Thinking fast and slow in intelligent systems

Peter Norstein

Thesis submitted for the degree of
Master in Robotics and Intelligent Systems
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2020

Thinking fast and slow in intelligent systems

Peter Norstein

© 2020 Peter Norstein

Thinking fast and slow in intelligent systems

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Humans have the ability to quickly and efficiently solve complex tasks by leveraging fast heuristical thinking, as well as slow methodical thinking. Taking inspiration from human cognition, this thesis presents a framework for identifying whether an experience is likely to be within the same task structure as previously experienced tasks.

Reinforcement Learning is the process of learning an unknown function from several trials where a reward is given to the agent. This approach is close to our current understanding of human learning through dopamine and reinforcement.

Identifying what type of domain is being presented to the agent is a prerequisite for a general artificial intelligence. This thesis will take inspiration from psychology and the theory of cognition based on two systems, one fast system and one slow system. The slow system is resource intensive and should be used sparingly. This thesis presents an approach to identify whether the slow system needs to be invoked or the fast system is sufficient for the given task. The system proposed is analogous to the evaluation done by the slow system on whether it needs to modify the decision by the fast system.

Using the fast network to obtain a solution and testing it in the environment will give some feedback on the performance of this action. This can be used to determine if the slower system should be invoked next time.

Distinguishing between using the fast and the slow system is a prerequisite for two system based cognition. We present Task Identification During Encounters, TIDE, an online task recognition algorithm used to aid a two system framework with selecting which of the two systems to use. Determining which system to use is done by designating a task as known or unknown.

Choosing whether the fast network is sufficient to solve the task without receiving feedback from the environment is challenging. Using the procedure presented in this thesis, differentiating between tasks is achieved with high accuracy on simple structured task descriptors.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor, Jim Tørresen, for his support and encouragement. I would also like to thank my fellow students who created a productive work environment. It was sorely missed in the latter part of the work, as we were all required to work from home.

Chapter 1

Introduction

General artificial intelligence is yet to be achieved in the artificial intelligence community. Much progress has been made in specialised intelligence in the last decade, especially in the domain of machine learning.

Specialised machine learning algorithms have high precision and good performance in a narrow field [24]. Learning or mastering multiple different domains remains a challenge for these highly successful approaches. Promising work is being done in the domain of meta-learning to mastering multiple different tasks within a similar domain, for example solving a walking challenge with changing slopes or control differences.

1.1 Kahneman's theory of human thinking, fast and slow

Thinking, fast and slow is the concept of human thought being controlled by two modes of thought, the fast system and the slow system. The fast mode is used in most daily interactions and solves problems with a generally known solution. The slow mode is the rational and methodical mode that obtains new knowledge, understands the world and creates models of it. The slow mode generates these models and makes them available to the fast mode, which then uses it to react to the world.

A person walking along a familiar road uses the fast system to position its legs and compensate for rough terrain. They do not, and should not, thoroughly consider the dynamics of their legs and the surface of the road to decide how to walk.

If you have tried to concentrate on a seemingly simple action, thereby interrupting your fast system, you might have noticed that some actions become frustratingly difficult when deliberated on. Some actions are not easy for humans to perform by thinking deeply, but are simple when solved quickly. One interpretation is that the slow system 'teaches' the fast system how to solve an action that the slow system would not be able to *perform* in realtime.

1.2 Machines thinking, fast and slow

General artificial intelligence is still far away for the machine learning community. Specialised machine learning algorithms have high precision and good performance in a narrow field. A major challenge in machine learning is learning a new task without reducing the performance of previously mastered tasks. It is called catastrophic forgetting when learning a new task destroys previous attained knowledge/performance. Catastrophic forgetting can be avoided somewhat by "learning to learn", also known as meta-learning, which makes the network learn a general structure of all tasks it will be exposed to.

Another approach is to save some information for each type of task the network will be exposed to, and use this information when the task at hand is identified. One such way of saving information is "snapshotting" the current network for a given task when it is mastered and thereby retaining a working solution for that task. This approach requires that the network is replaced after identifying which task it is currently occupied with.

Using the approach of trying to consolidate the knowledge obtained into kernels of "predispositions" could enable a machine to learn multiple tasks and use all these quick kernels at the same time to try to solve any task within its knowledge base. This resembles using multiple models at the same time, as in ensemble learning. However, most ensemble learning problems are focused on obtaining better results by using different algorithms on the same task.

In some regards one could view a deep neural network as the fast system (even though it is slow to train). When given a problem (feature-vector or observation), the solution is calculated quickly by applying the weights of the network to the inputs. Thus it will calculate a solution for all inputs, just as the fast system in humans will generate a solution. This solution might be completely incorrect, as is the case when the task has radically changed.

The slow system is in some papers described as a decision-making algorithm, such as a tree-search algorithm [2].

For a general implementation, the slow system should be able to use previous knowledge to generate better understanding and performance on a new problem. Currently there is no deep learning framework that can replicate this type of capability to my knowledge.

1.3 Confidence in task identification

Knowing if the task given to an algorithm is previously solved or likely to be solvable is important to have confidence in the solution. Humans seem to implicitly evaluate whether the task they are presented with can be solved simply by using the fast system, as opposed to expending more energy using the slow system. Thinking, Fast and Slow present multiple examples of people erroneously relying on the fast system, which points out the shortcomings of humans' ability to predict whether or not they need

to use their slow system to solve a task. This energy saving approach is not often used in machine learning, as the computation time of a fast and "slow" neural network is often comparable at test time. At some point, to recreate cognition at the human level, choosing to use the most efficient system for the task will become important.

1.4 Differentiating between the slow and fast system

Knowing whether to use the fast or slow system can in essence be described as identifying what sort of task you are being presented with. Have I seen it before? Does it have a similar structure to something I have seen before? Will I be able to solve this easily or do I have to think deeply about it? Answering these questions begin with evaluating what the task at hand is, and whether or not it is known to you.

Knowing which task you are presented with might not be evaluated before the fast system has activated and given an answer. If this is the case, the evaluations done by the fast system can be used to decide whether the slow system should be activated. Identifying which task is presented can therefore also use the evaluations of the fast system. If the fast system has already given a solution that corresponds well with the task, no further action needs to be taken. Simply use the solution the fast system has presented.

In some research identifying different tasks is removed from the learning models and instead explicitly given such as in Pritzel et al. [20] where each game of Atari played is done with a freshly reset agent instead of using the same agent in the new task. Notable exceptions are Ritter et al. [22] and Nagabandi et al. [19] which solve task-recognition integrated with solving the task.

Knowing when to use the slow and fast system is the central theme of this thesis, and the questions above are what we try to illuminate with the discussions and experiments.

1.5 Summary of terms

This section will describe terms I use throughout the thesis. Some terms relate specifically to the Multi-Armed Bandit problem that is explained in section 2.2.

1.5.1 Task

A task is a contextually defined problem within a domain. The domain can be Atari-games, while the tasks are different games. In this thesis tasks will be instances of a multi-armed bandit where one specific arm has higher probability of giving reward. All instances of a task will have the arm with higher probability of reward.

1.5.2 Task label

A task label is unique identification of a task. Identifying the same multi-armed bandit by the same task label each time is the goal of this thesis. Each task has a corresponding task label that should be predicted by the algorithm.

1.5.3 Task descriptor

The task descriptor contains the context of the task. From this task descriptor it should be possible to predict the task label. A task descriptor is a general term for something that is supplied in addition to a task for a learning algorithm. A task descriptor should in some way inform the algorithm of the context of the challenge. We will use a barcode as the task descriptor in this thesis. No two task descriptors are identical.

1.5.4 Task descriptor structure

The structure of a task descriptor in relation to the task. If there is some specific relation between a tasks descriptor and the dynamics of it's task, we say that there exists an explicit structure to the tasks. If there is no relation between the task descriptors and the task labels, the tasks are *unstructured*. Unstructured tasks are for example randomly generated task descriptors for each unique task.

1.5.5 Solution agent

Solution agent is the instantiation of an algorithm to solve a problem. We will use it to mean an agent that tries to solve a multi-armed bandit problem. The agents are being selected based on which task is predicted to be the current one. Each agent will see the multi-armed bandit without their context, the agents are only supplied the task label that the algorithm has assigned it. The agents interact with the multi-armed bandits without being informed of which one it sees. This division of responsibility makes it possible to have agents that act as if the different multi-armed bandits they are exposed to are the same one.

1.5.6 Barcode

A barcode is a task descriptor that consists of a number of characters in the set [0,1]. The barcode length is the number of characters in the barcode. Example: "0101101110" is a barcode of length 10 that acts as the task descriptor for a bandit.

1.6 Thesis structure

The structure of this thesis is as follows: (1) Introduction, (2) Background, (3) Project Description, (4) Experiments and Results, (5) Conclusion.

In the Background chapter we will summarise relevant research and knowledge within machine learning. Important topics are reinforcement learning, meta-learning and task identification. We will introduce previous work and how they view learning in machines. We also present the Multi-armed Bandit problem area which we will use later to create a multi-task learning environment.

The Project Description chapter will introduce and describe our proposed solution to identifying tasks, the Task Identification During Encounters, TIDE, system. We will first present what has been implemented and used to solve the simple Multi-task Multi-armed Bandit problem, and then describe a possible extention of TIDE to a broader range of reinforcement learning problems.

In the Experiments and Results chapter we will in detail describe the multi-task reinforcement learning environment, Multi-task Multi-armed Bandits, and their variations. The experimental setup for testing our system will be explained and the results of the experiments will be presented. Following this we will analyse the results and discuss some of their implications of the TIDE system.

Finally, the conclusion chapter will summarise our work, discuss some of the particularities of the thesis and discuss future work.

Chapter 2

Background

Trying to emulate humans' thinking is a hard endeavor. Making incremental progress is easier when reducing it to solving a series of sub-problems. A general artificial intelligence would have to be able to (1) learn efficiently and (2) simultaneously master multiple domains and tasks.

2.1 Machine Learning

Solving a problem with a machine usually involves creating an explicit algorithm that will solve the problem. For example defining if a student should pass their exam given their score as an input, and a rule that determines exactly where the threshold lies.

Machine learning does not explicitly define the rules for solving the task, but defines the rules for attempting to *learn* the solution to the task. When trying to learn the solution of a task, machine learning uses three main approaches. Supervised, unsupervised and reinforcement learning. Supervised learning is used when the ground truth for each example is known when training. In some examples this is a regression problem that can be solved by gradient descent. More complicated applications with high dimensionality, such as object detection and image recognition, are less easily solved by regression. Machine learning approaches have outperformed humans in some of these tasks [8] on some data sets.

Unsupervised learning is used when the ground truth label is not available. This approach clusters data together in some pattern, such that similar data points are grouped together.

Reinforcement learning is used when the ground truth label is not available, but there is some way of achieving a partially correct answer and receive some reward/punishment for this. Reinforcement learning has achieved great success on tasks with explicit reward structures such as games with score [18].

Artificial Neural Networks, ANNs, are popularly used in machine learning, as their many weights and non-linear activation functions allow them to (theoretically) solve most problems. Training the weights for an artificial neural network is done by

some optimisation procedure, stochastic gradient descent and ADAM [10] are popular choices.

2.1.1 Human inspired intelligence

Many approaches to achieve learning take heavy inspiration from humans and biology. While ANNs take inspiration from the brain by having nodes (neurons) and connections between those nodes to achieve learning, this seems to be insufficient to *learn to learn* as humans are able to. As shown by Elman [4], when evolving an architecture and their corresponding weights to learn to control a body it is important to evolve incrementally. Starting with a small network, training and then evolving the architecture sufficiently slowly for the training to take effect on each evolved architecture is important. This is in line with how the human brain evolves. The effective control of body and brain might be contingent on development of the two in tandem.

Human development is marked by very slow processes. Newborns are almost completely defenseless in comparison to many other animals, but grow to be able to do more than all other animals. This slow acquiring of knowledge over many domains might be important for *learning to learn* and generating a structure that is able to solve multiple tasks without deteriorating performance on others. In light of this, creating a method that can accumulate knowledge over long periods of time without losing old knowledge should help to bring the field closer to a general artificial intelligence.

2.2 Multi-armed Bandits

Multi-armed bandits are a simple environment that is often used in reinforcement learning. In the experiments in chapter 4, a **multi-task** multi-armed bandit problem is used. Different variations of multi-armed bandits are described below, including the multi-task multi-armed bandit.

2.2.1 Stationary Multi-armed Bandit problem

A bandit is a collection of arms that the agent can pull. Each pull of an arm results in a positive reward with some fixed probability. The probability for each arm is often decided to be high for one arm and much lower for all others. Figure 2.1 shows this problem. The multi-armed bandit is never changed, thus it is called the *stationary* multi-armed bandit problem.

The agent only observes the reward, and must learn which arm has the greatest expected reward by trial and error. Solving this problem requires some degree of exploration to adequately get a value for each arm, and exploitation to stop pulling inferior arms as early as possible to maximise reward.

Solutions to this problem include the Upper Confidence Bound [12] algorithm.

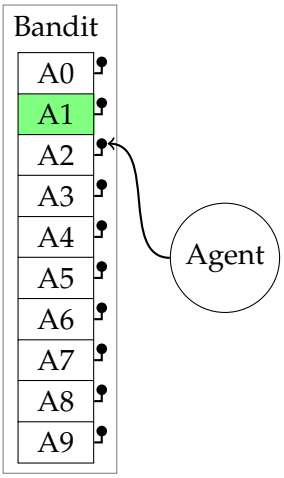


Figure 2.1: The Multi-Armed Bandit problem, A0-A9 are the arms of the bandit. A1 has higher probability of reward. Agent pulls A2 arm

2.2.2 The Upper Confidence Bound algorithm

To solve the Stationary Multi-arm Bandit problem, the Upper Confidence Bound, UCB, solution has guarantees for being able to find the most rewarding arm in a stationary environment. This section describes this algorithm. It will be used as a baseline and as a component in several of the agents in 4.2.7. It will be used as a component in the Distributed-UCB and Table-UCB in section 4.2.7.

UCB stores each arm it has selected and what reward it got from selecting that arm. UCB calculates a reward estimate for each arm by equation 2.1. This is simply the average reward from selecting that arm in the past.

$$r_t(a) = \frac{\sum_{i=0}^{t-1} r_i(a)}{t} \tag{2.1}$$

where $r_t(a)$ is the reward estimate for selecting arm a at time step t.

Choosing the arm with the maximum reward estimate would be the pure exploitation approach. In addition to this, UCB uses an exploration strategy. The upper confidence bound exploration term is shown in equation 2.2.

$$d_t(a) = \sqrt{2 \cdot \frac{\ln(t+1)}{N_t(a)}} \tag{2.2}$$

where $d_t(a)$ is the uncertainty for the reward estimate for selecting arm a at time step t. $N_t(a)$ is the total number of times arm a has been selected at time step t.

The arm which maximises the sum of reward estimate and exploration term is chosen at each time step as shown in equation 2.3.

$$A_t = argmax_a(r_t(a) + d_t(a)) \tag{2.3}$$

where A_t is the chosen arm at time step t.

This exploration-exploitation trade-off is very effective at solving the stationary multi-armed bandit problem. In only a few dozen experiences, the UCB algorithm finds the most rewarding arm in a stationary multi-armed bandit problem and continues picking that arm.

2.2.3 Multi-task Multi-armed Bandits problem

In the stationary multi-armed bandit problem, the reward structure behind the arms never changes. Introducing change to the problem, multiple stationary multi-armed bandits are shown to the player one at a time. Extending the multi-armed bandits problem to the multi-task multi-armed bandits problem includes adding task descriptors to each of the bandits. This problem is used in the experiments in chapter 4. Often described as contextual bandits, as the player is presented with a context for each experience with the bandit. This context is a task descriptor in the experiments in this thesis. A task descriptor can generally be anything, such as a picture, vector of floating point numbers or natural language text. The task descriptor is a string of characters in the experiments we perform. In this environment several multi-armed bandits exist, and each bandit has a fixed probability of giving reward for each arm. A single bandit is presented to the player for some number of experiences, and then another is drawn to replace the previous bandit. The selection of the next bandit to present is done by drawing uniformly from an urn of bandits. All the bandits that exist in the environment are duplicated 10 times and added to the urn. The urn is drawn from with replacement. There is no guarantee that the task changes when another bandit it drawn, the same bandit might be drawn again.

Generally, the agent is presented with some form of observation for each experience, and this observation is affected by the drawing of a new bandit. To solve this task, the agent must either have some way to sense that the task has changed from the observation or be explicitly told that the task has changed.

For the specific problems in this thesis, the observation given for each bandit is their task descriptor. The structure of these task descriptors will be described in section 4.2.5. The bandits used in the experiments will have 10 arms and pulling an arm will give reward with a probability of 0.1 for nine of the arms, and give reward with a probability of 0.9 for one of the arms. Each configuration of a bandit is defined as a *task*, so two bandits may belong to the same task, but have different task descriptors, as shown in 2.3.

In figure 2.2 we see three bandits with their most rewarding arms marked. The bandits all have a task descriptor and an agent is selecting an arm to pull. The agent cannot see the interior of the bandit, it can only choose which arm to pull.

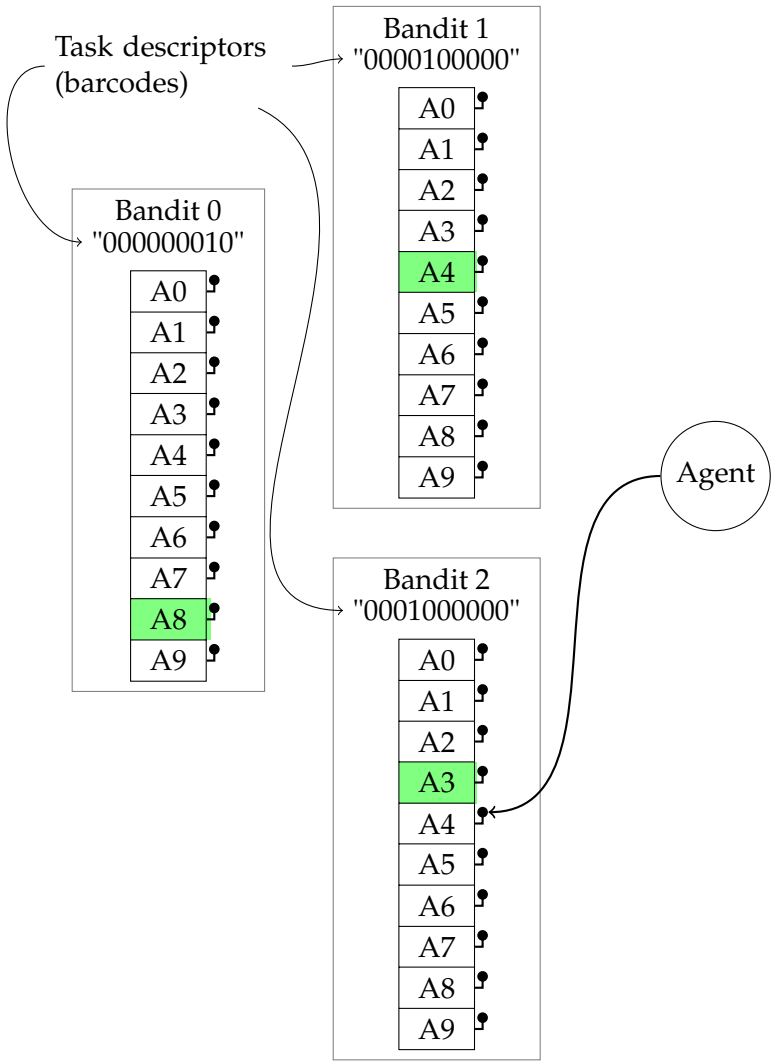


Figure 2.2: The Multi-task Multi-Armed Bandit problem, A0-A9 are the arms of the bandit. Different bandits have different distribution of probabilities of reward. Agent meets Bandit 2 in this instance and pulls A4 arm

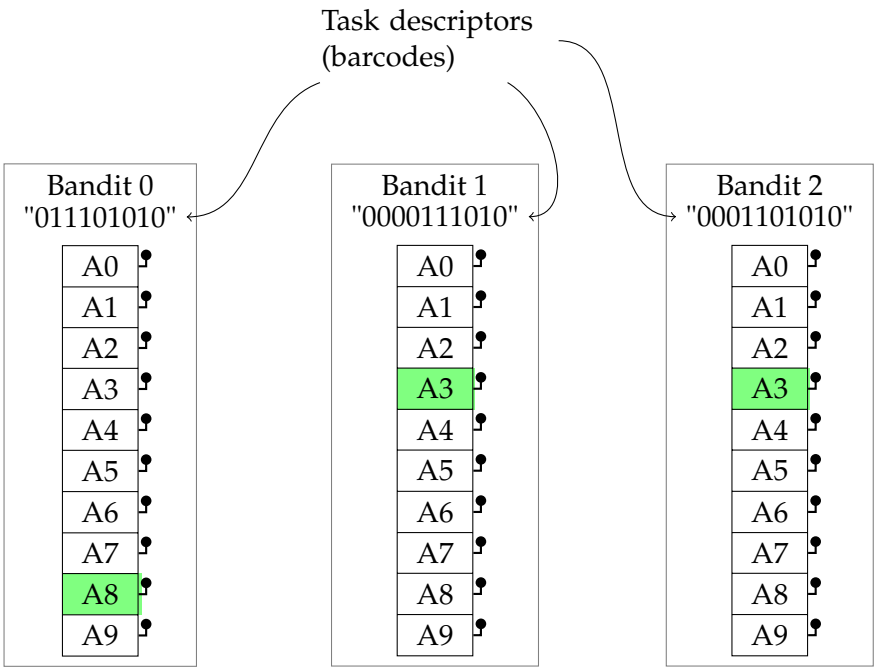


Figure 2.3: Several Multi-task Multi-Armed Bandits, A0-A9 are the arms of the bandit. Different tasks have different distribution of probabilities of reward. Bandit 1 and bandit 2 belong to the same task, with different task descriptors

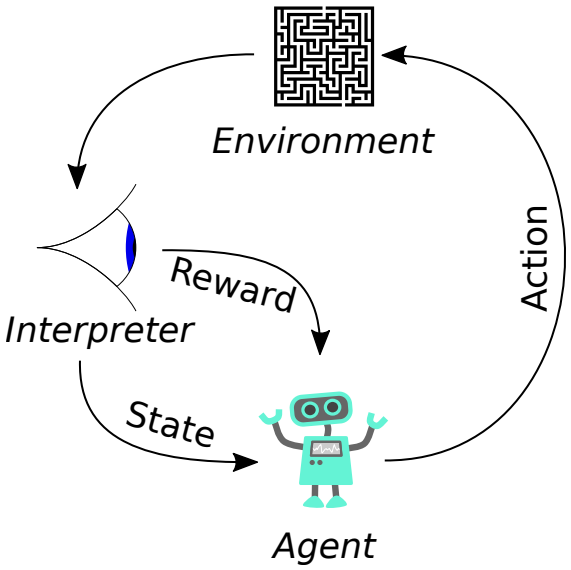


Figure 2.4: General Reinforcement Learning problem [28], an agent only sees the state of the environment and the reward from the last action. It uses this to generate the next action.

2.3 Reinforcement Learning

Many problems that humans face can be described as Reinforcement Learning problems, such as learning to walk, throwing a ball, pronouncing words or playing a game. Games can be simulated quite easily and are often the focus of testing for new reinforcement learning algorithms [3].

In Reinforcement Learning (RL), only the current state of the environment is used to determine the next action taken by the agent as shown in figure 2.4. This means we define the problem as a Markov Decision Process with state set S, action set A and reward function R. Each action made by the agent in the environment results in a positive, negative or zero valued reward. For example gaining score in a game, or the release of chemicals in the brain for humans. The goal is to learn a policy that maps these states to actions such that the total reward gained is as high as possible.

Calculating an expected reward for actions in a given state and then choosing the actions which gives the highest expected reward should result in a policy that maximises reward, given that the expected reward is close to the true rewards.

A way of calculating an expected reward is Q-learning [27]. This creates a Q-value for each action in each state. The Q-value uses the expected reward for the given action as well as an expected future reward.

After training, the Q-value generated by the architecture should be a close approximation to the "real" Q-value, which is the actual reward and possible future reward from that action. By choosing the action that maximises the Q-value at each

choice the algorithm tries to obtain the highest total reward.

2.3.1 Deep Reinforcement learning

In deep reinforcement learning, changes in the network weights are small for each sample, which leads to a need for many samples. The size of the changes needs to be small to ensure the network does not overwrite earlier learned knowledge, known as catastrophic inference [17], and maximises generalisation as well as ensuring the weights are not skipping a local or global optima. This incremental parameter adjustment is one source of slowness in deep reinforcement learning.

Another source is the weak inductive bias. If a deep reinforcement learning network is to learn something in few samples, the changes to the weights should in some way get closer to an optimal solution for the given task. The more general this solution is, ie. that it can account for a richer set of samples to solve correctly, the slower the learning must be. The bias-variance trade-off is the trade-off between fast learning and generalization. All deep reinforcement learning algorithms must balance this trade-off.

2.3.2 Learning efficiently

A recurring problem in reinforcement learning is the need for extensive training. When a problem needs extensive training, many millions of instances of a task must be run to learn to solve it [18]. Known as the sample efficiency problem, this reduces the impact of many reinforcement learning results, as many applications don't have enough samples available in reasonable time. For example learning to walk is impressive for a bipedal robot, but if it takes 25 years of training in the real world to gather enough samples to learn properly, the practical use is modest.

Several approaches try to address this. We will focus on episodic learning and meta-reinforcement learning. In summary, episodic learning 'remembers' how to solve a similar task, while meta-learning tries to *learn to learn* how to solve tasks that have some similarities.

These approaches have achieved impressive results on somewhat similar tasks.

2.4 Mastering multiple tasks simultaneously

2.4.1 Catastrophic forgetting

A major challenge in machine learning is learning a new task without reducing the performance of previously mastered tasks.

An artificial neural network, ANN, must update the weights of the network to learn new tasks. Some of these weights may be very important for solving a previously learned task. When the new task is learned, these important weights might be changed to solve the new problem. The previously mastered task is then forgotten. This is called catastrophic forgetting.

2.4.2 Meta-Reinforcement Learning

Multi-task Meta-Reinforcement Learning

Multi-task Meta-learning deals with trying to learn to solve a wide range of tasks that do not have a single general solution, but requires some sort of specialization. Specialization on each specific task negatively affects performance on other tasks. Multi-task Meta-learning tries to create a general starting point from which each task can be solved after some small amount of training.

In essence, it creates a 'good' initialization for its weights for a training regime on all tasks. 'Good' in this case means that it is sample-efficient when learning the specific task it is assigned to solve. The learned weights initialization θ_{start} should easily train to learn each of the specific tasks' optimal weights (θ_1, θ_2). Ideally, all tasks the agent is going to meet is solvable by performing a few gradient steps from θ_{start}.

2.4.3 Deep Meta-Reinforcement Learning

Trying to speed up the process of learning, Deep Meta-Reinforcement Learning [26] uses reinforcement learning to train a recurrent neural network (RNN) which in turn implements a reinforcement learning procedure.

A meta-learning approach usually has two parts, a part that learns over a short timescale and a part that learns over a long timescale. By using these to affect each other, the concept is that the slower part should learn aspects of the domain that persists over different subtasks.

2.4.4 Episodic Learning

Humans use their memories to influence their understanding of the tasks they are trying to understand or solve [7, 15]. Similar interactions or experiences have a substantial effect on the perception of a new interaction or experience. Episodic Learning [20] tries to harness this in machine learning. Using this, Pritzel et al. obtain substantially better sample efficiency than previous deep learning approaches. They show their results on Atari games at 10 million frames of learning.

The authors present the Differentiable Neural Dictionary, DND, as a memory module. The DND is used to steer learning within a single Atari game. The model is reset between different tasks, so there is no meta-learning. Differentiating the tasks is done outside the algorithm.

By encoding inputs as a representation within the neural network and assigning each input encoding a reward value, one can store previous experiences for future use. Comparing a new input encoding to the stored encodings and using a portion of the reward values stored to influence future actions will essentially enable a single encounter with a task to affect future learning.

2.4.5 Episodic Meta-Reinforcement Learning

Combining episodic learning and meta-reinforcement learning, Ritter et al. [22] achieve some success on simple problems that differ somewhat, differentiated by an identifier supplied to the learning algorithm.

2.5 Learning continually

In contrast to many machine learning approaches, continual learning deals with a more realistic perspective on learning, in relation to how a human learns. Humans don't remember all previous examples, examples are given from a temporally structured sequence and from different tasks. In addition, an example is unlikely to appear twice. Trying to replicate these conditions introduces many new challenges.

2.5.1 Continual learning in supervised learning

In supervised learning, a single experience is often represented as a tuple, consisting of a feature vector x_i and a target vector y_i. Creating a predictor that maps feature vectors to target vectors assumes that all feature vectors map to a target vector over the same distribution.

If there exists some structure such that a given feature vector should map to different target vectors depending on what task the feature vector is taken from, the predictor must in some way be aware of this difference. The predictor can be made aware of this by explicitly supplying the task description for each example, or the task can be inferred from the sequence of previous experiences. It is more challenging to infer the task than to have it supplied. Humans are able to do both these things. Many challenges that a human must overcome is preceded by a description of the way to evaluate success at least. Natural language sentences can be used to inform a human of what a task is, and it is possible to do the same for machine learning.

Gradient Episodic Memory (GEM) [14] show an approach to continual learning in the supervised learning framework. Each experience in this continual learning framework is shown in sequence, example by example, and the algorithm should predict the target vector from the feature vector and a task descriptor. GEM allocates a small number of memories for each task in the set of tasks. GEM makes sure that a weight update of the network does not negatively affect the memories that are stored for the current task. By doing this they propose that negative backward transfer is lowered. Because the memory storage is limited, the algorithm cannot guarantee negative backward transfer further back in time than the capacity of the memory storage.

2.6 Related Work

Combining episodic learning and meta-reinforcement learning, Ritter et al. [22] achieve some success on simple problems that differ somewhat. The problems are identified by a barcode. The authors point out that the identification of a task from barcodes will not always be possible, and in real-world problems identifying the nature of a task is often very difficult.

Ritter et al. [22] performed experiments with task recognition as the main theme. They tested their algorithm in three environments, multi-armed bandits, a water maze navigation task and a task from the neuroscience literature. Only the multi-armed bandit experiments are relevant to the experiments in this thesis.

If the model has encountered a task previously, a previously saved internal state will be reinstated, and used to continue completing that task. Ritter et al. aim to overcome catastrophic forgetting and enable further learning of the task by using Differentiable Neural Dictionary, DND, which stores embeddings of task contexts as keys and Long Short Term Memory, LSTM, cell states as values. Thus it uses the task context as keys and the internal state of the algorithm as values. It performs k-nearest-neighbours lookup in this DND to find the identity of a task.

A similar approach was used in a classical reinforcement learning benchmark. Pritzel et al. [20] stored convolutional embeddings of the game pixels as keys and state-action value estimates as values. They used this approach on 57 Atari games with state-of-the-art sample efficiency.

2.6.1 Meta Reinforcement Learning As Task Inference

Using experts to bootstrap learning of tasks is a venture that many have made. Using a belief network and a meta-learning network Humplik et al. [9] achieve good performance on continuous control environments requiring long-term memory. The belief network is responsible for creating a task description and should predict the correct task labels. This network does in essence the same thing as the system I am proposing, for a continuous environment. Many belief networks are trained unsupervised as the structure of the tasks is unknown. This paper proposes that learning the task structure unsupervised is in general unsolved. Using the fact that many meta-RL scenarios are created with access to the design of the underlying task structure, the authors use this knowledge to train their network. At test time they do not rely on any additional knowledge.

The paper uses three main components to solve environments: 1. Task description: The belief network directly predicts the true task description from the current trajectory. 2. Expert actions: Each training task has an expert agent that has only been trained on that task. 3. Task embeddings: There is a fixed and finite number K of tasks in the environment. The belief network then only predicts an index into this task embedding.

Training of the belief network is done in a supervised way. The network is trained using the trajectory to predict the task label and receiving the true labels from task structure knowledge.

The system uses a policy network, value network and belief network. The belief network uses a Multi Layer Perceptron, MLP, connected to a LSTM which is then connected to another MLP. The belief network creates the belief features which are fed to the policy and value networks along with the original observation, action and reward features. The policy and value networks use a MLP and a Linear layer to create the values and policies. From this the best action is selected.

2.6.2 Learning Context-aware Task Reasoning for Efficient Meta-reinforcement Learning

Trying to solve meta-learning tasks is a difficult endeavor. A good solution to solving complex challenges is often to partition the challenge into smaller challenges. Wang et al. [25] divide meta-RL into three components: Task-exploration, task-inference and task-solving.

The task-exploration module can interact with the environment to create multiple experiences that are used to train the task-inference module. The task-inference module in turn gives the task-solving module task information. The approach boasts of very good results compared to other solutions for meta-RL tasks.

2.6.3 Continuous Meta-Learning Without Tasks

Starting from the observation that many meta-learning approaches ignore the need for identifying a task at runtime, instead giving the algorithms knowledge of the task directly, Harrison et al. [6] tries to identify when the task has changed, without using task reoccurence. Their approach does not leverage earlier knowledge of the specific task. This is evident when a task switch occurs without their algorithm classifying it as a task switch. The new task is able to be sufficiently solved by the current formulation. The task switch is a small change, which could be classified as a very similar task instead of a new one. If a task reoccurs, the model does not leverage earlier knowledge of it. Leveraging earlier knowledge would of course require some form of memory of previous task solutions.

In the problem statement, a regression problem has a probability of changing parameters at each time step. This probability is called the hazard rate.

The framework used is based on Bayesian Online Changepoint Detection [1], which detects a change in time-series data. This is done by generating a belief distribution over how many of the last data points belong to a single distribution (belong to the same task).

The algorithm used retains a run-length belief of how likely it is that the current distribution started r_t time steps ago, and will continue for the next time step. This corresponds to having a flexible window length where the window length determines how long the agent should assume the distribution that inputs are drawn from is the same (how long the same task is repeated before changing).

Leveraging this belief when the agent sees the input x_t and after receiving the correct label y_t, it determines if the task has changed. Harrison et al. show promising

results in their regression problem. The results for the classification problems were less impressive, but still better, than using a model with constant window length.

2.6.4 Continuous Adaptation via Meta-Learning in Nonstationary and Competitive Environments

Al-Shedivat et al. [23] assume that there is a direct link from the previous task to the current one. The expectation for this task is directly dependent on the previous one. They use a nonstationary environment, which is an environment that has some radical changes that affects how an agent should go about solving it.

Their approach frames a nonstationary environment as a sequence of stationary tasks. This makes it possible to tackle a nonstationary environment as a multi-task learning problem. The authors uses Model-Agnostic Meta-Learning [5], MAML, and assumes a distribution of tasks with a solution trajectory for each one. Given a series of interactions with a new task, a new trajectory should be constructed to solve this new task.

These sequentially dependent tasks are then solved by incrementally updating a meta-learned solution to the environment. Their approach uses a Markov chain to emulate the tasks and their transitions. There is no explicit reuse of previous tasks, and their identification is similar to Bayesian Online Changepoint Detection which only switches tasks, never remembering if it has seen it before.

2.6.5 Continual learning

Nagabandi et al. [19] uses an algorithm that chooses whether to use a previously created model to solve a task or instantiate a new model. Using meta-learning and online learning, which trains on each experience once before discarding it, they achieve impressive results on matching tasks with a model without creating many more models than needed. Tasks are created in the MuJoCo environment. Both training and evaluation is done in an environment that changes at specific times. When it changes, it becomes a new task.

2.6.6 Overcoming catastrophic forgetting

Kirkpatrick et al. [11] present a method that slows down learning on weights that are important for a given task as it learns it. Thereby "freezing" these important weights for the given task. As the architecture learns a new task it is prevented from affecting these "frozen" weights. The authors describe the method as scalable and effective at learning several Atari games sequentially.

Masse et al. [16] present a different method for alleviating catastrophic forgetting by protecting most of the network from changes. This method uses a context-dependent gating signal to ensure only a subset of the available nodes in the network are active. In this way it limits the "complexity" of the network for each specific task, so that the complete network is a conglomeration of multiple subnets that each can

solve a single task or a small number of tasks. The authors propose that this is a complementary method to other approaches and can be combined easily. Their tests on the permuted MNIST digit classification task show that a base ANN can classify digits with a mean classification accuracy of 98.5% for any single permutation, drops to 52.5% when training of 10 permutations and 19.1% when training on 100 permutations. Their implementation learned 100 sequentially trained tasks with minimal loss in performance, dropping from 98.2% on the first task to a mean of 95.4% across all 100 tasks.

2.7 Approaches to thinking fast and slow

Humans are extraordinarily good at seeing connections, and often multiple different explanations for the same phenomena can seem quite logical. By using the abstraction of thinking fast and slow, multiple papers have proposed that their algorithms in some way mimic human thinking.

Some works [13] define the fast system as a computationally faster system than the slow system, while others [2] focus on the differing workings of the systems.

2.7.1 Expert Iteration

Emulating humans' ability to "think fast and slow" is the subject of Expert Iteration [2], which uses a deep learning algorithm as the fast system and a tree search algorithm as the slow system. The deep learning system is used as a way to generalise plans, while the tree search is used to discover plans. The slow system is somewhat similar to a tree search algorithm with heuristics to prune the search tree. In many implementations of tree search with heuristics the heuristics are in some form human-informed. Expert Iteration removes the process of creating heuristics manually.

The tree search algorithm is used to evaluate and choose a policy. The neural network, trained on the "expert" policy, is used to guide search by giving each node in the tree a specific value. This value is used by the tree search to easily skip subtrees that are most likely suboptimal.

2.7.2 Throwing the ball into the basket

Taking inspiration from Daniel Kahneman's Thinking: Fast and slow, Li et al. [13] take inspiration from the fast and slow systems concept to generate trajectories in a robot to throw a ball into a basket.

Their model of the fast system is described as a linear combination of polynomials of distances from the basket to determine the rotational speed and angle for shooting the ball into the basket. The slow system is a more complicated hand-crafted set of equations with a memory-buffer. The slow system is used if a throw is sufficiently bad, which then affects the fast system.

The authors claim that their experiments verify that the fast and slow thinking mode of humans is reasonable and effective in robots and that they "make the robots process with human-like thinking behavior".

Chapter 3

Implementation

A robotic system that uses the two-system model as described in section 1.2 will need to know whether to use the fast or slow system. Mimicking humans, we will prefer to use the fast system if it is feasible.

Task Identification During Encounters, TIDE, is an online task recognition algorithm used to aid a two system framework with selecting which of the two systems to use. In figure 3.1 the general sequence for a two system model interacting with an environment is shown. Determining which system to use is done by designating a task as known or unknown without memorizing earlier task descriptors explicitly. The proposed system will use a neural network and a set of agents to solve a simple multi-task environment. Defining if the fast or slow system should be invoked is done by identifying which task from a set of known tasks an experience is from, or if it is from an entirely new task.

In this chapter, we will first motivate why using TIDE should help solve multi-task problems in section 3.1. In section 3.2 we will present the structure of TIDE as implemented and evaluated on the Barcode Bandit multi-task environment (see section 2.2.3). The prediction algorithm, learning paradigm and implementation details follow in that section. Section 3.3 will present a possible expansion of TIDE to more general environments and agents.

3.1 Motivation for identifying tasks

All responses of an environment to a robots actions are decided by the dynamics of the environment. This results in a set of responses to actions that are drawn from some distribution. We divide these distribution of responses to actions into tasks. Each unique set of distribution of responses is defined as a task. Events in the world are in some way drawn from a distribution. Each distinct distribution is defined as a task. For example, while driving a car, maneuvering a parking lot and a highway are two distinct tasks.

The rules for solving the parking lot situation are unlike the rules for solving the highway situation. The inputs and outputs might be the same, but the environment will

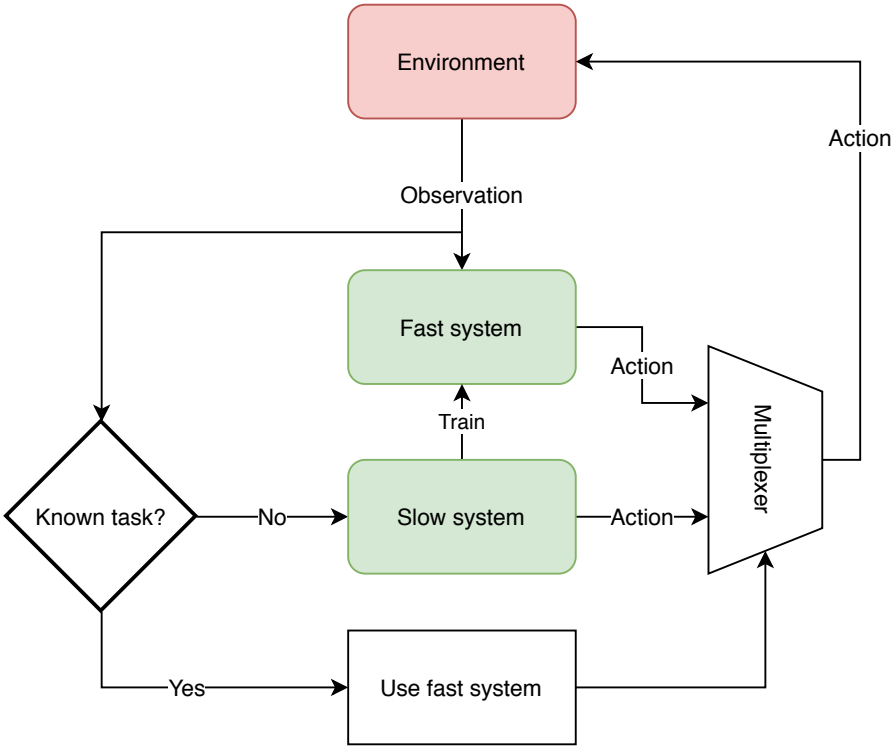


Figure 3.1: Schematic of the two-system model

react very differently to the same actions in the two situations. Even within the parking lot situation, the problem changes with different environmental factors. Solving the problem might be done quite differently if a school has an excursion nearby compared to the entire parking lot being vacant.

The multi-task multi-armed bandit environment presented in 2.2.3 has more in common with the multiple changing conditions of the parking lot than different types of environments (as the parking lot versus highway example). Solving this type of simplified environment is a stepping stone to being able to utilise these principles on a real world environment.

The challenge is then to identify which of the agents in the fast system to use. For this challenge, a simple decider is proposed. By deciding which agent to use, it effectively labels tasks as able to be solved by the fast system, for that task label. Inaccuracy in this decision is catastrophic as the agents will perform badly at any other task than their specialization.

3.2 TIDE: Task Identification During Encounters

At each timestep, an experience is had by the system. The interactions between the environment and TIDE is shown in figure 3.2. An experience for the TIDE system consists of being presented with a task descriptor, predicting which task label it corresponds to and thereby which agent should solve it. Receiving the task descriptor, TIDE chooses which agent to use with a single layer neural network.

The agent, selected by TIDE, chooses which action to take for this experience. The environment returns a reward for the action. TIDE uses this reward to generate a prediction of the correct task label for the given task descriptor. No previous experiences are used for training. Each experience is used to train and then discarded, the memory of the system lies in the weights of the neural network. In the end, TIDE should correctly predict which task descriptors correspond to which task labels.

Ideally, TIDE should be able to identify all the tasks that it is exposed to while invoking the slow system as few times as possible. Invoking the slow system is done by defining the current task as an unknown task. When the slow system is invoked, an agent is created based on the current task.

3.2.1 Details

Selecting which task label to apply for each task description is done using a single layer neural network as shown in figure 3.3. The input is the barcode preprocessed to a vector of length `barcode_length` where each entry is an integer with the value 0 or 1. The output of the neural network has `num_tasks` outputs. Output from the neural network is a probability vector over each of the possible labels. The label with highest probability is chosen, which selects the agent corresponding to that label. This agent selects which arm to pull according to its policy (described in section 4.2.7) and the reward is recorded.

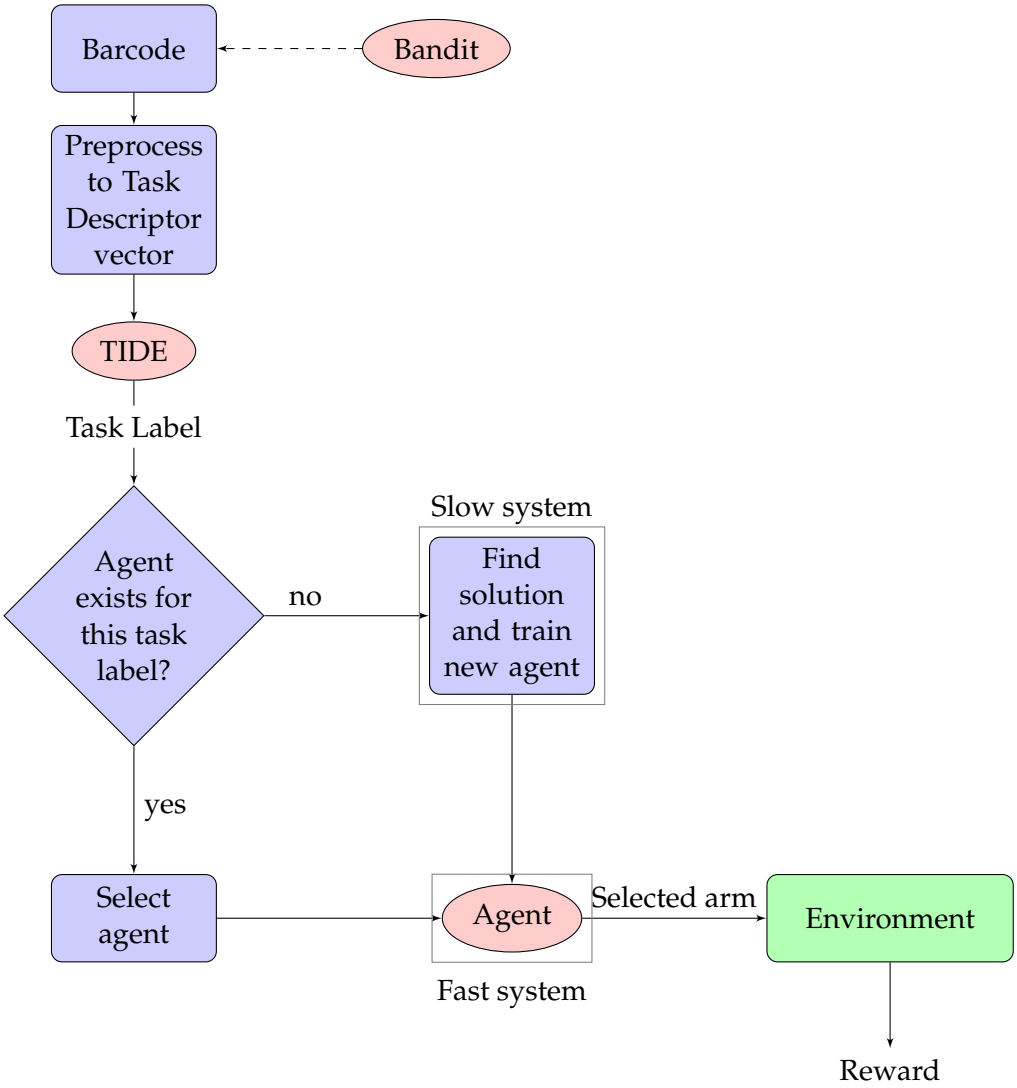


Figure 3.2: Diagram of the environment and interactions with the TIDE system in the Bandit environment

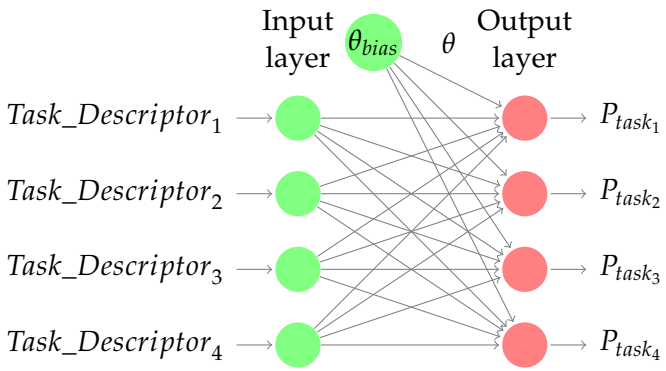


Figure 3.3: Architecture of TIDE. For simplification, in this figure, the number of tasks and number of task descriptor units is 4

The number of input nodes is decided by the length of the task descriptors, and the number of output nodes is decided by how many tasks there are. The length of the task descriptors (`barcode_length`) will be 10 or 120, while the number of tasks (`num_tasks`) will be 10 for all the experiments performed in 4.

3.2.2 Assumptions

It is assumed that there exists some previously trained fast and slow systems that have the following prerequisites:

- The slow system can solve all possible tasks presented without more interaction with the environment.
- The fast system is able to be trained by the slow system without needing to interact more with the environment.

In our implementation of TIDE-Oracle, when the fast system is instantiated for each task label it is given the correct solution. As long as it then assigned to the correct task label, it will always complete the task correctly.

3.2.3 Learning

Throughout the trials, TIDE must evaluate if the current task is represented by some solution agent or a new agent needs to be created. When a task descriptor is presented, TIDE tries to map it to an agent, if this agent does not solve the task sufficiently over the course of the trial, TIDE must re-map this task descriptor to another solution agent. This is done by optimization (using ADAM) of a neural network with a mean squared error as the loss function. The true task labels are created based on which experimental setting it is in. The experimental settings are the supervised learning setting and the reinforcement learning setting described below.

Supervised learning

Learning from supervised training is performed by receiving the true labels from the environment and training once for the given task descriptor for each experience. The training is done after selecting the task label.

Reinforcement learning

After receiving reward for an experience, learning is performed by approximating the true task labels for the experience. An exploration-exploitation paradigm is used to choose which of the other task labels can be selected as correct for the current experience. If the experience resulted in reward, no exploration is done. Exploration is done if the preceding experience does not result in reward.

Approximating the true task labels is done by using the choice, the probability for each task label from the neural network and the reward given for the action selected by the agent. The procedure is shown in algorithm 1 and described below.

Algorithm 1 Creating the correct labeling

procedure (a, r) ▷ a - chosen agent index, ▷ r - reward for experience [0,1]
  $y_{true}[0 \dots n_{tl}] \leftarrow 0$ ▷ Set all task labels to zero
  for all $Tasklabels$ **do**
    if $\epsilon > Randomnumber$ **then**
      $y_{true}[i] \leftarrow 1 - r$
    end if
  end for
  $y_{true}[a] \leftarrow prob_a * r$ ▷ probability of choosing a for the given task descriptor
  Train with x as input and y_{true} as true labels
  return
end procedure

If the selected action gave reward, the task label that was chosen is deemed correct for that task description. All other task labels are set to be 0, while the correct task label is set to the probability from the output of the neural network.

If the selected action did not give reward, the task label is deemed incorrect and the "correct labeling" will be selected differently. All task labels are initially set to 0 and each task label is then set to 1 with a probability equal to the epsilon value (ϵ in algorithm 1). The epsilon value is decreased through training by decaying from the initial value $\epsilon_{initial} = 0.2$ to the minimum value $\epsilon_{min} = 0.05$. The decay is performed by multiplying the epsilon value with the epsilon decay parameter ϵ_{decay} at the end of each time step. This ensures that the neural network explores possible task labels for each task descriptor. This "correct labeling" is used as the true labels in the same supervised learning weight update regime.

3.2.4 Detecting duplicate agents

At the identification of a task label, an agent is instantiated. If TIDE incorrectly identifies the same bandit type (task) as different task labels, multiple agents for the same task are created. Because there is a limited amount of output nodes in the neural network, there will not be room for one of the tasks to be solved. To mitigate this issue, at each labeling of a task descriptor, a check is done to see if the agent instantiated is a duplicate of another. Two agents are duplicates of each other if they will choose the same action for the same input. Marking duplicate agents is shown in algorithm 2. If the agent is a duplicate, it is overwritten if it is younger than the other duplicates.

Algorithm 2 Marking duplicate agents

```
1: procedure (a, agents)                    ▷ a - chosen agent index,
                                            ▷ agents - list of all agents
2:     for all agent in agents do
3:         if agents[a]'s best choice == agent's best choice then
4:             mark agents[a] and agent as duplicates
5:         end if
6:     end for
7:     return
8: end procedure
```

3.3 EATIDE: Environment Agnostic TIDE

This section will describe the workings of Environment Agnostic Task Identification During Encounters (EATIDE) corresponding to figure 3.4. This procedure is not implemented and tested, but is a theoretic design for an extention of TIDE. The algorithms that are used in the fast and slow systems must be able to generate an expected reward for the following procedure to work.

From the observations of an environment and a bank of previous experiences, EATIDE chooses whether to invoke the slow system,and possibly start a training-session for the fast system, or use the fast system.

For every action, we have the observation, reward given from performing that action and the new observation. In the experience bank, each observation-reward tuple from the environment and action-expected-reward tuple from the fast system will be stored. Together with this, the predicted task label and the actual reward for the action that was chosen for that experience is stored. The action-expected-reward tuple from the slow system is also stored if it was invoked.

A similarity measure for the experiences is the difference in activations of the hidden states in the fast system. If the error between expected and actual reward for similar experiences is low, the slow system will not be invoked. The task-label is then reinforced if the error from that action is smaller than previously. If the error is larger,

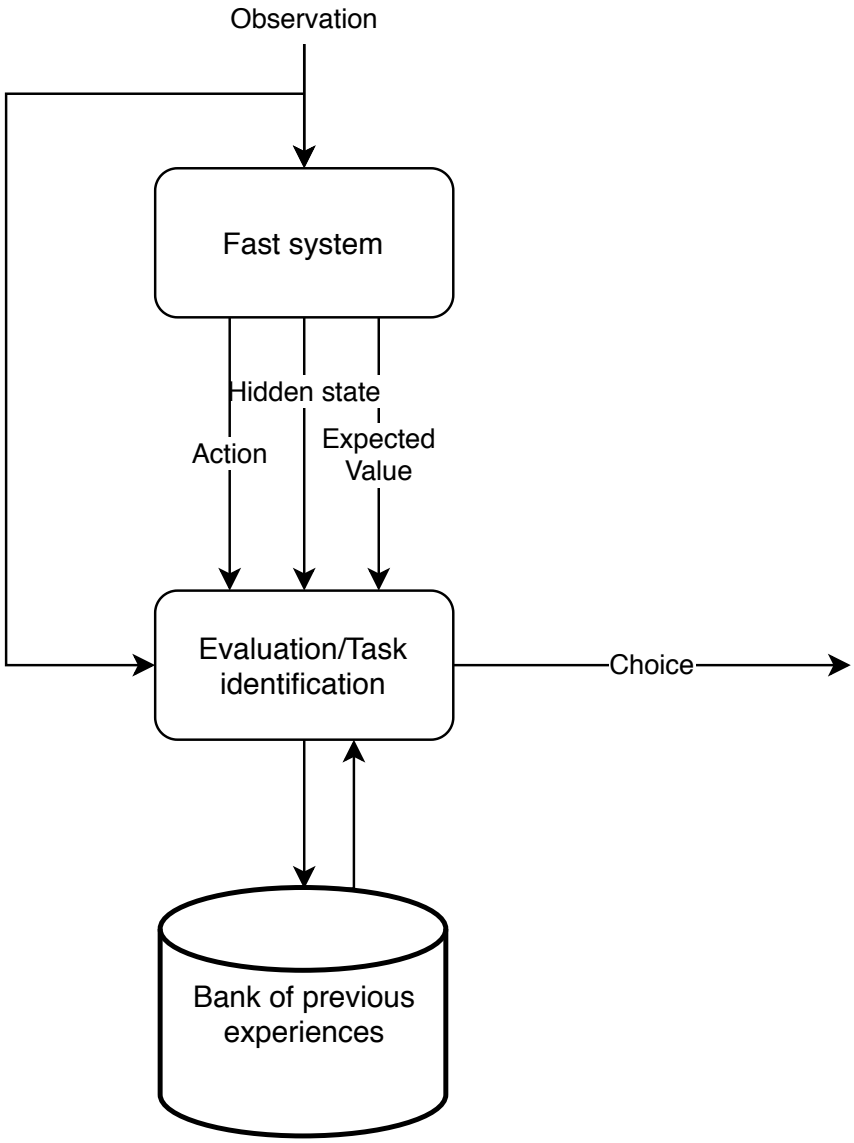


Figure 3.4: Task identification from observation and outputs of the fast system. The bank of previous experiences maps the inputs to a confidence value for a given task id. If the confidence in the action from the fast system for the given task id is too low, the slow system will be invoked

the task-label will either be weakened or the slow system will be invoked next time a similar experience occurs.

When the algorithm adds a new task, it can be thought of as a signal for the slow system to be invoked. When this is invoked, a new instantiation of a fast system should be created and made available to the algorithm.

3.3.1 Task segmentation and evolution

Throughout the trials, TIDE must evaluate if the current task is represented by some solution agent or a new agent needs to be created. When a task descriptor is presented, TIDE tries to map it to an agent, if this agent does not solve the task sufficiently over the course of the trial, TIDE must re-map this task descriptor to another solution agent. This is done by ordinary optimization of a neural network (stochastic gradient descent) with a mean squared error as the loss function. The labeled data is created by setting all task-to-agent predictions to 0 except the one that was selected, which is set to 0 if no reward was given for the episode. If the episode resulted in reward, the correct prediction is the one that was predicted. To enable agent re-mapping, each other prediction can be set to the highest prediction value instead of 0 with a decaying probability over the task-mappings lifetime.

3.3.2 Is the fast system good enough

Throughout a lifetime, EATIDE must evaluate if the fast system is solving something it has a good solution for, or the slow system should be engaged. To evaluate this, the reward given is used to determine if the previous action was a good choice and based on this reward, the probability of the fast system being able to solve this task is updated. If the probability of the fast system being able to solve this task is low, the probability of the slow system being invoked to "create" a new solution that should be incorporated in the fast system is increased.

Chapter 4

Experiments and Results

This chapter will describe the experiments performed with the TIDE system introduced in section 3.2. In section 4.1 we will describe the motivation behind the experiments. Section 4.2 describes the general conditions for the four different experimental settings that will be evaluated.

Using the **multi-task** multi-armed bandit environment (defined in section 2.2.3) as the multi-task environment, the task descriptors are *barcodes*, and the structure of these barcodes is described in detail in section 4.2.5. Different *solution agents* that the TIDE system uses are presented in 4.2.7. Sections 4.3-4.6 will describe the four experiments, present the results of these experiments and discuss some of the results. An overview of the results is then presented in section 4.7. The final section, section 4.8, is discussion of the thesis and some ideas that did not pan out.

4.1 Motivation

Throughout the experiments, TIDE will be tested in increasingly difficult conditions. All the experiments will be done on the TIDE implementation described in section 3.2, the number of unique tasks is 10. The challenge is to correctly label the tasks from their task descriptors. Evaluating the performance of the task identification algorithm is done by analysing the average of received reward from the bandit environment. Following the work of Ritter et al. [22] with "barcode bandits" as a task recognition challenge, the project aims to solve similar experiments with slightly different measures of success.

Experiments 1 and 2 (sections 4.3 and 4.4) will deal with a supervised learning setting. In experiment 3 and 4 (sections 4.5 and 4.6), there will not be any supervision of the learning, only a reward is given. Experiments 2 and 4 (sections 4.4 and 4.4) will increase the task descriptor length from 10 characters to 120. As the experiments progress, the realism of the challenges should increase. Being supplied the true task labels is not always possible. A larger task descriptor space is likely to be closer to real world problems. There are probably more environments without true labels available

Experiment	Optimiser	Initialiser	Learning Rate	Exploration Policy
1-2 Supervised	ADAM	Normal	0.002	N/A
3-4 Reinforcement	ADAM	Normal	0.002	Epsilon-greedy ($\epsilon = 0.2$ decay to $\epsilon_{min} = 0.05$ $\epsilon_{decay} = 0.99$)

Table 4.1: Table with parameters for neural networks

than there are environments with true labels available.

4.2 Description

TIDE will be presented with four different configurations of the environment in these experiments. The two first experiments will explore labeling tasks from small task descriptors and larger task descriptors, with access to the true task labels. The last two experiments will not have access to the true task labels.

4.2.1 Shared configurations

In all the experiments, the algorithm knows the number of task labels that exist. The number of tasks is 10 and the number of arms on each bandit is 10. The architecture has the number of task labels as the number of output nodes (as shown in figure 3.3). This means that there are 10 output nodes for the TIDE system in all experiments. The number of input nodes is variable according to the length of the task descriptors used. In experiments 1 and 3 there are 10 input nodes, while in experiments 2 and 4 there are 120 input nodes. Tasks are selected and shown to TIDE for a given number of experiences before switching to another task. The task descriptors are shown at each experience.

4.2.2 Creating the true labels

In experiments 1 and 2, TIDE is able to check whether the agent is chosen correctly. Creating the true labels is done by iterating through all the task labels and their corresponding agents to check which one would select the correct arm. The first agent that selects the correct arm is assigned as the correct task label. If no agents give the correct arm selection, the correct task label is the first task label that does not have an agent assigned to it. All other task labels are assigned as incorrect, and the vector consisting of zeroes and a single one is used as the true labels for the TIDE system.

4.2.3 Parameter choices

Choosing which learning rate to use was done by running a parameter search over a few selected values (0.001, 0.002, 0.005, 0.01, 0.05, 0.1, 0.2) and choosing the one that

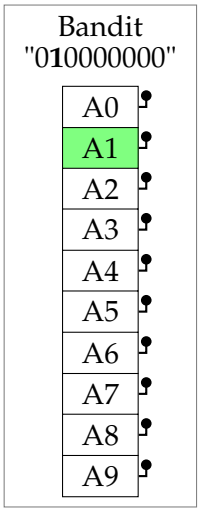


Figure 4.1: A single bandit from the Multi-Task Multi-Armed Bandit problem, A0-A9 are the arms of the bandit. A1 has higher probability of reward. This *descriptive* task descriptor explicitly shows which arm has higher probability of reward. The bolded 1 corresponds to the A1 arm.

converged fastest without using too many reinitializations, invoking the slow system, in the learning. 0.002 was chosen as the learning rate in the ADAM optimisation for these experiments. The epsilon-values used are shown in table 4.1.

4.2.4 Experimental Setup

At the start of each run, a number of bandits are initialised. The bandits are given a barcode with some structure (as described in section 2.2.3). Each episode consists of X experiences with a the same bandit. Each epoch consists of Y episodes. A bandit is drawn to replace the current one at the end of every epoch. X and Y are decided for each type of task switching frequency. For the **frequent** task switching setup, X is 10 and Y is 5000. For the **infrequent** task switching setup, X is 100 and Y is 1000. The bandits are drawn uniformly from an urn containing the set of bandits that exist, where each bandit is duplicated 10 times. Each bandit has `num_arms` arms with 0.1 probability of giving reward except one which has 0.9 probability of giving reward. Reward is 0 or 1.

Below, the different variations of a multi-task multi-arm bandit that we will use is described.

4.2.5 Barcode task descriptors

In the experiments, a multi-task multi-arm bandit problem is created with a unique string of characters as a task descriptor. An example of a bandit with a unique *descriptive* barcode is shown in figure 4.1.

The strings are in all experiments guaranteed to be unique, in that each string will correspond to exactly one bandit. A check is performed so that no strings are equal in an experiment.

Unique descriptive task descriptor

This *descriptive* task descriptor identifies the most rewarding arm in the bandit in figure 4.1. This would make the problem trivial if this information was used to solve the problem. This task descriptor is only used to map the task descriptor to a specific agent that chooses which arm to pull. In this setting, a working task identifier could simply map input to output to choose which of the possible agents to engage. We use 10 bandits.

Unique correlated task descriptor

This *correlated* task descriptor has some structure that relates to the properties of the arms of the bandit. 'Similar' task descriptors should map to 'similar' tasks. In the experimental setup, these *correlated* task descriptors are created by setting the characters at the index of and to each side of the most rewarding arm to one. For example, a bandit with barcode: "0111000000" has the third arm as its most rewarding. "1110000000" would mean that the second arm is the most rewarding. We use 10 bandits.

Unique random task descriptor

In the unique random experiment setting, the task descriptor is generated randomly as a string containing ones and zeroes. It contains no information about the arms of the bandit. There is no explicit structure in the *random* task descriptors. Identifying these task descriptors are akin to memorising them. An example of such a task descriptor would be "0110111000". The random barcode situations use 500 bandits.

4.2.6 Longer barcodes

To perform more challenging experiments for the algorithms, an increase in the number of barcode characters is used. Below we will describe the extention of the descriptive and correlated task descriptors. We will have barcodes of length 10 and 120 in the experiments. The number of arms on each bandit will still be 10. Therefore, the number of tasks is 10, one task for each position of the most rewarding arm in a bandit.

Extended descriptive task descriptor

The longer version of the *descriptive* barcodes are created by setting one of the characters of the string to 1, while the rest are kept at 0. The characters that can be set to 1 are the index of the most rewarding arm shifted along the string. In this setting "1000000000 0000000000" and "0000000000 1000000000" are two distinct barcodes that should both map to the same task label. We use 10, 60 and 120 bandits.

Extended correlated task descriptor

The longer version of the *correlated* barcodes are created by setting up to three of the characters of the string to 1, while the rest are kept at 0. The characters that can be set to 1 are the index of the most rewarding arm and its two neighbours, shifted along the string. In this setting "1100000000 0000000000" and "0000000001 1100000000" are two distinct barcodes that should both map to the same task label. We use 10, 60 and 120 bandits.

Extended random task descriptor

The longer version of the random task descriptor is generated identically to the normal version, but with a longer string of ones and zeroes. We use 500 bandits.

4.2.7 Solution agents

All solution agents used in the experiments will be described in this section. Some of the agents do not use TIDE, these are: Single UCB and Look-up Table UCB. These two solution agents are used to create a comparison for the TIDE algorithms.

The two solution agents that are used with TIDE are described last. When labeling has been applied, two different approaches have been used to solve the labeled task for TIDE. An oracle instantiation (TIDE-Oracle) and a UCB-algorithm instantiation (TIDE Distributed UCD). The difference between these is that the oracle instantiation chooses a single arm that it always pulls for each agent, while the UCB-algorithm instantiation uses the UCB algorithm to choose which arm to pull for each experience.

Single UCB

As the benchmark, an Upper Confidence Bound (UCB) algorithm will be used. In the figures it will appear as Single UCB. This algorithm is not aware of any task switches. It simply tries to solve the bandit problem by using Upper Confidence Bound at all times. This has good performance after a number of samples after each task switch. When it discovers that it is selecting a suboptimal arm it must explore again and therefore has worse performance for a while.

Look-up Table UCB

The simplest form of task identification is a look-up table that stores all task descriptors and solves each of its entries with the UCB algorithm. Each entry in this table is an UCB algorithm that is initialised for each different task descriptor.

Oracle

An oracle algorithm that simply selects the correct arm on each bandit would reach the maximum possible reward for each experience. The maximum possible reward is

shown in the figures as "Maximum". This is done to show the optimal solution and how close each of the algorithms come to it.

TIDE-Oracle

The TIDE system assigns the barcodes to a task label, this task label indexes into a store of agents. The agents are specialised to select a single arm. As long as this is able to map the barcodes correctly, the correct arm should be chosen each time.

This agent chooses the correct solution as long as it is selected for the same task as it was the first time. When the agent is created, it is allowed to see which arm is the most rewarding and saves that information. This corresponds to using the slow system to find the answer. Whenever it is retrieved from the memory of the TIDE-Oracle system, it will select that arm. Thereby it will generate the correct solution each time it is given a bandit corresponding to the same task. Should TIDE select the wrong agent for the task descriptor, it will select the wrong arm.

TIDE Distributed UCB

TIDE Distributed UCD is an instantiation of the Upper Confidence Bound solution for each task identified. TIDE Distributed UCD is a mixture of TIDE and Look-up Table UCB. TIDE is used to obtain the task label from a barcode, and indexes into a store of agents. The agents that determine what arm to pull uses the UCB algorithm. Each agent will then specialise to the tasks it receives. If TIDE is able to correctly label the tasks, it should outperform Single UCB. If TIDE does not label correctly, it should perform on par with Single UCB. It is expected that this has better performance than TIDE-Oracle if the task identification fails. TIDE-Oracle will always choose the "correct" arm according to the task label, while TIDE Distributed UCD is more easily adapted to a change in task for the same task label.

This solver can repurpose wrongly labeled tasks to choose correctly as long as it is consequently chosen for later task descriptors. Should TIDE select the wrong agent for the task descriptor, it will negatively influence the calculations for the UCB algorithm.

4.3 Experiment 1 - supervised learning with small task descriptors

The three different task descriptor structures as described in 4.2.5 are used to evaluate the algorithms, these are displayed along the top row of each figure.

The task descriptors used in this experiment consists of 10 character long strings of ones and zeroes. TIDE is trained with supervised learning, as it has access to the correct task labels.

This reduces the problem to a supervised learning problem. The correct mapping between task descriptors and task labels is generated as each unique bandit type (task) is encountered.

4.3.1 Results

Running the three types of agents (section 4.2.7) on the three barcode structures (section 4.2.5) with task descriptor (barcode) length of 10 is shown in figures 4.2 and 4.3.

Frequent task switches

We change the active bandit every 10 experiences in the frequently changing task situation. Figure 4.2 shows that TIDE manages to correctly label most of the task descriptors after less than 1000 experiences for the correlated and descriptive barcode-structures. For this setup, TIDE gains more reward than Look-up Table UCB, which has the advantage of a direct mapping of task descriptors to agents.

TIDE Distributed UCD has been omitted from the supervised situation, as it had no meaningful way of having a "Best guess" at the beginning, which led to creating a true label impossible. When using its predicted arm as its best guess, it had the same performance as Single UCB.

As expected, the performance of TIDE on random barcodes is bad. Look-up Table UCB overtakes Single UCB eventually, as the bandits are eventually reused.

Infrequent task switches

We change the active bandit every 100 experiences.

From figure 4.3 we see that TIDE is able to correctly label the tasks for the correlated and descriptive barcode structures after a few thousand experiences. Single UCB performs much better than on the frequently changing tasks situation.

On the random barcode structures, we see that it is more rewarding to ignore the task descriptors and adapt to the new bandit when it changes as Single UCB does. Given enough training, Look-up Table UCB would surpass Single UCB. Both TIDE and Look-up Table UCB has bad performance on the random barcode structure. Look-up table eventually surpasses Single UCB once the task descriptors begin repeating more often.

4.3.2 Analysis

The bad performance on random barcodes is partially due to the randomness of the task descriptor structure, and also due to the large number of distinct task descriptors (the random barcodes situation generates 500 bandits). The randomness in the task descriptor structure makes it more difficult to create a general mapping from task descriptor to task label. However, the biggest factor is the fact that the barcodelength is very small. Memorisation is not possible when the task descriptors are so small. There is too much overlap in the task descriptors for different tasks. When given the true task labels, TIDE-Oracle is able to create a mapping of task descriptos to task labels efficiently. The effect of this is that the correct agent is selected and rewarding actions are performed.

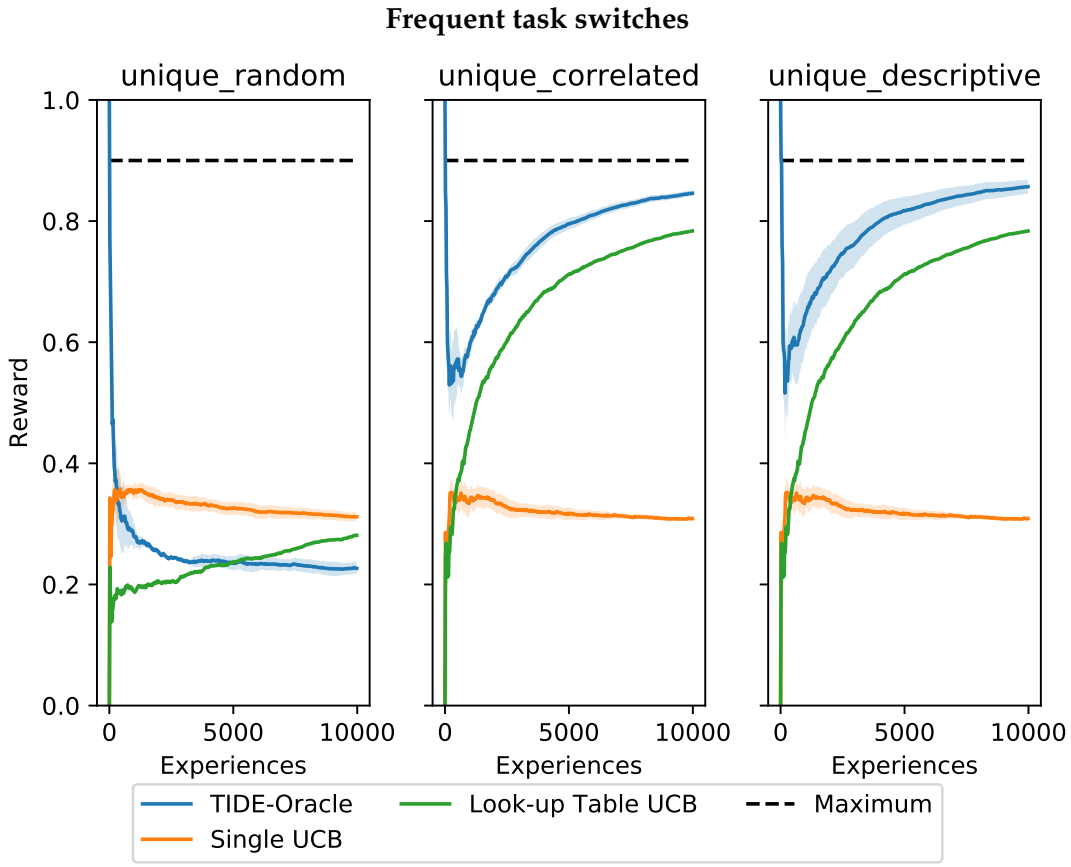


Figure 4.2: Running average reward for **Experiment 1** on three different barcode structures with frequently changing tasks. Task descriptors (barcodes) are 10 characters long. Tasks are changed every 10th experience and are changed 1000 times in total. X-axis shows experiences, Y-axis shows running mean of the reward gained from each experience. Only the first 10000 experiences are shown. The colored area is the confidence interval of one standard deviation.

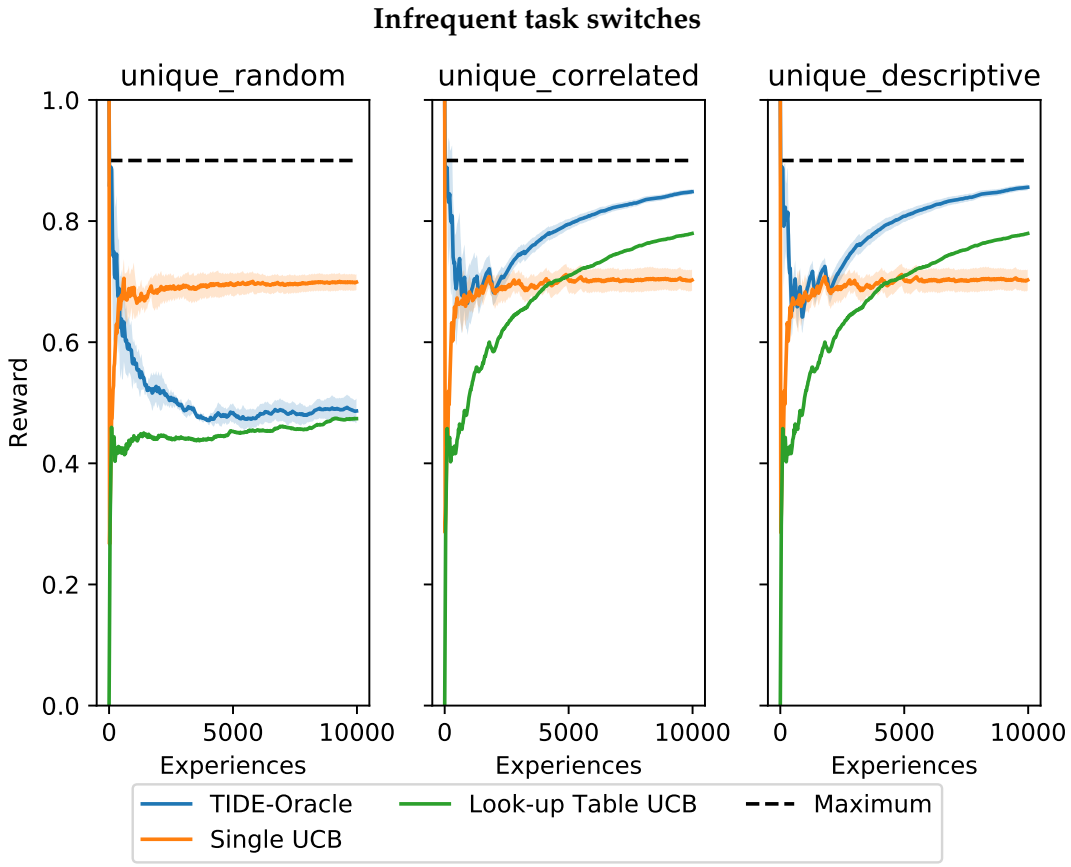


Figure 4.3: Running average reward for **Experiment 1** on three different barcode structures with infrequently changing tasks. Task descriptors (barcodes) are 10 characters long. Tasks are changed every 100th experience and are changed 1000 times in total. X-axis shows experiences, Y-axis shows running mean of the reward gained from each experience. Only the first 10000 experiences are shown. The colored area is the confidence interval of one standard deviation.

4.4 Experiment 2 - supervised learning with larger task descriptors

Increasing the number of characters in the barcode from 10 to 120, we investigate the effect of having a much larger task descriptor. If TIDE is able to generalise, it should have not have substantially more difficulty with this than the shorter task descriptors. Increasing the number of characters in the barcode makes the space of task descriptors much larger. The number of possible combinations increases and the number of weights in the neural network is also increased substantially. Training of the weights should take longer.

We vary the number of bandits from 10 to 60 and 120 in order to investigate what effect this has on performance. The number of arms on each bandit and therefore the number of distinct tasks is kept at 10. When increasing the number of bandits, the algorithm is forced to generalise. As each task can have multiple task descriptors, the algorithms must either memorise all task descriptors or find a general pattern in the task descriptors.

4.4.1 Results

Expanding the number of possible task descriptors to 120 is shown in figures 4.4 and 4.5. The number of bandits is set to 10, 60 and 120 along the rows respectively.

Frequent task switches

We see in figure 4.4 that increasing the number of bandits to label slows down training. TIDE is able to correctly label tasks even when the number of bandits increases. We can also see that Look-up Table UCB gets less reward as the task descriptors increase in size. Single UCB does not get significant reward as the tasks change often.

Infrequent task switches

When the tasks are changed rarely, training takes longer as seen in figure 4.5. The number of experiences needed to meet all the different tasks is increased tenfold when each task is exposed for 10 times as many experiences. Increasing the number of bandits increases training time, as there are more different task descriptors to label. Look-up Table UCB performs worse than TIDE, and lags farther behind as the number of bandits increases. Training initially is more time-consuming for the descriptive task descriptors than the correlated ones for TIDE. Single UCB performs much better on infrequently switching tasks than frequently switching ones.

4.4.2 Analysis

If there are many bandits and high enough dimensionality of the task descriptors, learning the mapping from task descriptor to task takes so long that simply ignoring

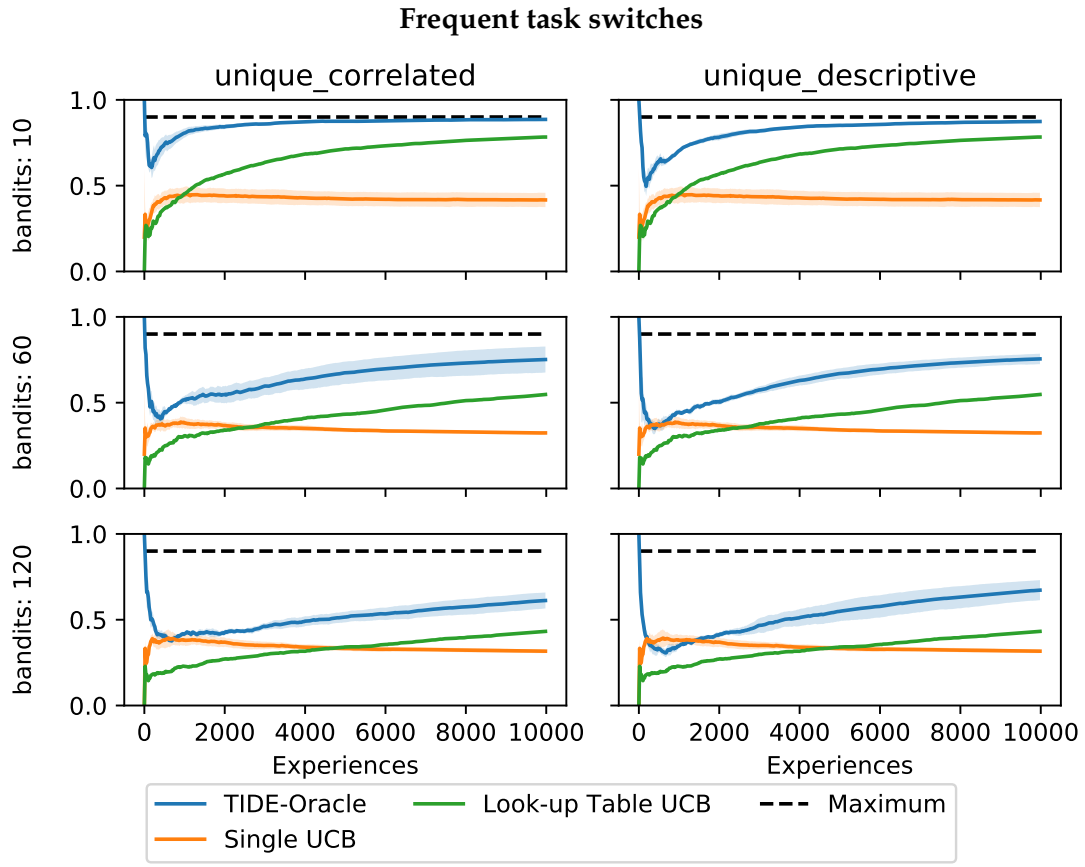


Figure 4.4: Running average reward for **Experiment 2** on two different barcode structures with frequently changing tasks. Task descriptors (barcodes) are 120 characters long. The number of bandits in total are 10, 60 or 120 (row 1, 2 and 3 respectively). Tasks are changed every 10th experience and are changed 5000 times in total. X-axis shows experiences, Y-axis shows running mean of the reward gained from each experience. Only the first 10000 experiences are shown. The colored area is the confidence interval of one standard deviation.

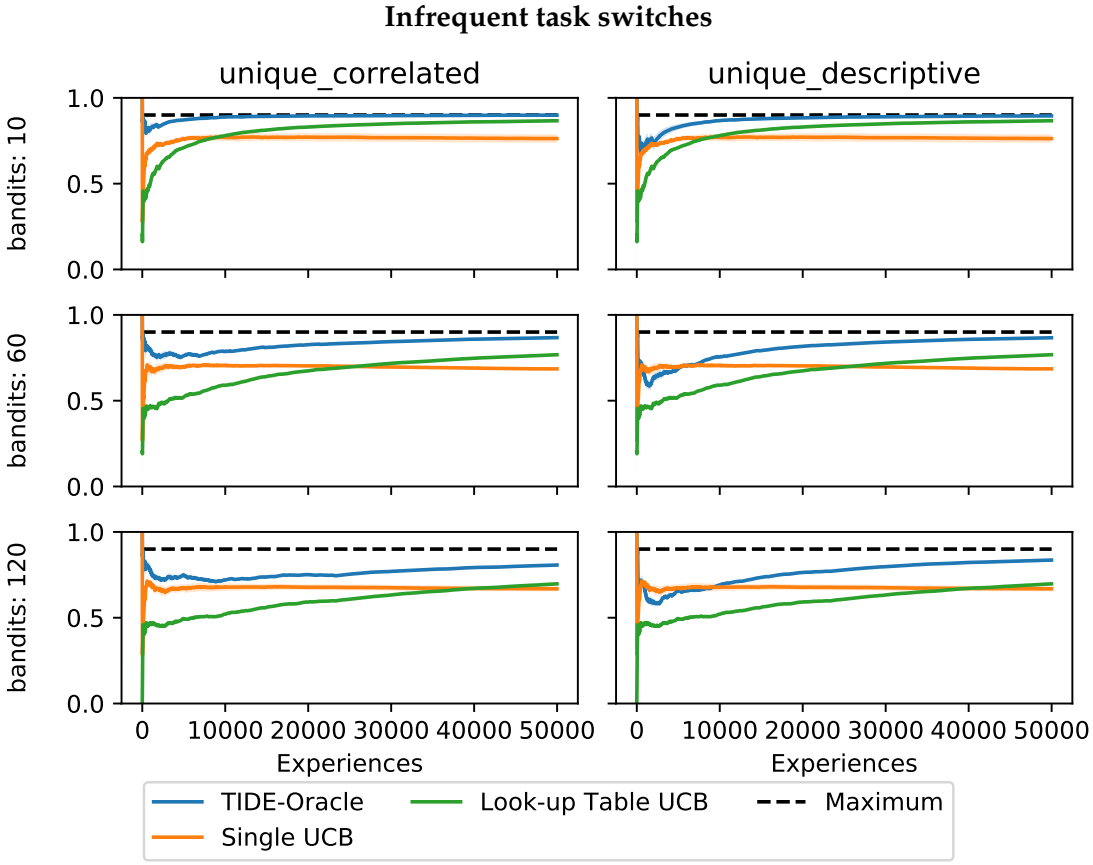


Figure 4.5: Running average reward for **Experiment 2** on two different barcode structures with infrequently changing tasks. Task descriptors (barcodes) are 120 characters long. The number of bandits in total are 10, 60 or 120 (row 1, 2 and 3 respectively). Tasks are changed every 100th experience and are changed 1000 times in total. X-axis shows experiences, Y-axis shows running mean of the reward gained from each experience. Only the first 50000 experiences are shown. The colored area is the confidence interval of one standard deviation.

that there are different tasks is more rewarding for quite a while. However, after sufficient training, it is more rewarding to remember tasks. With this in mind, if there are extremely many different instances of a type of task descriptor and many different tasks it might be more beneficial to learn to adapt instead of remembering previous tasks.

Look-up Table UCB has worse performance on the larger task descriptors because there are more entries in the table, and task solutions are not reused. When increasing the number of bandits, the number of different bandits that correspond to the same task increases. The variance in the order of the bandits increases the probability of meeting a different task after each switch. This forces Single UCB to re-explore often.

Task solution reuse is one the reasons TIDE is more successful than Look-up Table UCB. When a solution has been found to a task, that solution is reused when another task descriptor maps to that task. Identifying that multiple task descriptors all map to the same task enables TIDE to leverage previously learned solutions to solve rarely seen task descriptors.

A situation where the tasks are changed rarely is better suited for adaptation than task reuse in the beginning as we can see in the difference between Single UCB and Look-up Table UCB early in the run.

4.5 Experiment 3 - reinforcement learning with small task descriptors

Changing the setting from experiment 1 to no longer give the algorithm access to the correct task labels, but still able to see what reward the agents are given makes this a reinforcement learning problem. We will generate a substitute for the correct task labels according to the routine described in 3.2.3.

Updating the weights is done solely based on the given reward. Because the environment gives reward stochastically, reward is sometimes given when performing the wrong action. This will result in detrimental learning sometimes, but should in general still converge to the correct task labelings.

4.5.1 Results

Removing the information about the correct task labels given to the algorithm, results of the runs with reinforcement learning with 10 character length barcodes is shown in figures 4.6 and 4.7. TIDE Distributed UCD is added to the algorithms being tested.

Frequent task switches

Look-up Table UCB performs best in the results in 4.6, with either TIDE-Oracle or TIDE Distributed UCD slightly worse for the structured task descriptor situations (correlated and descriptive). On the randomly generated task descriptors, Look-up

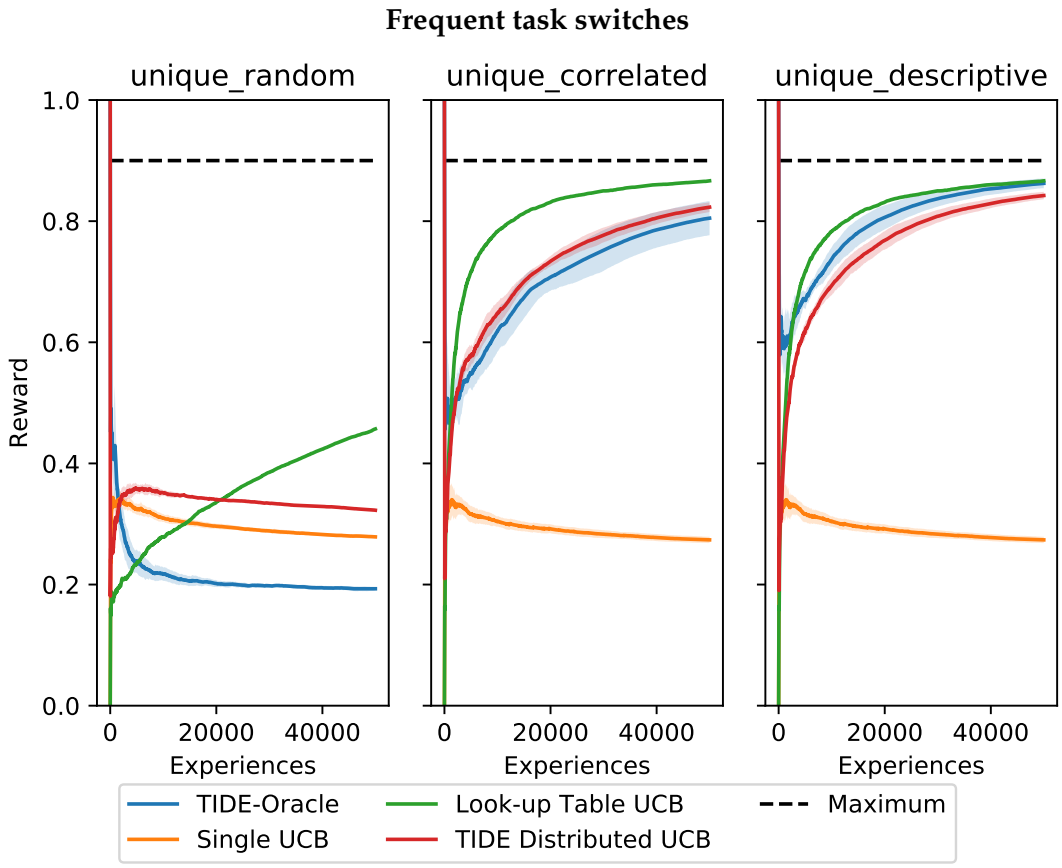


Figure 4.6: Running average reward for **Experiment 3** on three different barcode structures with frequently changing tasks. Task descriptors (barcodes) are 10 characters long. Tasks are changed each 10th experience and are changed 1000 times in total. X-axis shows experiences, Y-axis shows running mean of the reward gained from each experience. The colored area is the confidence interval of one standard deviation

Table UCB performs very well in comparison to the other algorithms. TIDE has very low performance. TIDE Distributed UCD performs slightly better than Single UCB.

Infrequent task switches

TIDE Distributed UCD performs better on infrequent than frequent task switching. After some time, TIDE overtakes TIDE Distributed UCD on the descriptive task descriptors. Look-up Table UCB performs best in the results in 4.7, with either TIDE-Oracle or TIDE Distributed UCD slightly worse for the structured task descriptor situations (correlated and descriptive).

On the randomly generated task descriptors, Look-up Table UCB performs very well in comparison to the other algorithms. TIDE has very low performance.

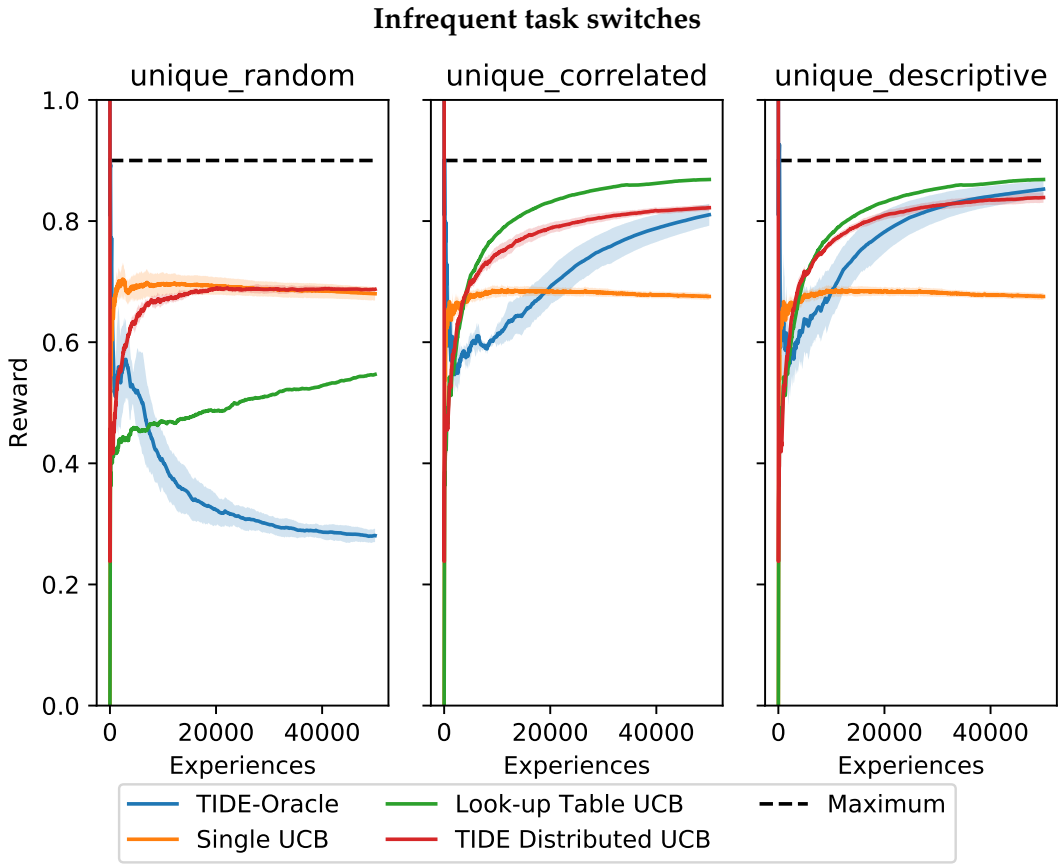


Figure 4.7: Running average reward for **Experiment 3** on three different barcode structures with infrequently changing tasks. Task descriptors (barcodes) are 10 characters long. Tasks are changed every 100th experience and are changed 1000 times in total. X-axis shows experiences, Y-axis shows running mean of the reward gained from each experience. Only the first 50000 experiences are shown. The colored area is the confidence interval of one standard deviation.

4.5.2 Analysis

We see that the reinforcement learning does not perform nearly as well after a significant period of training compared to the supervised situation. However, it does perform much better than Single UCB after a moderate amount of training. This shows that for a reinforcement learning environment, using TIDE can substitute memorisation, as long as there is an underlying structure in the task descriptors.

4.6 Experiment 4 - reinforcement learning with larger task descriptors

By increasing the number of barcode characters to use, multiple barcodes can all map to the same task. This makes the process of reusing old knowledge about the tasks more important. These experiments test the generalization of the task identification. We use barcodes of 120 character length as the task descriptors. We vary the number of bandits from 10 to 60 and 120. The number of arms on each bandit and therefore the number of distinct tasks is kept at 10.

Modifying experiment 2 to use reward instead of being supervised makes this a reinforcement learning problem. We will generate a substitute for the correct task labels according to the routine described in 3.2.3.

Running the four types of agents on two barcode structures with varying number of bandits is shown in figure 4.8 and 4.10. Each of these 10, 60 or 120 bandits belong to one of the 10 different tasks that exist.

4.6.1 Results

Frequent task switches

We see that the algorithms there is a huge difference between correctly labeling 10 different task descriptors to 10 task labels and correctly labeling 120 different task descriptors to the same 10 task labels. TIDE manages to correctly label most of the task descriptors when only 10 different bandits exist after less than 10000 experiences for the correlated and descriptive barcode structures as seen in figure 4.8.

On the random barcode structure situation in figure 4.9, TIDE Distributed UCB performs better than TIDE-Oracle. Look-up Table UCB performs best.

Infrequent task switches

In figure 4.10 we see the results of running on the infrequent situations. When there are only 10 bandits, the performance of TIDE, TIDE Distributed UCD and Look-up Table UCB perform almost equally well. TIDE and TIDE Distributed UCD are trained within reasonable time. When the number of bandits increases, the time it takes to train increases massively. On the run with 120 bandits, after 100000 experiences, the performance of TIDE has still not gotten to the same level as ignoring task information,

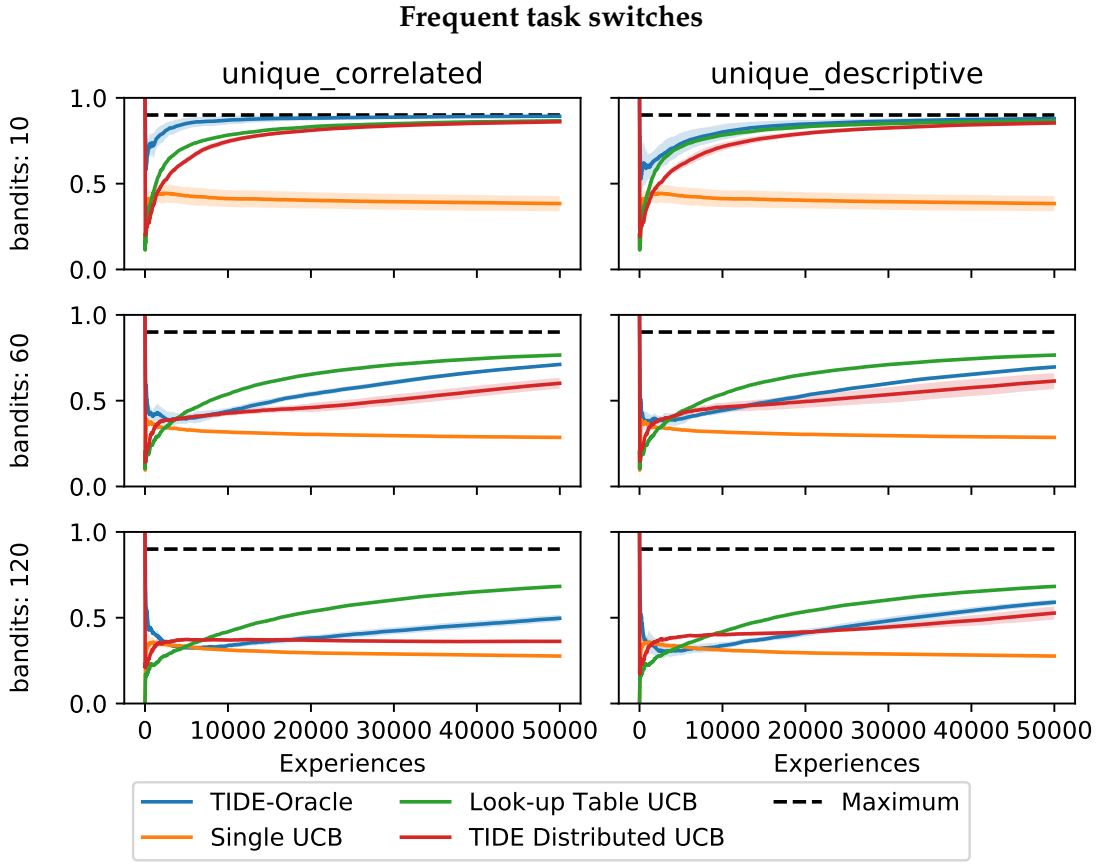


Figure 4.8: Running average reward for **Experiment 4** on two different barcode structures with frequently changing tasks. Task descriptors (barcodes) are 120 characters long. Task descriptors (barcodes) are 120 characters long. The number of bandits in total are 10, 60 or 120 (row 1, 2 and 3 respectively). Tasks are changed every 10th experience and are changed 5000 times in total. X-axis shows episodes, Y-axis shows running mean of the reward gained from each experience. The colored area is the confidence interval of one standard deviation.

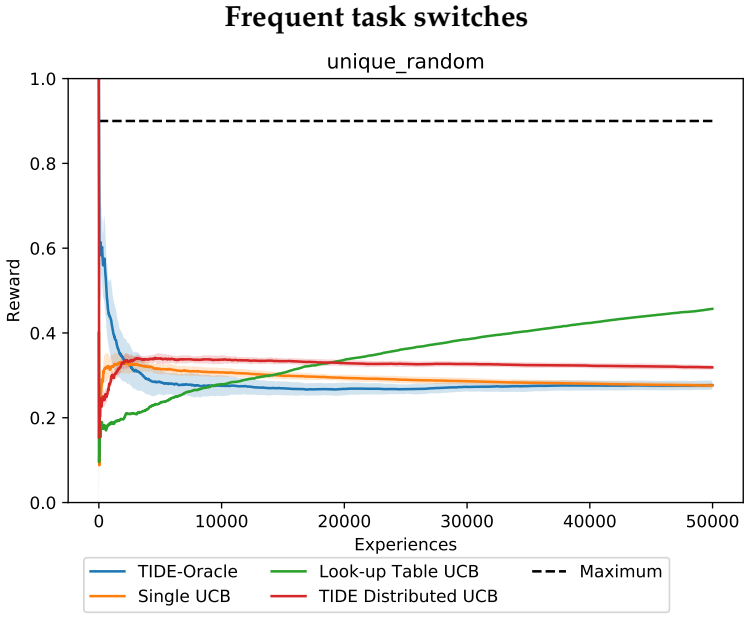


Figure 4.9: Running average reward for **Experiment 4** on the random barcode structure with frequently changing tasks. Task descriptors (barcodes) are 120 characters long. Task descriptors (barcodes) are 120 characters long. The number of bandits in total are 500. Tasks are changed every 10th experience and are changed 5000 times in total. X-axis shows episodes, Y-axis shows running mean of the reward gained from each experience. The colored area is the confidence interval of one standard deviation.

like Single UCB does. After approximately 40000 experiences in the 120 bandit situation Look-up Table UCB performs better than Single UCB.

Figure 4.11 shows the performance of the four algorithms on the random barcode structure with 120 character long task descriptors and 500 bandits. TIDE-Oracle has low performance, while TIDE Distributed UCB manages to perform on par with Single UCB after about 40000 experiences. Look-up Table UCB is close to performing as well as Single UCB.

4.6.2 Analysis

We see that even when there is a large task descriptor space, as long as there are few bandits, TIDE and TIDE Distributed UCD is able to identify the tasks with somewhat good probability after extensive training. As we saw with the supervised situation in experiment 2, not paying attention to the task descriptors results in good performance in the beginning. As the task descriptor space and number of task instances (bandits) increases, the value of having some adaptation in the agents increases, this can be seen from the better performance of TIDE Distributed UCB compared to TIDE-Oracle. When TIDE doesn't manage to properly label the tasks, the adaptation of the UCB algorithm helps TIDE Distributed UCB to perform better. When the UCB algorithm in TIDE Distributed UCB changes which arm to pull, the agent changes into the solution for a different task than it was originally assigned to solve.

On the random barcode setting with infrequent task switching, TIDE-Oracle does not manage to learn the mapping from task descriptors to task labels well within the experiences in the run. In the beginning, each time a new task is encountered, an agent is initialised to be able to solve that task. Because TIDE-Oracle uses an agent that is guaranteed to solve the task it is initialised on, TIDE-Oracle will receive reward for the first 100 experiences each time a task is mapped to a new label. When presented with new task descriptors when all task labels have initialised agents, there is a chance that the agent is reinitialised as described in 3.2.4. This reinitialisation works for a while, until the number of times each task label has been selected increases and each task label is assigned to a unique agent. TIDE-Oracle then performs worse as the labels are stabilised, and the correct labels are not applied.

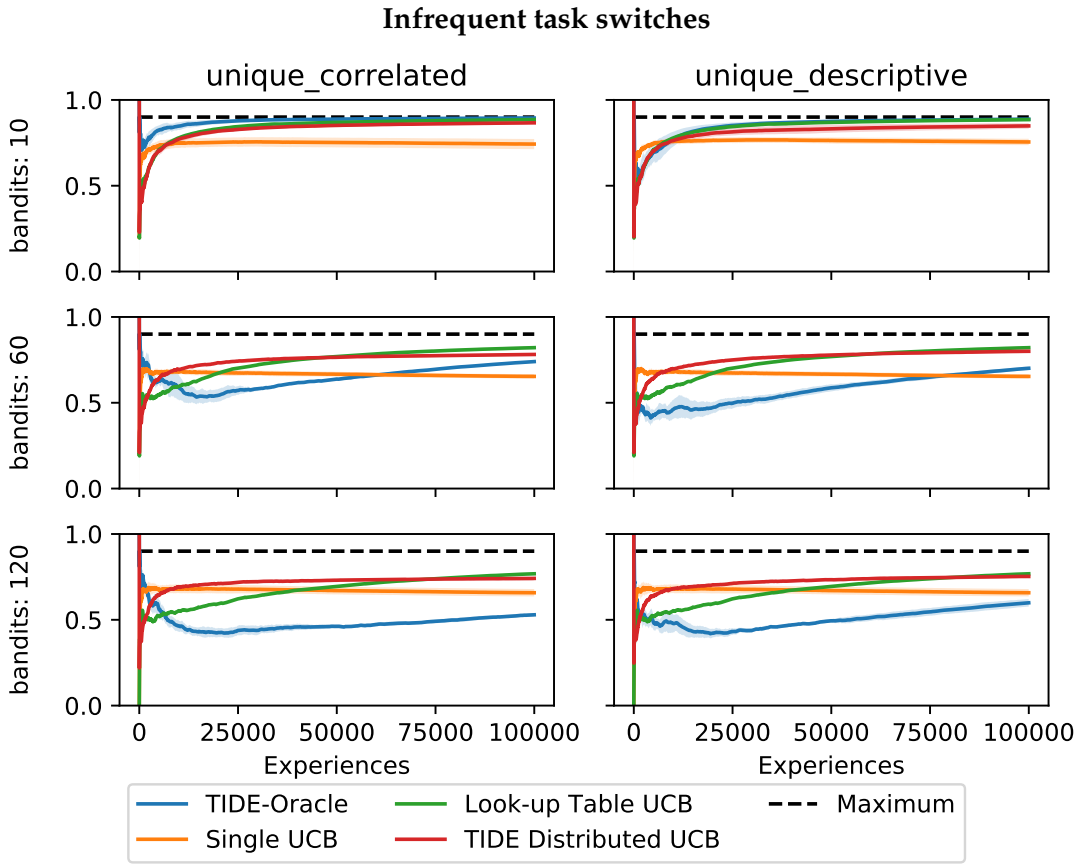


Figure 4.10: Running average reward for **Experiment 4** on two different barcode structures with infrequently changing tasks. Task descriptors (barcodes) are 120 characters long. The number of bandits in total are 10, 60 or 120 (row 1, 2 and 3 respectively). Tasks are changed every 100th experience and are changed 1000 times in total. X-axis shows experiences, Y-axis shows running mean of the reward gained from each experience. The colored area is the confidence interval of one standard deviation.

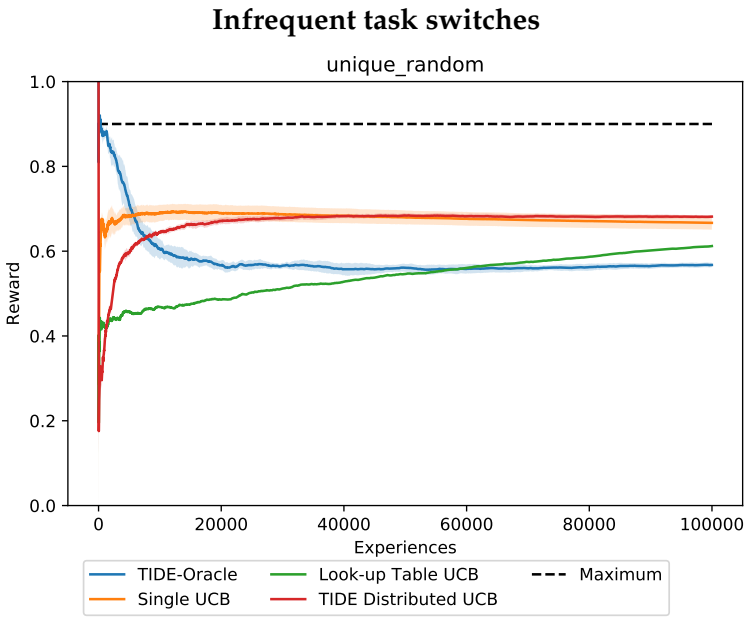


Figure 4.11: Running average reward for **Experiment 4** on the random barcode structure with infrequently changing tasks. Task descriptors (barcodes) are 120 characters long. The number of bandits in total are 500. Tasks are changed every 100th experience and are changed 1000 times in total. X-axis shows experiences, Y-axis shows running mean of the reward gained from each experience. The colored area is the confidence interval of one standard deviation.

4.7 Overview of results

Our most promising result is that learning to recognise which task is currently being presented, and reacting to this, improves performance by a great deal in a domain with frequently changing tasks. This is true as long as the task descriptors are structured in some exploitable way. If there is no structure to the task descriptors, as is the case for randomly generated task descriptors, memorising each task descriptor by the neural network is the best one can hope for.

Comparing TIDE to a few other algorithms that have good performance on general multi-armed bandit scenarios, we see that using TIDE with a simulated fast and slow system (TIDE-Oracle) on frequently changing tasks results in good performance. As long as there is some structure to the task descriptors in relation to which task it belongs to, TIDE performs very well. When there is no structure to the task descriptors, TIDE performs worse than ignoring task differences and simply relearning at each task switch. The multi-armed bandit environment is a simple environment where it is easy to adapt to a new task quickly. If this was not the case, the value of using TIDE might be higher. Learning the mappings from task descriptors to task labels is more time consuming when the tasks are changed infrequently.

As seen, solving a multi-task environment by identifying the underlying task structure and exploiting it had a big impact on performance. This relies on the tasks being switched often enough. If the environment is stationary for long periods, it is more efficient to simply use an algorithm that is able to adapt to changes slowly.

Look-up Table UCB has bad performance if there is no or little reuse of the specific barcodes for each bandit. If the barcodes uniquely identify a bandit, and each task is only identified by a single barcode, this results in good performance, as each UCB algorithm is specific to that task.

Overall, being aware of which task is currently being solved helps performance on all experiments. Look-up Table UCB performs better than Single UCB on all experiments, even on random barcodes after a long time. Being able to extract the structure of the task descriptors help in labeling the task correctly and thereby attaining higher reward.

Using supervised learning instead of reinforcement learning to learn the task labels significantly boosts performance. As such, using supervised learning when training recognition of tasks should be used where it is possible. When it is not possible to use supervised learning, using a reinforcement learning approach to train the task labeling algorithm can result in comparable performance.

4.8 Discussion

4.8.1 Tuning of parameters

Choosing parameters for TIDE-Oracle affects the results substantially. When using the supervised learning situations, setting the learning rate much higher makes the

algorithm reach good performance much quicker at the expense of possibly overwriting progress on identification of other bandits. Choosing to have the learning rate low decreases the risk of destroying earlier knowledge.

For the reinforcement learning experiments with longer barcodes and many bandits, it would probably be better to use a more exploration-focused approach. For example by setting the minimimum value of epsilon to a high value or making the exploration decay much slower. Especially on the infrequent task switching parts, many experiences has elapsed before all the different bandits have been presented. When it takes many experiences to be exposed to all the different task descriptors, the exploration part has decayed to the minimum value before all possible task descriptors have been seen. Because of this, it might not be a good idea to have a linearly decaying eploration policy with such a large task descriptor space and number of bandits.

4.8.2 Particularity of duplicated agents

If a single bandit is assigned to two different task labels, the agents that are instantiated at those task labels are labeled duplicates of each other. The next time one of those task labels are selected, the algortihm will reinstantiate the agent at that task label. This in effect recalculates the optimal choice for that task label. By doing this, the problem of a single task being solved by multiple agents is solved.

To make sure that the most successful task label for an agent is retained, only the agent that has been selected the fewest times since it was last reinstantiated will be reinstantiated. This should make sure that good solutions are not discarded.

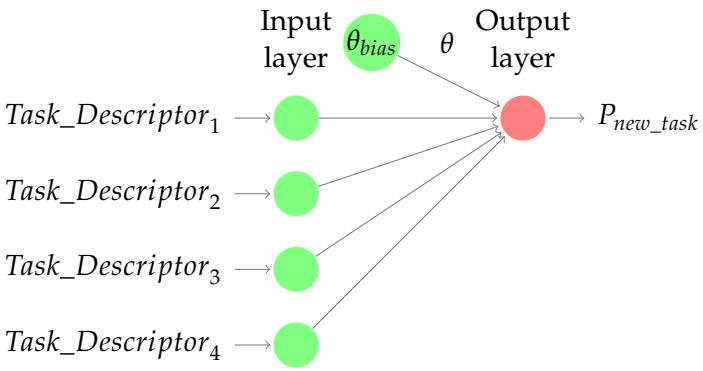
On random barcode structures, not discarding agents that have been invoked many times hinders performance. The assumption that an agent that has been invoked many times is correctly labeled does not hold for the random barcodes. Here we see that enhancing the capabilities of recognizing structure hinders the algorithm when met with no structure at all.

4.8.3 Alternative architecture for TIDE

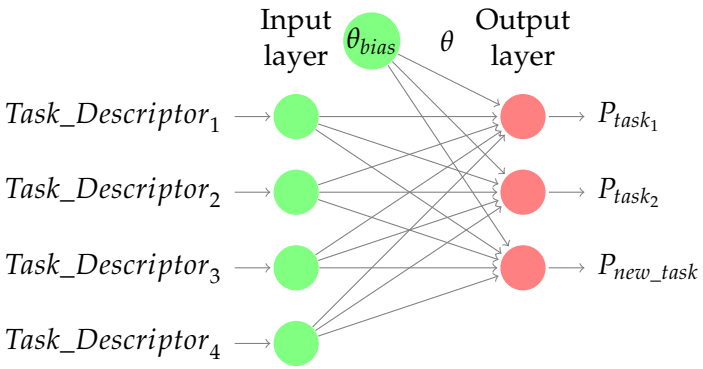
In TIDE, we assume that the number of tasks are static and known beforehand. Ideally, the TIDE system should not need to be informed of how many tasks exist in the world. It should figure this out on its own from the interactions with the environment and create distinct task labels for all different tasks it encounters.

Learning which tasks exist and dynamically adding these tasks to the space of known tasks would make the TIDE system much closer to being a lifelong learning algorithm. There are many challenges to doing this, and we have not been able to overcome them. We have tried to create an architecture for TIDE that can dynamically add tasks, but have not managed to implement it sufficiently to test it properly.

The architecture is visualised in figure 4.12. This architecture is designed to begin without any tasks defined, with a single output node that corresponds to creating a new task label and agent to solve that task label.



(a) Architecture at initialization



(b) Architecture after choosing to add a new task two times

Figure 4.12: Architecture of TIDE for dynamic number of tasks.

When a new task is created, the node that indicated a new task is repurposed to be the node for the newly created task label. A new node is created and the weights initialised. This new node corresponds to creating a new task. Figure 4.12b shows this architecture after adding two different tasks.

Initialisation of the weights for the new node should be done so that the task descriptors that already have a task label with a good agent are not overidden. This initialisation is quite difficult to get right, and as a consequence of it not being done right, previously solved tasks were erroneously identified as new tasks. This let to the creation of many more nodes than was neccessary in the architecture.

Chapter 5

Conclusion

5.1 Conclusion

Work done in artificial intelligence is far from achieving the goal of a general artificial intelligence. Multiple avenues of research are being pursued. Some of the research done in the domain of reinforcement learning, especially meta-reinforcement learning, is summarised, and some discussion on the use of these is presented. Discussion on possible implementations of human-inspired attention division is presented through the use of the two system theory. An attempt is made to reason that a prerequisite for the two system theory is an ability to recognise which system should be invoked, and a simple implementation of this recognition is presented. As a proxy for recognizing which system to invoke, knowing which task a given observation belongs to is used. The goal is being able to identify which task is being presented. The TIDE framework is proposed as a solution to this problem statement and tested on a simple multi-task environment.

Evaluating the performance of the TIDE framework on supervised learning, it performs very well. Using long task descriptors it is shown that the framework reuses knowledge for multiple instances of the same task. The TIDE framework is also tested using reinforcement learning, and performs adequately in comparison to the supervised training. Task recognition and assignment in the multi-task multi-armed bandit environment is shown to be solvable using a simple framework that should be extendible to other environments.

5.2 Future work

5.2.1 Dynamic number of tasks

TIDE relies on being informed of how many tasks exist to determine the number of task labels it should create. Extending TIDE to the architecture described in section 4.8.3 might enable TIDE to adapt to any number of tasks. By making TIDE able to

dynamically decide how many tasks to create, any number of different tasks can be solved without recreating the network.

5.2.2 Other environments

Expanding to other environments, by using the EATIDE framework presented in section 3.3, would be interesting future work. Using an environment where each task hinders performance on other tasks would be interesting. If adapting to each task is infeasible, knowing which task to choose is more important.

Solving multiple Atari games with pretrained agents from the Stable Baselines Zoo [21] would be a good test for EATIDE's ability to recognize tasks. Trying to determine which environment (Atari game) we are currently in equates to labeling the task. Using the pretrained agents that can each solve a given environment can be used as the solution agents in the same manner as the UCB algorithm was a solution agent for the multi-armed bandit environment.

5.2.3 Fast and slow system

In this thesis, the fast and slow systems are not implemented in a meaningful way as the scope would be to large. Implementing a fast and slow system with TIDE as the system selector is natural future work. Using some of the fast systems internal state as the task descriptors is a possible way of using TIDE with a fast and slow system.

Bibliography

[1] Ryan Prescott Adams and David J. C. MacKay. *Bayesian Online Changepoint Detection*. 2007. arXiv: 0710.3742 [stat.ML].

[2] Thomas Anthony, Zheng Tian and David Barber. 'Thinking Fast and Slow with Deep Learning and Tree Search'. In: *arXiv e-prints*, arXiv:1705.08439 (May 2017), arXiv:1705.08439. arXiv: 1705.08439 [cs.AI].

[3] Greg Brockman et al. 'OpenAI Gym'. In: *arXiv e-prints*, arXiv:1606.01540 (June 2016), arXiv:1606.01540. arXiv: 1606.01540 [cs.LG].

[4] J. L. Elman. 'Learning and development in neural networks: the importance of starting small'. English. In: *Cognition* 48.1 (1993). Cited By :701, pp. 71–99. URL: www.scopus.com.

[5] Chelsea Finn, Pieter Abbeel and Sergey Levine. 'Model-agnostic meta-learning for fast adaptation of deep networks'. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 1126–1135.

[6] James Harrison et al. *Continuous Meta-Learning without Tasks*. 2019. arXiv: 1912.08866 [cs.LG].

[7] Abigail N. Hoskin et al. 'Refresh my memory: Episodic memory reinstatements intrude on working memory maintenance'. In: *Cognitive, Affective, & Behavioral Neuroscience* 19.2 (Apr. 2019), pp. 338–354. ISSN: 1531-135X. DOI: 10.3758/s13415-018-00674-z. URL: https://doi.org/10.3758/s13415-018-00674-z.

[8] Jie Hu et al. *Squeeze-and-Excitation Networks*. 2017. arXiv: 1709.01507 [cs.CV].

[9] Jan Humplik et al. *Meta reinforcement learning as task inference*. 2019. arXiv: 1905.06424 [cs.LG].

[10] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[11] James Kirkpatrick et al. 'Overcoming catastrophic forgetting in neural networks'. In: *arXiv e-prints*, arXiv:1612.00796 (Dec. 2016), arXiv:1612.00796. arXiv: 1612.00796 [cs.LG].

[12] T.L Lai and Herbert Robbins. 'Asymptotically efficient adaptive allocation rules'. In: *Advances in Applied Mathematics* 6.1 (1985), pp. 4–22. ISSN: 0196-8858. DOI: https://doi.org/10.1016/0196-8858(85)90002-8. URL: http://www.sciencedirect.com/science/article/pii/0196885885900028.

[13] T. S. Li et al. 'Robots That Think Fast and Slow: An Example of Throwing the Ball Into the Basket'. In: *IEEE Access* 4 (2016), pp. 5052–5064. DOI: 10.1109/ACCESS.2016.2601167.

[14] David Lopez-Paz and Marc'Aurelio Ranzato. 'Gradient episodic memory for continual learning'. In: *Advances in Neural Information Processing Systems*. 2017, pp. 6467–6476.

[15] D. Marr. 'Simple Memory: A Theory for Archicortex'. In: *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences* 262.841 (1971), pp. 23–81. ISSN: 00804622. URL: http://www.jstor.org/stable/2417171.

[16] Nicolas Y. Masse, Gregory D. Grant and David J. Freedman. 'Alleviating catastrophic forgetting using context-dependent gating and synaptic stabilization'. In: *Proceedings of the National Academy of Sciences* 115.44 (2018), E10467–E10475. ISSN: 0027-8424. DOI: 10.1073/pnas.1803839115. eprint: https://www.pnas.org/content/115/44/E10467.full.pdf. URL: https://www.pnas.org/content/115/44/E10467.

[17] Michael McCloskey and Neal J. Cohen. 'Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem'. In: ed. by Gordon H. Bower. Vol. 24. Psychology of Learning and Motivation. Academic Press, 1989, pp. 109–165. DOI: https://doi.org/10.1016/S0079-7421(08)60536-8. URL: http://www.sciencedirect.com/science/article/pii/S0079742108605368.

[18] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[19] Anusha Nagabandi, Chelsea Finn and Sergey Levine. 'Deep Online Learning via Meta-Learning: Continual Adaptation for Model-Based RL'. In: *arXiv e-prints*, arXiv:1812.07671 (Dec. 2018), arXiv:1812.07671. arXiv: 1812.07671 [cs.LG].

[20] Alexander Pritzel et al. *Neural Episodic Control*. 2017. arXiv: 1703.01988 [cs.LG].

[21] Antonin Raffin. *RL Baselines Zoo*. https://github.com/araffin/rl-baselines-zoo. 2018.

[22] Samuel Ritter et al. 'Been There, Done That: Meta-Learning with Episodic Recall'. In: *arXiv e-prints*, arXiv:1805.09692 (May 2018), arXiv:1805.09692. arXiv: 1805.09692 [stat.ML].

[23] Maruan Al-Shedivat et al. *Continuous Adaptation via Meta-Learning in Nonstationary and Competitive Environments*. 2017. arXiv: 1710.03641 [cs.LG].

[24] David Silver et al. 'Mastering the game of Go without human knowledge'. In: *Nature* 550 (2017), pp. 354–359.

[25] Haozhe Wang, Jiale Zhou and Xuming He. *Learning Context-aware Task Reasoning for Efficient Meta-reinforcement Learning*. 2020. arXiv: 2003.01373 [cs.LG].

[26] Jane X Wang et al. 'Learning to reinforcement learn'. In: *arXiv e-prints*, arXiv:1611.05763 (Nov. 2016), arXiv:1611.05763. arXiv: 1611.05763 [cs.LG].

[27] Christopher Watkins and Peter Dayan. 'Technical Note: Q-Learning'. In: *Machine Learning* 8 (May 1992), pp. 279–292. DOI: 10.1007/BF00992698.

[28] the free encyclopedia Wikipedia. *Reinforcement learning diagram*. [Online; accessed February 4, 2020]. 2017. URL: https://commons.wikimedia.org/wiki/File:Reinforcement_learning_diagram.svg.