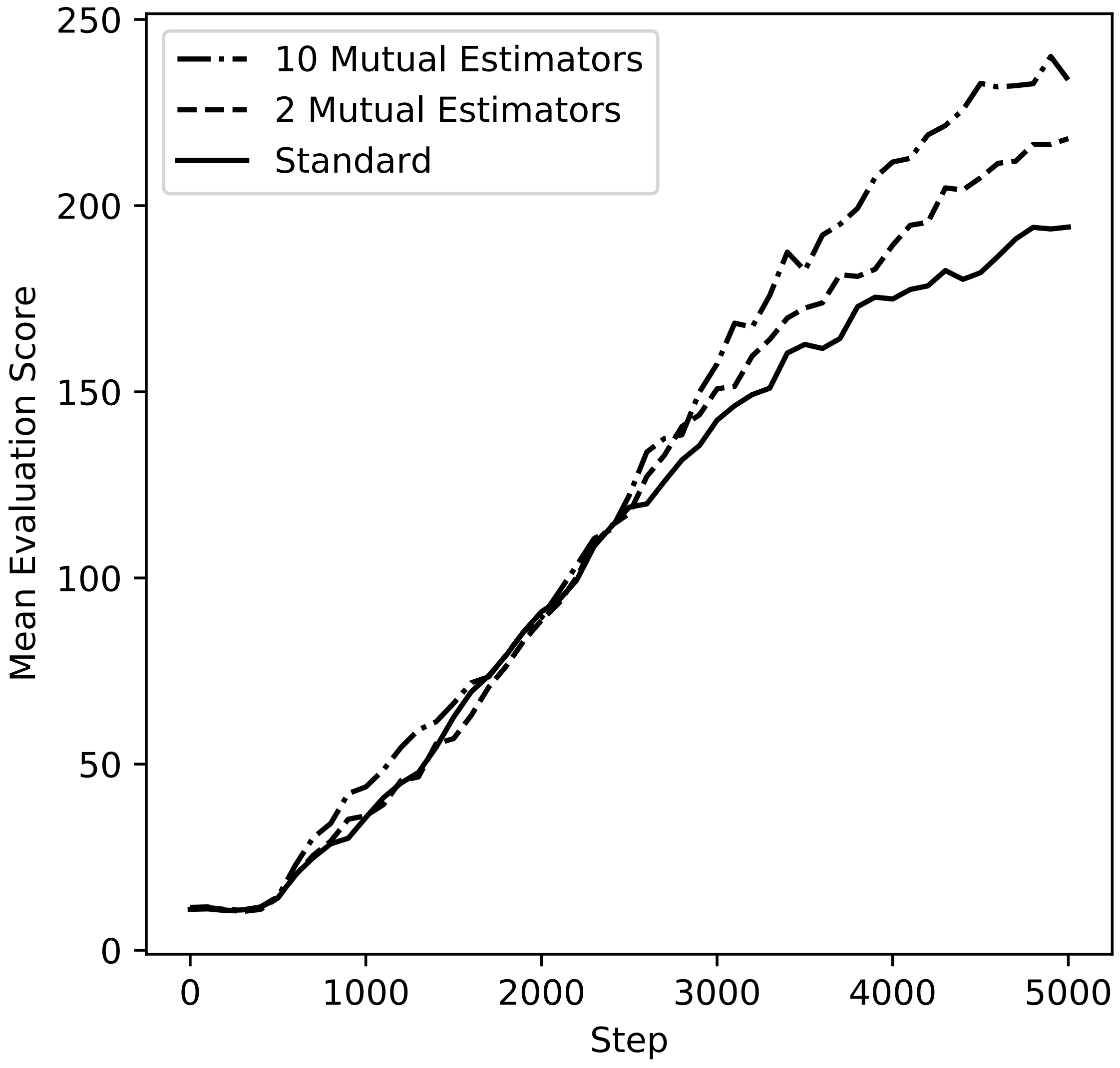


Figure 2.4. Impact of varying the number of mutually learning estimators

With ten mutually learning estimators, the agent achieves an average peak evaluation score of approximately 240, which represents a 27% improvement over the Q learning baseline and a 15% improvement over the mutual learning algorithm with only two estimators.

While more estimators appear to be more effective, it's important to consider that each additional agent added to the system increases the amount of communication overhead by a factor of $\mathbb{O}(n^2)$.

2.2.4 Decaying the Mutual Learning Term Weight

Figure 2.5 shows how the algorithm performs with decaying the mutual learning term weight according to Eq. 2.4. In this case, the final mean score is an improvement over standard Q-learning, but not as significant as mutual Q learning without decay.

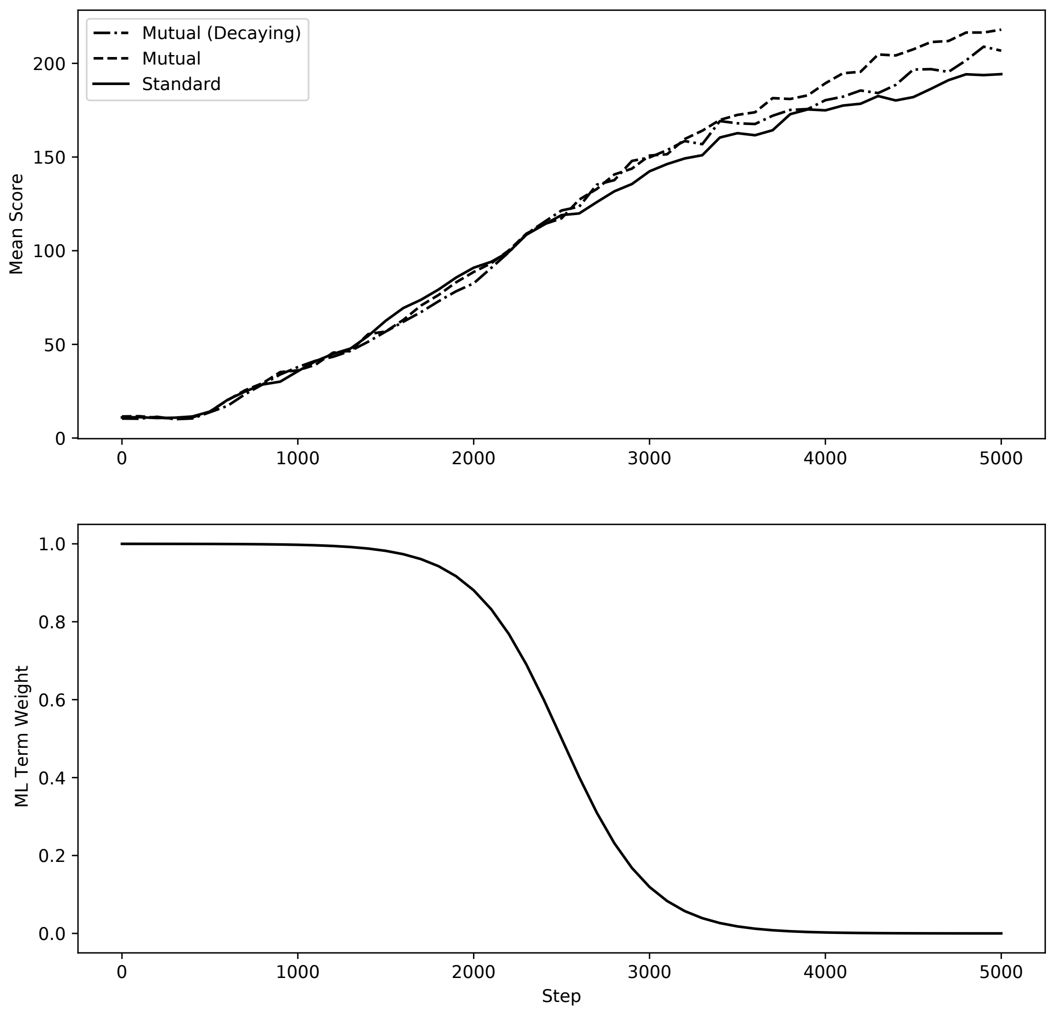


Figure 2.5. Impact of reducing the weight of the mutual learning term according to the mirrored sigmoid schedule with $\mu = 2500$ and $\sigma = 250$

However, if the mutual learning weight is kept higher for longer by shifting μ, better performance can be achieved. Figure 2.6 shows the results of this simulation: although the mutual learning term decays to only 1.8% of its initial value, performance continues to grow at a rate similar to the non-decaying implementation.

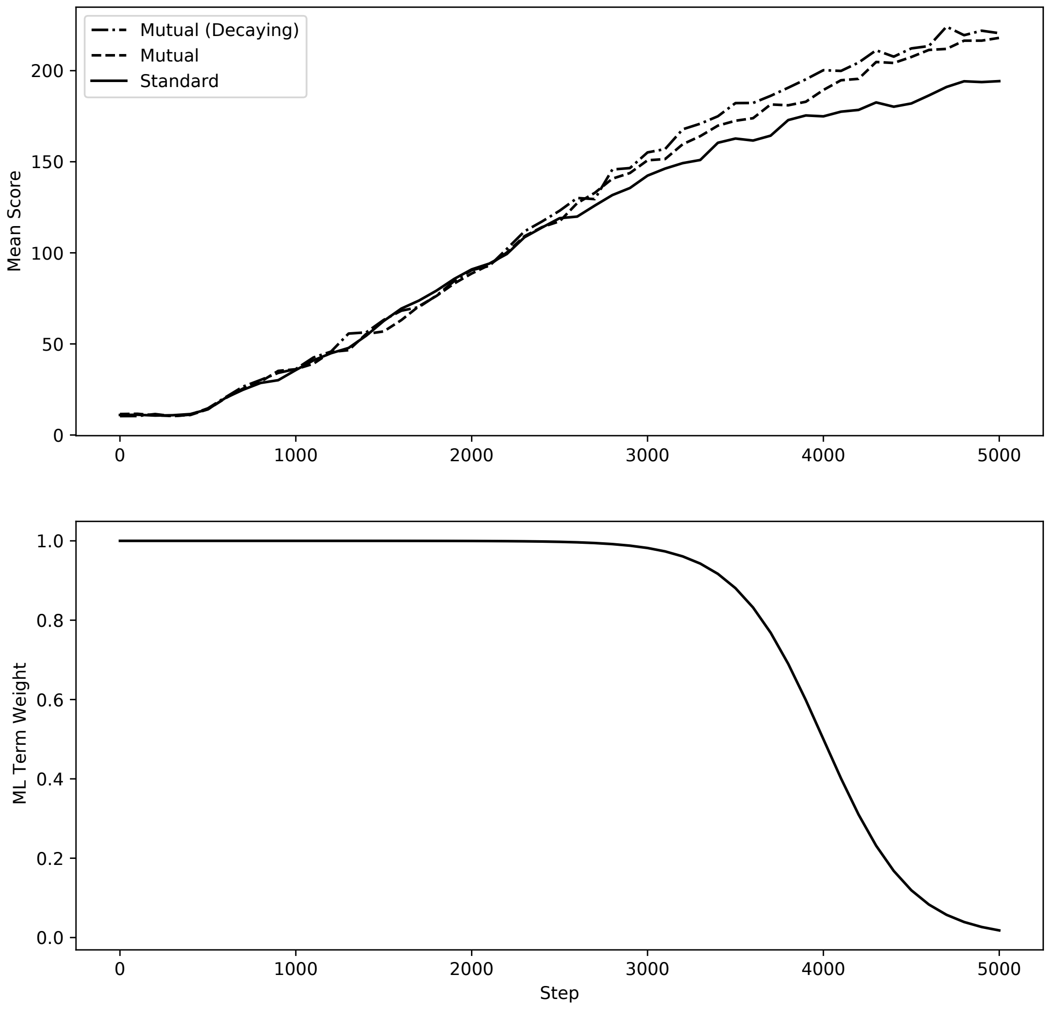


Figure 2.6. Impact of reducing the weight of the mutual learning term according to a mirrored sigmoid schedule with $\mu = 4000$ and $\sigma = 250$

2.3 Discussion

In this section, I presented a Q learning algorithm which is improved by the addition of a mutual learning factor. Notably, adding one agent to the system and using this mutual learning algorithm appears to have a modest but noticable impact on performance. Perhaps more interesting, increasing the number of mutually learning agents from 2 to 10 appears to improve performance still further. However, the communication overhead scales as $\mathcal{O}(n^2)$, so adding more agents requires significantly more computation to achieve that. This is a significant limitation that I do not attempt to address here.

This algorithm also makes no attempt to quantify trust among the mutually learning agents. It is interesting, then, that blind trust at all steps still leads to improved performance.

3. HETEROGENEOUS MUTUAL LEARNING

In addition to the work in Chapter 2 of this thesis, I also wanted to explore how agents which learn in very different ways might communicate. In particular, model-based methods such as approximate dynamic programming (or *ADP*) can achieve great performance with significantly fewer interactions with the environment than are necessary for a model-free method. On the other hand, model-free methods generally require significantly less computation for each update.

In this chapter, I describe how I approached the problem of allowing a model-free learning agent using deep Q-learning to receive knowledge from a model-based learning agent using ADP. The work presented here is submitted for publication as [30].

3.1 Methods

Because ADP generates both a model of the environment dynamics F and an estimate of the state value function V, one can derive an estimate of the Q-function $\hat{Q}_{\text{ADP}}$ by using the estimate of the V function of the state which, according to the transition function, results after taking an action, as shown in Eq. 3.1

$$\hat{Q}_{\text{ADP}}(s, a) = \mathbb{E}\left(\hat{V}(F(s, a))\right) \tag{3.1}$$

This estimate is correct when the environment and reward models are correct. However, this is not often the case, and so the DQL agent must account for sources of error.

First, recall that ADP requires a discretized representation of the state space, while DQL takes continuous states directly. This means that a range of continuous values will all appear the same to the ADP agent. If the states are distributed uniformly across the state space, the estimate produced by 3.1 will be good when the continuous state is near the middle of a bin, but grow worse as it gets away from that value. To correct for this, the DQL agent may weight the information it gets from the ADP agent by the cosine similarity of the true state value with the centroid of the binned state. This allows the DQL agent to essentially disregard information from the ADP agent which is likely inaccurate.

Second, the discrete state space that the ADP agent must explore grows exponentially with the dimension of the true state space. As such, even after many interactions with the environment, the ADP value function may be fairly sparse, and the estimates of the transition and reward functions may only be informed by a few samples. To account for this, I suggest an approach similar to that taken in [31] and add a weighting factor that scales with the number of visits that the ADP agent has taken to a state. I refer to this weighting term as the *insistence* of the ADP agent and denote it $\mathbf{I}$, as shown in Eq. 3.2, where N_s is the number of times that the ADP agent has visited state s. This term is bound between 0 and 1, and approaches 1 when the ADP agent has seen the state in question many times, indicating higher confidence in the model.

$$\mathbf{I}(s|\text{ADP}) = 1 - \frac{1}{\sqrt{N_s}} \tag{3.2}$$

Third, we must also consider that the ADP algorithm, by its nature, will generate Q values which are near the theoretical convergence value, while DQL with an appropriate learning rate and target Q-network will produce Q-values which grow very slowly, but whose relative values are still insightful. For example, consider a trivial task which never ends, with only a single possible state and action, and which provides the agent with a reward of 1 every step, and assume $\gamma = 0.99$. After a single step, the ADP agent will produce a Q-value very near to $\frac{1}{1-\gamma} = 100$, which is the true long-term value of the only action in the only state. However, depending on initialization conditions, the Q-network will only update a bit in the direction of that value; perhaps from 0.1 to 0.11. If you update the target network only periodically, the Q-learning agent will have no hope of approaching the true value for many steps, and the relative scale of the Q-values produced by each agent make them useless for direct comparison. To address this, I further suggest comparing the softmax distribution of the generated Q-values using the Kullback-Leibler divergence, rather than the squared error of the Q-values themselves.

The considerations above result in the mutual learning loss term shown in Eq. 3.3, where $\bar{s}$ indicates the discretized state s, $\mu_{\bar{s}}$ denotes the centroid of the values of that discretized state, and P is the softmax function.

$$\begin{aligned}\mathcal{L}_{\mathrm{M}}(s) = \mathbf{KL}\left(P(\hat{Q}_{\mathrm{DQL}}(s)), P(\hat{Q}_{\mathrm{ADP}}(\bar{s}))\right) \times \\ \times \mathbf{I}_{\mathrm{ADP}}(s) \times \sqrt{\frac{s \cdot \mu_{\bar{s}}}{||s||||\mu_{\bar{s}}||}}\end{aligned} \tag{3.3}$$

3.2 Methods

I implement three learning agents in the cart-pole environment. The first learns via Q-learning; the second learns via adaptive dynamic programming; and the third learns via a mutual learning algorithm informed by the discussion above.

To evaluate their learning progress, I run the agents in the cart-pole problem for 5000 steps. Every 100 steps, I pause training and run a number of evaluations which use a deterministic greedy policy, and I record the mean episode reward from those evaluations. 5000 steps constitute a single trial, and results presented here are averaged over 100 trials.

3.2.1 Q-Learning Agent

For the Q-learning agent, the Q function is approximated by a simple neural network, similar to the one discussed in Chapter 2: 4 input nodes take the state from the environment as input. There is a single dense hidden layer with 8 units which applies a leaky rectifier, and a linear output layer with 2 units, each of which outputs a corresponding estimated action value for that state.

Training is done using experience replay [7] at each step of the simulation, with 100 tuples sampled from an effectively infinite memory buffer (the agent will have access to all experiences of the trial). The discount factor was set to 0.99. Optimization was done using Adam [29]. In contrast to the experiments of Chapter 2, a target network was used to generate the update targets. The target network was updated every 100 steps.

Once again, all deep learning functionality was provided by PyTorch 1.3.1 [26] with default initialization and hyperparameters unless otherwise noted.

3.2.2 ADP Agent

For the ADP agent, the state space is first discretized into 7 evenly-sized bins. The agent builds the models of the environment and reward functions by simply keeping track of the number of the number of times it has visited a state, the actions it took in that state, and the resulting next state and reward, and then computing the necessary probabilities empirically. At each step, the agent iterates a dynamic programming pass with discount factor $\gamma = 0.99$ until the maximum error value seen across any update is less than 0.01.

3.2.3 Mutual Learning Algorithm

We use Eq. 3.3 to develop an algorithm which "drives" a single agent comprising a DQL algorithm and an ADP algorithm. This allows for a simpler analysis, as both algorithms will encounter exactly the same set of experience tuples in the same order.

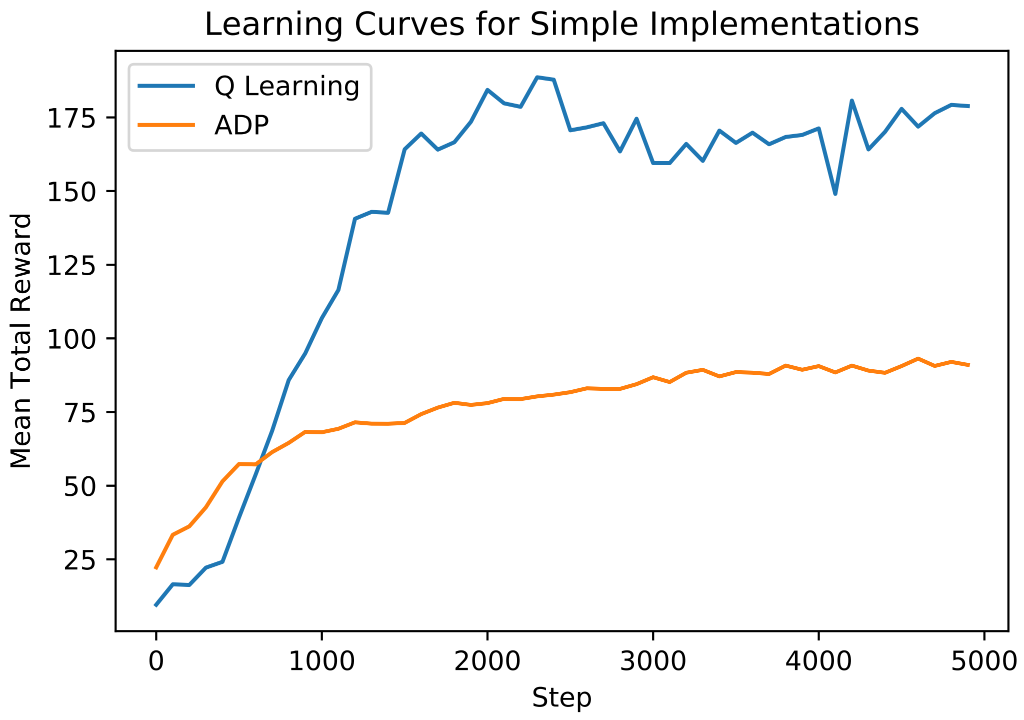
We modify the traditional deep Q-learning algorithm by minimizing 3.3 in addition to the typical temporal difference loss used for Q learning. That mutual loss term is computed purely on-line with the state under consideration, rather than drawing samples from the replay buffer. Algorithm 1 outlines the general procedure.

Algorithm 1: Heterogeneous Mutual Learning

```
Initialize deep Q learning agent DQL parameterized by $\theta$
Initialize ADP agent $A$
Initialize $\boldsymbol{\pi}_{\text{DQL}}$, for example $\epsilon$-greedy
Initialize the environment with initial state $s$
for each step of training process do
    $a \leftarrow \boldsymbol{\pi}_{\text{DQL}}(s)$
    Take action $a$, observe reward $r$ and new state $s'$
    Handle transition $\langle s, a, r, s' \rangle$ in Q learning agent and ADP agent as normal
    $\omega_l \leftarrow \text{similarity}(s, \mu_{\bar{s}}) \times \mathbf{I}_A(s)$
    $l = \omega_l \mathbf{KL}\left(P(\hat{Q}_{\text{DQL}}(s, \star)), P(\hat{Q}_{\text{ADP}}(s, \star))\right)$
    $\theta \leftarrow \theta - \nabla_\theta l$
end
```

3.3 Simulation Results

First, I provide the learning curves of our unaltered algorithms for comparison. Figure 3.1 shows those results.

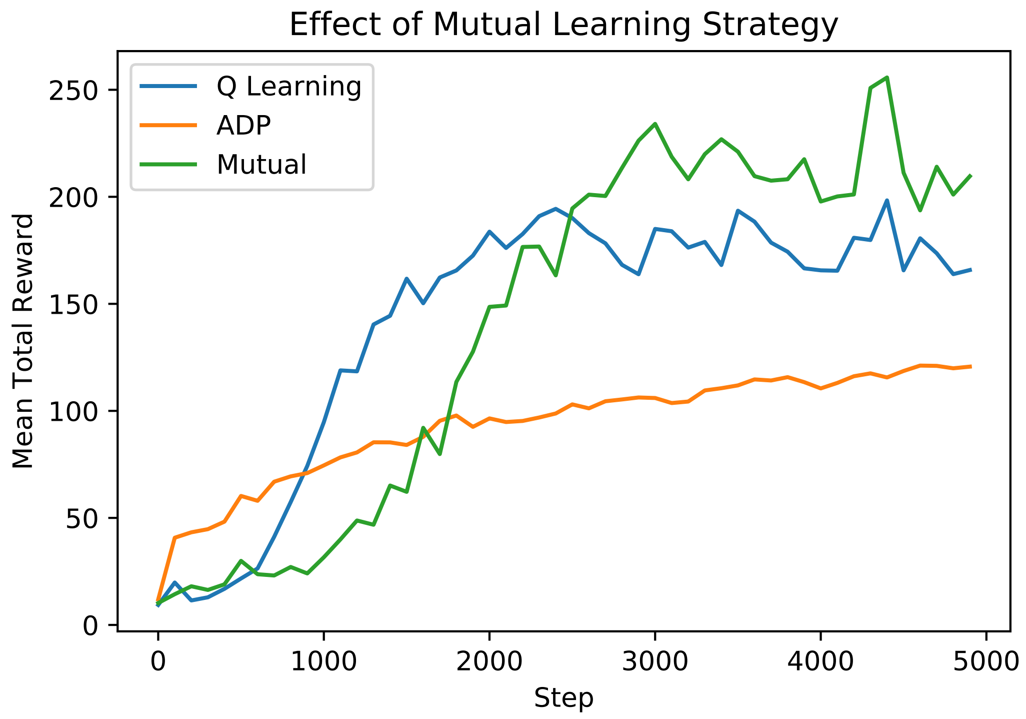


[H]

Figure 3.1. Mean total reward achieved by each learning algorithm during evaluation over the course of training

This shows that, as expected, ADP outperforms Q-learning for small numbers of interactions, and then the DQL agent begins to outperform the ADP agent once it has collected enough experience to take advantage of its ability to generalize and distinguish finer details.

Second, I show a comparison of these learning curves to an implementation of Algorithm 1 in Figure 3.2



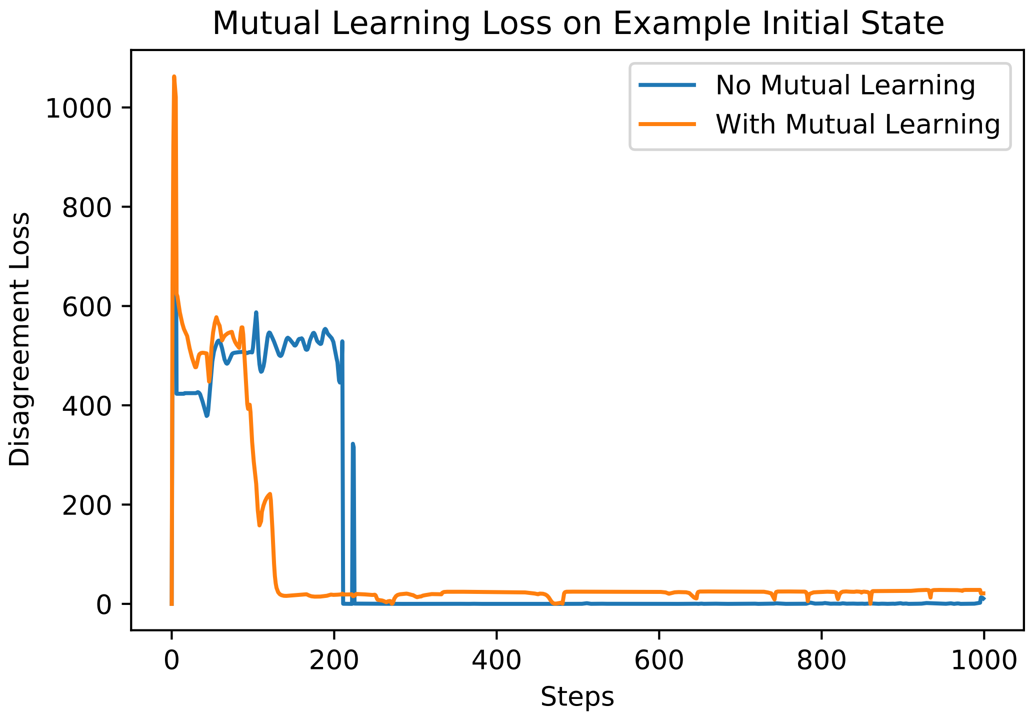
[H]

Figure 3.2. Comparison of the learning curves of the standard deep Q-learning implementation and the implementation informed by our mutual learning analysis. Initially, the model informing the ADP is poor and hinders learning for the DQL agent; however, once the ADP agent has built a reasonable model, it gives good advice to the DQL agent and improves performance.

This shows the impact of including the mutual learning term in the loss function. Initially, the DQL agent receives bad information from the ADP agent due to insufficient samples informing its model. As training progresses, the ADP model in the ADP agent gets better, the advice provided by the ADP agent improves. With the cosine similarity weighting, the DQL agent is able to discount advice about states the ADP agent is likely to have a poor model of because of the discretization. The result is that the mutual learning algorithm achieves moderately improved performance over either of the constituent algorithms on average.

Next, I examine experiments where a DQL agent, an ADP agent, and an agent trained with Algorithm 1 were applied to the same set of 1000 training steps. The training data was constructed by running an ADP algorithm in the Cart-Pole environment used for the other experiment. Then, experience tuples were fed to each algorithm in order. I hold a set of states and record the mean value of Eq. 3.3 on those states as training progresses for both the unaltered DQL/ADP pair and the mutual learning algorithm.

Figure 3.3 shows how the disagreement term in Eq. 3.3 changes with and without including it in the loss function.



[H]

Figure 3.3. Comparison of the disagreement term when applied to the loss function. This plot shows disagreement on an example initial state.

As expected, including Eq. 3.3 in the loss function causes the agents to reach a measure of agreement more quickly than they do without the loss function.

4. CONCLUSIONS AND FUTURE WORK

Mutual learning is proving to be an exciting area of research for allowing simpler models to achieve better performance. In this thesis I have presented what I believe is the first work discussing mutual learning for deep reinforcement learning systems. I have also explored in depth the problems of mutual learning for systems which use very different learning methods, and I've discussed some methods for solving those problems. In doing so, this work also begins to address the issue of translating knowledge between agents and establishing trust among agents in the context of mutual learning, which are fundamental requirements of effective mutual learning.

While the work presented here is a good first step, I believe there is significant opportunity for further research.

4.1 Mutual Reinforcement Learning in High Dimensional Problems

While the cart-pole problem is a useful problem to study, the system is described completely by four variables. However, many important practical applications of reinforcement learning involve very high-dimensional state spaces. Further research could be directed to exploring how the techniques presented here could be applied to much higher-dimensional problems. For example, a typical formulation of problems in the Atari domain involves states consisting of 4 stacked RGB frames of dimension 210×160. This implies a $210 \times 160 \times 3 \times 4 = 403,200$ dimensional state-space, and that represents one of the most simple visual reinforcement learning problems. It will be important to explore the performance of mutual reinforcement learning in these spaces so that the techniques presented in this thesis could be applied to more practical, real-world problems.

4.2 Mutual Policy Gradient Algorithms

As discussed in Section 1.3, a significant amount of the field of deep reinforcement learning for robotics is dominated by policy-gradient, actor-critic algorithms. While it's possible that the mutual learning techniques presented will help the critic learn more effectively, that should be explored in future research. Furthermore, there may be interesting applications of mutual learning which allow the actor function to also learn more quickly. Future research that explores improvement of actor-critic methods with the application of mutual learning could have significant impact in real-world domains such as robotics and self-driving.

4.3 Mutual Learning in Different Task Formulations

In this work, we only study the problem of mutual learning when the agents are attempting to learn exactly the same problem – that is, the state space, action space, reward function, and transition functions are all identical. One interesting area of exploration would be how agents can communicate when some of these are different. For example, imagine a scenario where you have one bipedal robot and one quadrupedal robot, and you wish to achieve mutual learning on a task which involves walking on rough terrain. The gait that each robot learns will clearly be quite different from robot to robot. However, imagine now that the learning scenario includes occasional canyons which must be leapt over. At a low level (i.e., the torques applied to motors), the process of jumping will clearly be different. However, there will be commonalities: build up speed, apply significant power near the edge, and so on. In this scenario and in many other real-world scenarios, there are opportunities for mutual learning which do not exactly line up with the output of the Q function, or the policy function, or any other obvious place. However, a human can watch a gazelle leaping through a field and (perhaps poorly) mimic that behavior. Future research should explore how we can achieve transfer learning in these more abstract, higher-level actions.

Similarly, consider the Atari domain. This problem set consists of many different video games, but most of these games belong to one of a few very broad classes of games. For example, Asteroid, Centipede, and River Raid all share a format (enemies move vertically down the screen, and you can shoot or avoid them). In essentially all of these games, everything on the screen represents either a boundry, an obstacle, a power-up, or the player, and many human players can tell at first look which class a given object belongs to. While research has been done in the field of transfer learning for problems like this, future research could formulate a mutual learning problem of many agents playing many different games and sharing knowledge among one another to learn more quickly.

4.4 Limits of Mutual Learning

As noted in Chapter 2, increasing the number of mutually learning agents in a system adds a significant amount of communication overhead to the problem, which scales with $\mathcal{O}(n^2)$ (where n is the number of agents), but adding more agents appears to improve performance. Future research should explore this trade-off, and attempt to address the question: how does performance improve as we add more agents? Additionally, researchers may look to adapt the "communication budget" mentality that exists in the student/teacher advising literature and explore techniques inspired by that work to reduce the amount of inter-agent communication without sacrificing increased performance (or, indeed, achieving better performance by avoiding negative transfer).

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[2] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[3] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[4] A. G. Barto, S. J. Bradtke, and S. P. Singh, “Learning to act using real-time dynamic programming,” *Artificial intelligence*, vol. 72, no. 1-2, pp. 81–138, 1995.

[5] R. Bellman, “Dynamic programming,” *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[7] L.-J. Lin, “Self-improving reactive agents based on reinforcement learning, planning and teaching,” *Machine learning*, vol. 8, no. 3-4, pp. 293–321, 1992.

[8] H. Van Hasselt, Y. Doron, F. Strub, M. Hessel, N. Sonnerat, and J. Modayil, “Deep reinforcement learning and the deadly triad,” *arXiv preprint arXiv:1812.02648*, 2018.

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” *arXiv preprint arXiv:1509.02971*, 2015.

[10] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch, “Emergent tool use from multi-agent autocurricula,” *arXiv preprint arXiv:1909.07528*, 2019.

[11] T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch, “Emergent complexity via multi-agent competition,” *arXiv preprint arXiv:1710.03748*, 2017.

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv preprint arXiv:1707.06347*, 2017.

[13] K. Weiss, T. M. Khoshgoftaar, and D. Wang, “A survey of transfer learning,” *Journal of Big data*, vol. 3, no. 1, pp. 1–40, 2016.

[14] L. Torrey and M. Taylor, "Teaching on a budget: Agents advising agents in reinforcement learning," in *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, 2013, pp. 1053–1060.

[15] F. L. Da Silva, R. Glatt, and A. H. R. Costa, "Simultaneously learning and advising in multiagent reinforcement learning," in *Proceedings of the 16th conference on autonomous agents and multiagent systems*, 2017, pp. 1100–1108.

[16] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," *arXiv preprint arXiv:2006.05525*, 2020.

[17] C. Bucilua, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 535–541.

[18] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[19] K. S. Narendra and S. Mukhopadhyay, "Mutual learning: Part i-learning automata," in *2019 American Control Conference (ACC)*, IEEE, 2019, pp. 916–921.

[20] K. S. Narendra and S. Mukhopadhyay, "Mutual learning: Part ii–reinforcement learning," in *2020 American Control Conference (ACC)*, IEEE, 2020, pp. 1105–1110.

[21] Y. Zhang, T. Xiang, T. M. Hospedales, and H. Lu, "Deep mutual learning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4320–4328.

[22] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Openai gym*, 2016. arXiv: 1606.01540 [cs.LG].

[23] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE transactions on systems, man, and cybernetics*, no. 5, pp. 834–846, 1983.

[24] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, *et al.*, "Model-based reinforcement learning for atari," *arXiv preprint arXiv:1903.00374*, 2019.

[25] D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba, "Mastering atari with discrete world models," *arXiv preprint arXiv:2010.02193*, 2020.

[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[27] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.

[28] C. Reid and S. Mukhopadhyay, "Mutual q-learning," in *2020 3rd International Conference on Control and Robots (ICCR)*, IEEE, 2020, pp. 128–133.

[29] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. arXiv: 1412.6980 [cs.LG].

[30] C. Reid and S. Mukhopadhyay, "Mutual reinforcement learning with heterogeneous agents," Forthcoming.

[31] F. L. Da Silva, R. Glatt, and A. H. R. Costa, "Simultaneously learning and advising in multiagent reinforcement learning," in *Proceedings of the 16th conference on autonomous agents and multiagent systems*, 2017, pp. 1100–1108.