

https://dspace.org/

Improving reinforcement learning with ensembles of different learners

Crafford, Gerrie

2021

https://hdl.handle.net/10539/32383

Crafford, Gerrie (2021) Improving reinforcement learning with ensembles of different learners, University of the Witwatersrand, Johannesburg, <http://hdl.handle.net/10539/32383>

Downloaded from a DSpace repository. https://dspace.org/

Improving Reinforcement Learning with Ensembles of Different Learners



WITS
UNIVERSITY

Gerrie Crafford
2288909

A dissertation, supervised by
Prof. Benjamin Rosman

submitted to the
Faculty of Science, University of the Witwatersrand,
in fulfilment of the requirements for the degree Master of Science.

April 2021

Declaration

I, Gerrie Crafford, hereby declare the contents of this research dissertation to be my own work unless otherwise explicitly state. This dissertation is submitted for the degree of Master of Science (Dissertation) at the University of the Witwatersrand, Johannesburg. This work has not been submitted to any other university, nor any other degree.



Signature

2021-04-12

Date

Acknowledgements

I would like to extend my gratitude toward my supervisor, Prof. Benjamin Rosman. His patient guidance, keen insight, and constant encouragement made this research possible. I could not have asked for a better supervisor.

I am grateful that the Lord gave me the opportunity to further my studies and I am thankful for His provision for every step of the way. I would also like to thank my wife for her support and for listening to my ramblings about Reinforcement Learning, without complaint, and truly entering my world by learning about RL and being excited with me about successes in my work. I am also grateful for my mother's interest in and support of my studies.

Finally, I would like to thank the Council for Scientific and Industrial Research for supporting my studies financially, providing guidance and making sure that I had everything I needed to complete my studies successfully.

Abstract

Different reinforcement learning methods exist to address the problem of combining multiple different learners to generate a superior learner, from ensemble methods to policy reuse methods. These methods usually assume that each learner uses the same algorithm and/or state representation and often require learners to be pre-trained. This assumption prevents very different types of learners, that can potentially complement each other well, from being used together.

We propose a novel algorithm, Adaptive Probabilistic Ensemble Learning (APEL), which is an ensemble learner that combines a set of base reinforcement learners and leverages the strengths of the different base learners online, while remaining agnostic to the inner workings of the base learners, thereby allowing it to combine very different types of learners. The ensemble learner selects the base learners that perform best on average by keeping track of the performance of the base learners and then probabilistically selecting a base learner for each episode according the historical performance of the base learners. Along with a description of the proposed algorithm, we present a theoretical analysis of its behaviour and performance.

We demonstrate the proposed ensemble learner's ability to select the best base learner on average, combine the strengths of multiple base learners, including Q-learning, deep Q-network (DQN), Actor-Critic with Experience Replay (ACER), and learners with different state representations, as well as its ability to adapt to changes in base learner performance on grid world navigation tasks, the Cartpole domain, and the Atari Breakout domain. The effect that the ensemble learner's hyperparameter has on its behaviour and performance is also quantified through different experiments.

Contents

1 Introduction **1**

2 Problem Setup and Background **3**
2.1 Ensemble learner setting . . . 3
2.2 k-armed Bandit . . . 4
2.2.1 Exploring bandits with confidence . . . 4
2.3 Reinforcement Learning . . . 5
2.3.1 Policy . . . 6
2.3.2 Value functions . . . 6
2.3.3 Exploration strategies . . . 7
2.4 Different types of learners . . . 7
2.4.1 Learning on-policy vs off-policy . . . 8
2.4.2 Learning the value function . . . 8
2.4.3 Learning the policy . . . 9
2.4.4 Learning with an actor and a critic . . . 9
2.4.5 Learning the model . . . 10
2.4.6 State space and action space representation . . . 10
2.5 Ensemble learning . . . 10
2.5.1 Ensemble Algorithms in Reinforcement Learning . . . 12
2.6 Learners as black boxes . . . 13
2.7 Summary . . . 14

3 Learning with an Ensemble of Learners **15**
3.1 Concentration parameters . . . 16
3.1.1 Dirichlet distribution . . . 16
3.1.2 Initialisation of concentration parameters . . . 16
3.2 Learner selection . . . 17
3.3 Adjusting concentration parameters . . . 17
3.4 Decaying concentration parameters . . . 18
3.5 Evolution of concentration parameters . . . 19
3.5.1 Incremental updates with decay . . . 19
3.5.2 Decay only . . . 20
3.5.3 Adapting to changing base learner performance . . . 20
3.6 Selecting the decay parameter . . . 21
3.6.1 Decay according to adaptation time . . . 21
3.6.2 Decay according to rise time . . . 22

3.6.3 Decay according to steady-state exploration probability . . . 23
3.6.4 Decay according to steady-state performance . . . 23
3.7 Summary . . . 24

4 Experiments **25**
4.1 Experimental setup . . . 25
4.1.1 Maze domain . . . 25
4.1.2 Cartpole domain . . . 26
4.1.3 Breakout domain . . . 27
4.1.4 Random seeds . . . 27
4.2 Qualitative experiments . . . 27
4.2.1 Selection behaviour with two base learners . . . 27
4.2.2 Adaptation behaviour with two base learners . . . 28
4.2.3 Leveraging the strengths of multiple learners online . . . 31
4.3 Quantitative experiments . . . 32
4.3.1 Outperforming individual learners on average . . . 32
4.3.2 Adapting to changing learner performance . . . 36
4.3.3 Effect of decay parameter on steady-state performance . . . 40
4.3.4 Effect of additional learners on performance . . . 41
4.3.5 Comparison with majority voting ensemble learners . . . 44
4.3.6 Hyperparameter selection . . . 45
4.4 Summary . . . 46

5 Related Work **48**
5.1 Framing algorithm selection as a bandit problem . . . 48
5.2 Policy reuse . . . 49
5.3 Task subdivision . . . 50
5.4 AutoML . . . 51
5.5 Summary . . . 51

6 Conclusions and Future Work **52**

List of Figures

2.1 Agent interaction with environment in Reinforcement Learning (RL) setting. . . 5
2.2 Diagram of the DQN algorithm's neural network architecture which consists of four convolutional layers followed by two fully connected layers (Mnih, 2017). . . 9
2.3 Mixture of experts gated by gating network. Adapted from Jacobs et al. (1991). 12
2.4 Depiction of information sharing between black box learners. 14

3.1 Diagram of ensemble learner's interaction with base learners. 16
3.2 Progression of concentration parameters and the subsequently generated distributions. 20
3.3 Evolution of Adaptive Probabilistic Ensemble Learning (APEL)'s values after optimal base learner has been switched. 21
3.4 Concentration parameter metrics as a function of the decay parameter. 22

4.1 Examples of mazes generated by the maze environment. The agent starting locations and goal locations are shown by blue and green dots respectively. 26
4.2 Screenshots of the Cartpole environment in different states. 27
4.3 Screenshots of the Atari Breakout environment in different states. 28
4.4 Curves showing the change in APEL's parameters over time for $\nu = 0.01$ with variances shown by shaded regions. 29
4.5 Curves showing the change in APEL's parameters after switching the optimal base learner for $\nu = 0.01$. The switch point is indicated by the dashed vertical line and the variances of the curves are shown by shaded regions. 30
4.6 Learning curves of different learners on Breakout domain. 33
4.7 Mean train time for different learners, with error bars showing standard deviation. 34
4.8 Distribution of train times for different learners. 35
4.9 Mean train time for learners, averaged over both domains. 36
4.10 Mean train time for learners for each domain. 37
4.11 Time to adapt to base learner changes for different switch points. 38
4.12 Distribution of adaptation times for different switch points. 39
4.13 Effect of ν on the time it takes APEL to adapt. 39
4.14 Effect of ν on APEL's steady-state performance when using a bad learner or a slow learner. Q_{bad} is a straight line at -1000 that is left out to keep the other lines visible. 41
4.15 Effect of additional (redundant) base learners on the train time of the ensemble learner. 42
4.16 Effect of additional base learners on the train time of the ensemble learner when using Q-learners and ACER-learners on the maze environment. 43

4.17 Effect of additional base learners on the train time of the ensemble learner when using Q-learners and ACER-learners on the Cartpole environment. 43
4.18 Mean train time for base learners and ensemble learners. 44
4.19 Learning curves for base learners and ensemble learners. 45
4.20 Performance of APEL and SSBAS as their hyperparameters are varied. 46

Acronyms

AC-Teach Actor Critic with Teacher Ensembles.

ACER Actor-Critic with Experience Replay.

APEL Adaptive Probabilistic Ensemble Learning.

BPR Bayesian Policy Reuse.

CAPS Context-Aware Policy Reuse.

DQN deep Q-network.

ESBAS Epochal Stochastic Bandit Algorithm Selection.

KL Kullback-Leibler.

MAd-RL Multi-Advisor Reinforcement Learning.

MBMF Mode-Based Model-Free.

MDP Markov Decision Process.

ME-TRPO Model-Ensemble Trust Region Policy Optimisation.

PPO Proximal Policy Optimisation.

PRQL Policy Reuse in Q-learning.

RL Reinforcement Learning.

SAC Soft Actor-Critic.

SSBAS Sliding Stochastic Bandit Algorithm Selection.

TD Temporal-Difference.

TRPO Trust-Region Policy Optimisation.

UCB Upper Confidence Bound.

Chapter 1

Introduction

In recent years, Reinforcement Learning (RL) has proven its ability to solve a variety of different problems, from superhuman game performance, to solving robotics tasks. Despite RL's success in many areas, a designer is still faced with a choice of a plethora of different algorithms, each with many parameters to tune, in order to design an RL system capable of solving any given problem.

Selecting the best algorithm or parameters for an algorithm beforehand, or even combining the benefits of different approaches, is challenging because of the different properties of each algorithm that often only become apparent after the task has been solved and the algorithm's performance has been analysed.

More generally, given a few different learners (which could even have different state representations), some learners might perform better on some tasks while performing worse on other tasks due to the properties of the learner or algorithm. Consider, for example, a model-based learner and a model-free learner training on the same task. The model-based learner would usually have better sample complexity than the model-free learner, but if the model-based learner's model was inaccurate, its performance or sample complexity would degrade, and the model-free learner would instead perform better. One could also consider learners with different sets of state features (a full feature set and a reduced feature set, for example). On some tasks, a reduced feature set could be helpful and allow the learner to learn faster, but on other tasks the reduced feature set could be inadequate and prevent the learner from reaching optimal performance or even completing the task.

Ensemble machine learning methods are methods that use multiple base models and combine the results of the models to obtain improved outputs. Ensemble methods are used successfully in many machine learning applications, including in RL where they can be used to combine multiple policies or value functions to obtain an agent with improved performance and/or sample complexity. A variety of methods can be used to determine which action the agent should take, from rank voting to Boltzmann multiplication (Wiering and Van Hasselt, 2008) or even using Temporal-Difference (TD)-learning to learn weights to use when combining the action-value functions of the set of base agents (Marivate and Littman, 2013).

Existing ensemble RL methods use a set of learners that use the same internal representations (all learners use action-value functions internally, for example) and the same state representations (Chen et al. 2017; Wiering and Van Hasselt 2008). The only related method that uses

an ensemble of learners but also remains agnostic to the learners' internal representations and learning mechanisms (allowing, for example, value-based learners and actor-critic learners to be used together) is the Sliding Stochastic Bandit Algorithm Selection (SSBAS) algorithm (Laroche and Feraud, 2018), which we use as a baseline.

We propose a novel algorithm, Adaptive Probabilistic Ensemble Learning (APEL), which is an ensemble learner capable of handling the more general case (handling learners with different state representations and learning mechanisms). APEL runs multiple base learners online and selects the base learners that have the best performance throughout training while only requiring base learners to be able to learn from off-policy experience and making no assumptions about the state representation or learning mechanism of base learners.

APEL uses a Dirichlet distribution to shift probability mass to base learners that perform well, while still allowing periodic random exploration of other base learners to allow it to adapt to changes in the performance of the base learners over time.

We show that in tasks where base learners sometimes perform well and other times perform poorly, APEL performs better than any of the individual base learners on average, i.e. APEL's average performance over multiple tasks is better than any of the base learners' average performance over multiple tasks.

The contribution of this work is the APEL algorithm which is capable of: (1) outperforming any individual base learner on average over multiple tasks, (2) adapting to changing base learner performance, and (3) leveraging the strengths of multiple base learners online.

We provide an overview of RL, the different types of RL learners, ensemble learning and the setting considered in this work in Chapter 2. In Chapter 3 we present the APEL algorithm and show how it selects good base learners during training. We present various experiments that show APEL's behaviour, compare it to SSBAS, and quantify its performance in Chapter 4. The SSBAS algorithm, as well as other related work, is discussed in Chapter 5. The thesis is concluded and avenues for future work are discussed in Chapter 6.

Chapter 2

Problem Setup and Background

In this chapter we describe the setting we consider in this work (Section 2.1), introduce the k-armed bandit problem (Section 2.2), give a high-level overview of Reinforcement Learning (RL) and different types of RL algorithms (Section 2.3 and 2.4) as well as an overview of ensemble learning and its applications to RL (Section 2.5). Finally, we detail our approach to sharing information between different base learners (Section 2.6).

2.1 Ensemble learner setting

We consider the case where a high-level learner has access to multiple base learners and needs to decide which base learner to use at each timestep. We call this high-level learner an ensemble learner in this work.

In this setting, there are two timescales that are of interest:

1. the base learner timescale, t, which denotes a timestep within an episode, and
2. the ensemble learner timescale, e_k, which denotes an episode.

Base learners make decisions and learn at each timestep t. The ensemble learner, on the other hand, only makes decisions and learns at the start of each episode e_k (not at each timestep t within the episode).

An ensemble learner is similar to bandit algorithms in the sense that it can take actions and receive rewards. The ensemble learner acts by selecting a base learner for an episode and the reward it observes is the episode's return (the cumulative reward obtained by the selected base learner while acting over the entire episode). An ensemble learner can also keep track of certain state information to aid it in choosing base learners and in this case the ensemble learner is similar to a contextual bandit.

2.2 k-armed Bandit

The k-armed bandit problem considers a setting where an agent has a choice of k actions and receives some numerical payoff or reward each time after selecting an action (Berry and Fristedt, 1985). The goal of the agent is to learn a strategy to select actions in such a way that the reward it receives is maximised.

The problem can be formalised by defining Q as the action-value function of the k-armed bandit as shown in Equation 2.1, where R_t is the reward received by the agent at time t, A_t is the action a executed by the agent at time t and $\mathbb{E}[X]$ is the expected value of the random variable X (Sutton and Barto, 2018). The action-value function gives the value (expected reward) of executing each action. The optimal action-value function is denoted by Q_*, whereas Q denotes any action-value function (not necessarily the optimal one).

$$Q_*(a) \doteq \mathbb{E}\left[R_t | A_t = a\right] \tag{2.1}$$

Once the agent knows the value of Q_* with certainty, the agent can select the best action each time, according to Equation 2.2, by simply selecting the action that maximises Q_*.

$$a_* = \underset{a}{\operatorname{argmax}}\, Q_*(a) = \underset{a}{\operatorname{argmax}}\, \mathbb{E}\left[R_t | A_t = a\right] \tag{2.2}$$

Simple k-armed bandits can be extended to contextual bandits by adding state information (Sutton and Barto, 2018), where the best action to take may be different for different states.

The reward given by each action can be (and is in most interesting problems) stochastic. The agent therefore needs to select each action many times to be able to learn the expected reward given by each action. This results in competing objectives:

- Exploration: the agent needs to select actions it has not tried yet (or has not tried enough) in order to gain information, i.e. to improve its estimate of that action's expected reward.
- Exploitation: the agent needs to select actions that, according to its current knowledge, give high reward in order to maximise its total reward.

Exploring too much limits the reward obtained by the agent over a period of time because actions that give low reward on average are selected often, whereas too much exploitation might cause the agent to continually select a suboptimal action because it has not explored enough to know that one of the other actions gives higher reward on average.

There are many different strategies for balancing the agent's exploration and exploitation in bandit problems. One simple solution, ε-greedy exploration, selects a random action with probability ε and selects the optimal action (see Equation 2.2) the rest of the time. Another solution, shown in Section 2.2.1, chooses to explore based on uncertainty about each action's value.

2.2.1 Exploring bandits with confidence

The Upper Confidence Bound (UCB) algorithm, shown in Algorithm 1, incorporates a measure of uncertainty into the action-value itself and then always selects the action with the highest action-value (Auer et al., 2002). The estimated action-value for each action, $\bar{Q}_j$, is augmented with an extra term as shown in Equation 2.3, where n_j is the number of times action j has been taken and n is the total number of actions that have been taken so far. The augmented action-value,

Q'_j, grows when actions other than action j are selected. Intuitively, Q'_j grows as the uncertainty about its value grows but eventually saturates in such a way that the growing uncertainty does not outweigh a big difference in expected reward between the different actions.

$$Q'_j = \bar{Q}_j + \sqrt{\frac{2 \ln n}{n_j}} \tag{2.3}$$

Algorithm 1 UCB1

Require: j actions
1: **Initialization:** Execute each action once
2: **for** iteration in iterations **do**
3: $Q'_j \leftarrow \bar{Q}_j + \sqrt{\frac{2 \ln n}{n_j}}$
4: $j \leftarrow \text{argmax}_j \, Q'_j$
5: Take action j and receive reward r
6: Update $\bar{Q}$ with r
7: **end for**

2.3 Reinforcement Learning

Reinforcement Learning (RL) considers an agent that can interact with and observe an environment. The agent interacts with the environment by performing actions in the environment and can observe the state of the environment as well as the reward given to it by the environment for its actions. This agent-environment interaction is depicted in Figure 2.1. RL is the extension of the contextual k-armed bandit problem to a long horizon setting. In the contextual bandit setting, the agent observes a state, chooses an action, receives a reward and then the process repeats. RL, instead, allows the agent to keep on performing actions, moving to and observing new states, and receiving rewards (often only receiving reward many steps after having performed the necessary actions).

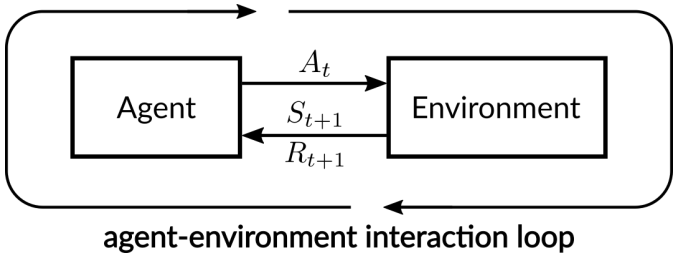


Figure 2.1: Agent interaction with environment in RL setting.

RL problems are often framed as Markov Decision Processes which have the following elements: (Puterman, 1994).

- $\mathcal{S}$: The set of states that the environment can have and that the agent can observe.
- $\mathcal{A}(s)$: The set of actions that the agent can take in any given state $s \in \mathcal{S}$. The simplifying assumption is often made that all actions are available in all states.

- P: The MDP's dynamics function, defined by $P : \mathcal{S} \times \mathcal{A} \to \mathcal{P}$, defines the probability for the agent to transition to any state, given its current state and the action taken by the agent.
- R: The reward function, defined by $R : \mathcal{S} \times \mathcal{A} \to \mathcal{R}$, determines the rewards received by the agent through its interaction with the MDP.
- γ: The discount factor of the MDP which controls how myopic the agent is by de-valuing future rewards.

A trajectory, τ, of length T is a sequence tuples, $\tau = \langle (s_1, a_1, r_1), \ldots, (s_T, a_T, r_T) \rangle$, generated by an agent through its interaction with the MDP.

The return G is the cumulative reward received by the agent over a period of time. In the simplest case (Equation 2.4) the return is simply the sum of the rewards the agent will receive for the actions it takes at each timestep.

$$G_t \doteq R_{t+1} + \cdots + R_T \tag{2.4}$$

In a discounted MDP, the return is the sum of the rewards weighted by the discount factor (Equation 2.5).

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.5}$$

In this work we consider episodic MDPs, where an episode is terminated either when the agent reaches a terminal state, or when the agent has interacted with the MDP T times.

2.3.1 Policy

An agent's policy determines the actions it takes in each state. The policy can either be deterministic (always selecting the same action in any given state) or stochastic (selecting actions according to a probability distribution). Formally, a policy is a function $\pi : \mathcal{S} \to \mathcal{A}$ that determines the action taken by the agent in each state $s \in \mathcal{S}$ defined for any action $a \in \mathcal{A}(s)$ (Sutton and Barto, 2018).

2.3.2 Value functions

Value functions estimate the value of being in a certain state – it tells the agent how good it is to be in that particular state. Good states are defined as states where the agent can expect to receive a high future reward, i.e. where the expected return G_t is high. The value function under a particular policy π is denoted as $v_\pi(s)$ and gives the expected return when starting in state s and following policy π thereafter. The Bellman equation for $v_\pi(s)$ is given by Equation 2.6 (Bellman, 1957), where $\mathbb{E}_\pi$ is the expected value when following policy π. The Bellman equation defines the relationship between a state's value and the values of its successor states.

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right], \forall s \in \mathcal{S} \tag{2.6}$$
$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s') \right], \forall s \in \mathcal{S}$$

Similar to the value function, the action-value function tells the agent how good it is to take a certain action in a particular state. The definition of the action-value function of taking action a while in state s under π is shown in Equation 2.7

$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right], \forall s \in \mathcal{S} \tag{2.7}$$

A policy can be derived from a value function by selecting the action that maximises the value function in each state (also called a greedy policy). It follows that the optimal policy can be obtained by first learning the optimal value function and then deriving the optimal policy from it.

2.3.3 Exploration strategies

Exploration is a crucial part of RL, since it is the exploration (as opposed to exploitation) that allows the agent to learn which actions are rewarding. There is a trade-off between exploration and exploitation since too much of either will lead to suboptimal performance. Too much exploration will cause the agent to select random actions (which will often be suboptimal) often, whereas too much exploitation can cause the agent to constantly use suboptimal actions because it has not explored enough to find the optimal actions. Most exploration strategies perform a lot of exploration at the start of training, but explore less (and therefore exploit more) as training progresses. Although the amount of exploration decreases as training progresses, most exploration strategies never completely stop exploring to allow the agent to discover new high-value actions if the environment's reward dynamics change.

A common exploration strategy, is the ϵ-greedy exploration strategy, that selects the optimal action with probability $1 - \epsilon$ and selects a random action otherwise Sutton and Barto (2018). This simple heuristic works well in practice, especially when ϵ is decreased as training progresses.

Another exploration strategy, Boltzmann exploration, instead uses a Boltzmann distribution to choose actions probabilistically according to their expected rewards, selecting actions with high expected rewards with higher probability than actions with low expected rewards Kaelbling et al. (1996). The exploration-exploitation trade-off is controlled by the Boltzmann distribution's temperature parameter. This temperature parameter is also decreased as training progresses to decrease the amount of exploration performed as the agent becomes more certain of its value estimates.

2.4 Different types of learners

In this work, we propose an ensemble learner that remains agnostic to the inner workings of its base learners. We see a learner's inner workings as its learning mechanism and state/action

representation and give an overview of the different learning mechanisms and state/action representations that learners can use in this section.

When the learning mechanisms of different learners are the same, information can often be shared between the learners by, for example, averaging over the action-value functions of the learners or by sharing weights between neural networks. When these mechanisms are different, however, it becomes difficult to share information between learners. We detail our approach to sharing information between base learners while remaining agnostic to their inner workings in Section 2.6.

2.4.1 Learning on-policy vs off-policy

RL algorithms can learn from on-policy or off-policy experience. An algorithm learns on-policy when the same policy that is being learnt is also used to generate actions. Off-policy learning, in contrast, learns from data generated by a different policy from the policy that is being learnt.

On-policy algorithms, such as Trust-Region Policy Optimisation (TRPO) Schulman et al. (2015) and Proximal Policy Optimisation (PPO) Schulman et al. (2017), are generally more stable, since they use data generated by the current policy to learn the value function/policy, but require more experience to train. Off-policy algorithms, such as the DQN algorithm and ACER, are more data-efficient since they can use techniques like experience replay, but are not as stable as on-policy algorithms when training.

2.4.2 Learning the value function

A policy can be derived from a learned value function by selecting the action that has the highest value in each state. Subsequently, a class of methods solve the RL problem by first learning the value (or action-value) function and then using that value (or action-value) function to derive a policy to allow the agent to act optimally in the environment.

Q-learning (Watkins and Dayan, 1992) is one such algorithm that learns an action-value function and uses the learned action-value function to derive a policy. The update rule of Q-learning is shown in Equation 2.8, where s' and a' are the next state and action respectively (at time $t+1$).

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{2.8}$$

The Q-learning algorithm is shown in Algorithm 2. This version of the Q-learning algorithm assumes that tasks are episodic.

The deep Q-network (DQN) algorithm (Mnih et al., 2015) extends the standard Q-learning algorithm to continuous state spaces by using a neural network to approximate the action-value function. A convolutional neural network is used to allow the algorithm to learn from images. The architecture employed by the DQN algorithm is shown in Figure 2.2.

The Q-learning algorithm and the DQN algorithm are both off-policy algorithms, which means that they can learn from experience that was generated by a policy different from the algorithm's current policy. In practice this means that these algorithms can learn from experience that was obtained by the algorithm at an earlier stage (when the policy was different) or even from experience that was obtained by a different version of the same algorithm (e.g. a different

Algorithm 2 Q-learning

Initialise $Q(s, a)$
for episode in episodes **do**
    Initialise s
    for step in episode **do**
        $a \leftarrow \text{argmax}_a\, Q(s, a)$
        Execute a
        Observe s' and r
        Update $Q(s, a)$ according to Equation 2.8
        $s \leftarrow s'$
    end for
end for

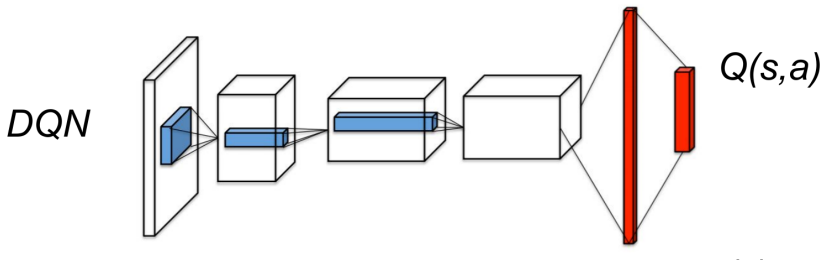


Figure 2.2: Diagram of the DQN algorithm's neural network architecture which consists of four convolutional layers followed by two fully connected layers (Mnih, 2017).

instantiation of Q-learning with different parameters). Q-learning and the DQN algorithm's property of being off-policy also allows them to be used in our work and therefore we use both these algorithms in our experiments.

2.4.3 Learning the policy

Instead of first learning the value (or action-value) function and then using that to derive the policy, the policy can also be learnt directly as a parametrised function through policy optimisation. Policy optimisation uses gradient ascent to learn the parameters of the policy that result in the highest return G_t.

REINFORCE (Williams, 1992) is an instantiation of policy optimisation that uses Monte-Carlo sampling to estimate returns with which it updates the policy's parameters.

2.4.4 Learning with an actor and a critic

The Actor-Critic approach combines the previous two approaches and has a critic that learns the value (or action-value) function in addition to an actor that learns the policy. The actor chooses actions to take and the critic gives a TD-error (the difference between value estimates at consecutive timesteps) that estimates how good the action was. This error term is then used by the actor and the critic for learning.

Soft Actor-Critic (SAC) (Haarnoja et al., 2018) is an instantiation of the actor-critic algorithm that uses off-policy experience and follows the maximum entropy RL framework. Actor-Critic

with Experience Replay (ACER) (Wang et al., 2017) is another example of an actor-critic algorithm. It incorporates experience replay into the actor-critic model and employs an off-policy value estimator. The stability of this estimator is controlled by using Retrace Q-value estimation and truncating the importance weights (used in importance sampling) with bias correction. Trust region policy updates, where a Kullback-Leibler (KL) divergence constraint is used to limit the size of policy updates, are also used to improve training stability. The ACER algorithm's pseudocode is shown in Algorithm 3 and 4, where $\mu(\cdot|x_t)$ is the behaviour policy (the policy that the agent uses to choose actions), $f(\cdot|\phi_{\theta'}(x_i))$ is the target policy (the policy that is being trained) and $Q_{\theta'_v}(x_i, \cdot)$ is the action-value estimate at timestep i. x_i, a_i, r_i denote the state, action and reward at timestep i.

Algorithm 3 ACER for discrete actions (master algorithm) Wang et al. (2017)

repeat
  Call ACER on-policy, Algorithm 4
  $n \leftarrow \text{Poisson}(r)$
  for $i \in \{1, \cdots, n\}$ **do**
    Call ACER off-policy, Algorithm 4
  end for
until Max iteration or time reached.

2.4.5 Learning the model

The aforementioned methods of learning all did so without being given a model of the world, i.e. they are so-called model-free RL methods. Alternatively, an RL agent can learn to solve a task by using a model of the world to perform planning to determine the optimal policy for the task. This approach is called model-based RL. In practice, world models are rarely available beforehand and therefore model-based RL methods, such as Mode-Based Model-Free (MBMF) (Nagabandi et al., 2018) and Model-Ensemble Trust Region Policy Optimisation (ME-TRPO) (Kurutach et al., 2018), often learn the world model from experience while they are learning to solve the task.

2.4.6 State space and action space representation

Learners can use either discrete state spaces, where value functions and policies can be represented in tabular form, or they can use continuous state spaces where value functions and policies are represented by parametrised functions. An example of an environment with a discrete state space is a grid world environment. On the other hand, state variables like a robot's joint angles are typically considered to be continuous.

Similarly, a learner's action space can also be either discrete or continuous. Discrete action spaces are commonly used with higher level actions (such as "move left"in a grid world, "fire" in an Atari game environment), whereas lower level actions (such as robot motor torques or forces applied to a pendulum) are commonly represented by continuous action spaces.

2.5 Ensemble learning

Ensemble learning is a field in machine learning that studies the use of multiple models (an ensemble) to solve a single problem (that would ordinarily be solved with a single model). Multiple models are trained on the available data and then the output from the multiple models are

Algorithm 4 ACER for discrete actions Wang et al. (2017)

Reset gradients $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
Initialise parameters $\theta' \leftarrow \theta$ and $\theta'_v \leftarrow \theta_v$
if not On-Policy **then**
    Sample the trajectory $\{x_0, a_0, r_0, \mu(\cdot|x_0), \cdots, x_k, a_k, r_k, \mu(\cdot|x_k)\}$ from the replay memory.
else
    Get state x_0
end if
for $i \in \{0, \cdots, k\}$ **do**
    Compute $f(\cdot|\phi_{\theta'}(x_i)), Q_{\theta'_v}(x_i, \cdot)$ and $f(\cdot|\phi_{\theta_a}(x_i))$
    if On-Policy **then**
        Perform a_i according to $f(\cdot|\phi'_\theta(x_i))$
        Receive reward r_i and new state x_{i+1}
        $\mu(\cdot|x_i) \leftarrow f(\cdot|\phi_{\theta'}(x_i))$
    end if
    $\bar{\rho}_i \leftarrow \min\left\{1, \frac{f(a_i|\phi_{\theta'}(x_i))}{\mu(a_i|x_i)}\right\}$
end for
$Q^{ret} \leftarrow \begin{cases} 0 & \text{for terminal } x_k \\ \sum_a Q_{\theta'_v}(x_k, a) f(a|\phi_{\theta'}(x_k)) & \text{otherwise} \end{cases}$
for $i \in \{k-1, \cdots, 0\}$ **do**
    $Q^{ret} \leftarrow r_i + \gamma Q^{ret}$
    $V_i \leftarrow \sum_a Q_{\theta'_v}(x_i, a) f(a|\phi_{\theta'}(x_i))$
    Computing quantities needed for trust region updating:

$$g \leftarrow \min\{c, \rho_i(a_i)\} \nabla_{\phi_{\theta'}(x_i)} \log f(a|\phi_{\theta'}(x_i)) \left(Q_{\theta'_v}(x_i, a_i) - V_i\right) + \sum_a \left[1 - \frac{c}{\rho_i(a)}\right]_+ f(a|\phi_{\theta'}(x_i)) \nabla_{\phi_{\theta'}(x_i)} \log f(a|\phi_{\theta'}(x_i)) \left(Q_{\theta'_v}(x_i, a_i) - V_i\right)$$

$$k \leftarrow \nabla_{\phi_{\theta'}(x_i)} \mathrm{D}_{\mathrm{KL}}\left[f(\cdot|\phi_{\theta_a}(x_i)) || f(\cdot|\phi_{\theta'}(x_i))\right]$$

    Accumulate gradients wrt θ' : $d\theta' \leftarrow d\theta' + \frac{\partial \phi_{\theta'}(x_i)}{\partial \theta'} \left(g - \max\left\{0, \frac{k^T g - \delta}{\|k\|_2^2}\right\} k\right)$
    Accumulate gradients wrt θ'_v : $d\theta_v \leftarrow d\theta_v + \nabla_{\theta'_v} \left(Q^{ret} - Q_{\theta'_v}(x_i, a)\right)^2$
    Update Retrace target: $Q^{ret} \leftarrow \bar{\rho}_i \left(Q^{ret} - Q_{\theta'_v}(x_i, a_i)\right) + V_i$
end for
Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$
Update the average policy network: $\theta_a \leftarrow \alpha\theta_a + (1-\alpha)\theta$

combined in different ways to achieve higher performance than would be possible with a single model.

Ensemble learning can be used in supervised learning by training multiple classifiers and then aggregating the classifiers' predictions. The bagging predictors method (Breiman, 1996) trains multiple predictors on slightly different versions of the training data set and then combines the predictions of the classifiers by averaging (for numerical predictions) or using a majority vote (for class predictions). Aggregating the predictions in this way improves the accuracy of decision tree and linear regression models substantially.

Boosting algorithms, such as AdaBoost (Freund and Schapire, 1996), run an algorithm multiple times over different distributions of the data to generate multiple models, which are then combined into a single composite model. On each different run, the distribution of the data is changed to emphasise training data that previous models mis-classified. AdaBoost works especially well with "weak" learning algorithms that generate relatively simple classifiers.

Jacobs et al. (1991) approach ensemble learning differently and, instead of training each model on a large portion of the training data, train the different models to be local experts. A gating network is used to generate weights that are fed to a stochastic one-out-of-n selector that determines which expert is assigned to each new case, as shown in Figure 2.3. The experts are only trained on cases that were assigned to them, ensuring that their network weights are decoupled from the weights of the other experts' networks. This causes experts to specialise in different cases and makes this approach suitable for data where cases can be naturally divided into different types.

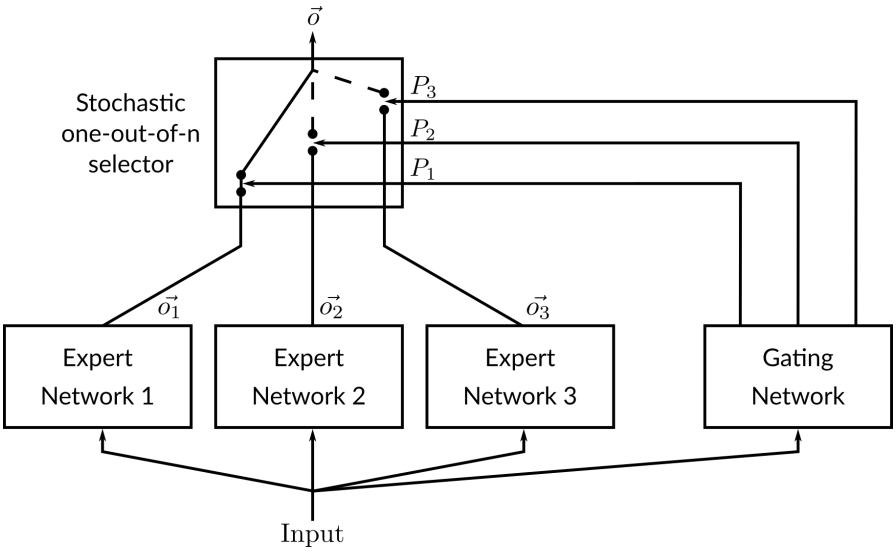


Figure 2.3: Mixture of experts gated by gating network. Adapted from Jacobs et al. (1991).

2.5.1 Ensemble Algorithms in Reinforcement Learning

The ensemble learning methods used in supervised learning can also be extended to RL algorithms. An ensemble of decision trees can be used to solve the batch RL problem by approximating the Q-function and then selecting actions that maximise the Q-value function, as shown by Ernst et al. (2005). The mixture of local experts method (Jacobs et al., 1991) can be extended to

reinforcement learning by subdividing RL problems into elemental MDPs and composite MDPs to achieve learning by sharing information between different experts (agents that specialised on different elemental MDPs) (Singh, 1992). RL can also be used to combine the value estimates of an ensemble of agents by using TD-learning to learn a linear combination of the Q-functions of the different agents (Marivate and Littman, 2013).

Wiering and Van Hasselt (2008) propose different voting methods that can be used to aggregate the output of multiple RL algorithms and generate a single action for the agent to execute.

The different voting methods calculate weights, also called preference values, for the different actions in order to determine which action should be taken. A Boltzmann distribution over the generated preference values, p_t, is used to ensure that the agent still explores (instead of using epsilon-greedy exploration). The distribution over p at timestep t is shown in Equation 2.9, with action i given by $a[i]$, where $a[.]$ is the discrete set of actions available to the agent.

$$\pi_t(s_t, a[i]) = \frac{e^{p_t(s_t, a[i]/\tau)}}{\sum_k e^{p_t(s_t, a[k])/\tau}} \tag{2.9}$$

Majority Voting. The majority voting rule calculates the preference values for n RL algorithms as shown in Equation 2.10, where $I(x)$ is the indicator function and a_t^j is the action selected by algorithm j at timestep t.

$$p_t(s_t, a[i]) = \sum_{j=1}^{n} I(a[i], a_t^j) \tag{2.10}$$

The majority voting method is similar to the bagging predictors method since it also assigns the highest weight to actions that were chosen by many algorithms, but differs from the bagging predictors method by sometimes selecting an action that is not preferred by the ensemble due to the need for exploration.

Rank Voting. The rank voting rule calculates the preference values for the ensemble according to Equation 2.11, where $r_t^j(a[1]), \cdots, r_t^j(a[m])$ denote the weights given to each action according to the probability of selecting each action and r_t^j is the rank assigned to algorithm j's action at timestep t.

$$p_t(s_t, a[i]) = \sum_j r_t^j(a[i]) \tag{2.11}$$

The action with the highest probability is weighted m times, the action with the second highest probability $m-1$ times and so on.

2.6 Learners as black boxes

We take an approach similar to Laroche and Feraud (2018) and share only trajectories between base learners (instead of sharing action-value functions, etc.). This allows the ensemble learner to consider base learners as black boxes and remain agnostic to the inner workings of the base learners.

The interaction process between the environment, the ensemble learner and the base learners is shown in Figure 2.4. The ensemble learner queries its selected learner for an action, given the current state, and then takes that action in the environment. The new experience, a $< s, a, r >$ $-$tuple, is given to each base learner as if they were interacting with the environment directly. From each base learner's perspective, the ensemble learner is effectively transparent (as shown in the figure). Each base learner is then free to use any state representation and method of learning internally.

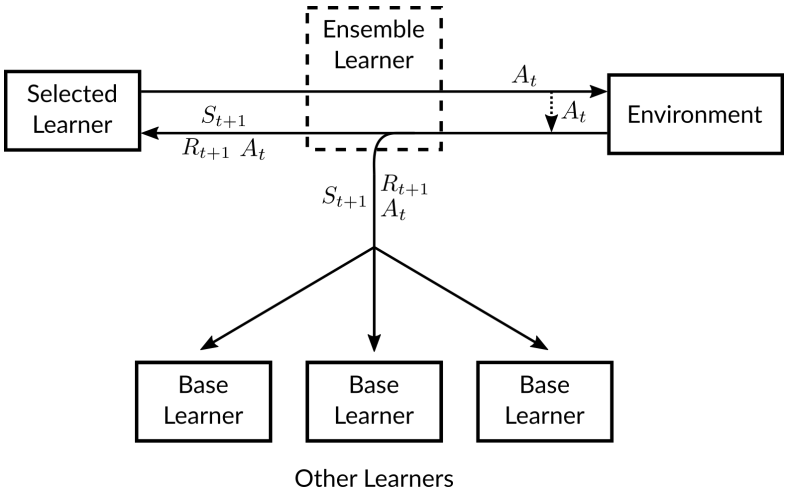


Figure 2.4: Depiction of information sharing between black box learners.

2.7 Summary

In this chapter we defined our problem setting, the ensemble learner setting, showed how it relates to RL and how these ensemble learners interact with the environment and their base learners. We showed how learners can be treated as black boxes, allowing an ensemble learner to stay agnostic to the learning mechanisms and internal representations of its base learners. We introduced k-armed bandit algorithms and showed how previous work solved these problems by using confidence bounds and optimism in the face of uncertainty.

We also introduced the RL problem, the machinery used to model and solve RL problems, as well as different methods of exploring the actions available to an RL agent. We gave an overview of the different types of RL learners, from learners that learn the value function, to learners that learn a model of the world and use that to find an optimal policy, and showed some widely used algorithms of each of these different types of learners.

Finally, we gave an overview of ensemble learning methods in machine learning and showed how ensemble learning methods have been applied to RL in the literature.

Chapter 3

Learning with an Ensemble of Learners

Using an ensemble of RL algorithms or learners can yield improved performance over a single learner as well as better generalisation across tasks (Marivate and Littman 2013; Wiering and Van Hasselt 2008). Existing methods, however, often make assumptions about the representations of the learners in the ensemble (requiring all learners to be Q-learners, for example).

In this chapter we propose an ensemble learner that can use a set of base learners while remaining agnostic to the learners' inner workings. Remaining agnostic to the inner workings of its base learners gives the ensemble learner the ability to use base learners that learn in completely different ways and use different state or action representations. The ensemble learner can use a tabular Q-learner and a DQN-learner together, for example, or a learner that uses a model-based learning approach together with a learner that uses a model-free learning approach. Refer to Section 2.4 for an overview of the different types of RL learners.

The ensemble learner selects different base learners for every episode and then uses the chosen base learner to choose actions to execute during the episode, as depicted in Figure 3.1. Although the ensemble learner only selects a base learner once per episode, it trains every base learner on each timestep during an episode, and as such requires base learners to use off-policy learning algorithms (see Section 2.4.2). Off-policy algorithms are a large class of algorithms and restricting the ensemble learner to the use of off-policy algorithms is therefore not hugely onerous.

We define the following as desirable properties for such an ensemble learner:

1. The ensemble learner should select the best performing base learner with high probability.
2. The performance of base learners is non-stationary during training (since base learners are trained online), and the environment could be non-stationary as well, and therefore the ensemble learner should keep exploring the different base learners throughout training (the probability of selecting the best learner should not become so large that other learners are no longer selected).

We propose the Adaptive Probabilistic Ensemble Learning (APEL) algorithm as an ensemble learner that has the aforementioned properties. The APEL ensemble learner uses a Dirichlet distribution to keep a distribution over the probability of selecting each base learner. The

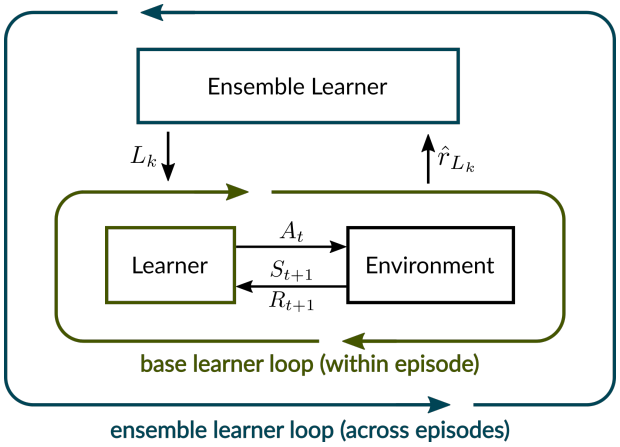


Figure 3.1: Diagram of ensemble learner's interaction with base learners.

concentration parameters of the Dirichlet distribution are initialised as shown in Section 3.1. The Dirichlet distribution is then sampled at the start of each episode to obtain a probability distribution according to which a base learner is selected (see Section 3.2 for details). Next, the Dirichlet distribution's probability mass is shifted to base learners that perform well by increasing their concentration parameters, as described in Section 3.3. Finally, the concentration parameters are decayed each time a base learner is selected (described in Section 3.4). We present a theoretical analysis of the aforementioned concentration parameter decay mechanism, as well as different methods of selecting the decay parameter, in Section 3.5 and 3.6 respectively, and describe the complete algorithm in Algorithm 5.

3.1 Concentration parameters

3.1.1 Dirichlet distribution

The Dirichlet distribution is a distribution over distributions, i.e. sampling from a Dirichlet distribution yields a distribution. The Dirichlet distribution is parametrised by concentration parameters, $\alpha_1, \cdots, \alpha_n = \boldsymbol{\alpha}$ and its mean and variance are given in Equation 3.1 (Kotz et al., 2000).

$$\begin{aligned} \mu &= \frac{\alpha_i}{\sum_{k=1}^{K} \alpha_k} \\ \sigma^2 &= \frac{\bar{\alpha}_i(1-\bar{\alpha}_i)}{\alpha_0+1} \quad \text{where } \bar{\alpha}_i = \frac{\alpha_i}{\alpha_0} \text{ and } \alpha_0 = \sum_{i=1}^{K} \alpha_i \end{aligned} \tag{3.1}$$

3.1.2 Initialisation of concentration parameters

The concentration parameters of the Dirichlet distribution are initialised to 1, i.e. $\boldsymbol{\alpha} = (1, \ldots, 1)$, at the start of training to give all base learners equal probability of being selected. Prior knowledge about the performance of base learners (specifically the performance at the start of training) can be encoded by assigning higher initial values to the concentration parameters corresponding to some of the base learners.

Algorithm 5 Adaptive Probabilistic Ensemble Learning (APEL).

Require: base learners $L_1, \ldots, L_n$, decay parameter ν

1: Initialise performance estimates and concentration parameters for each base learner

$$\hat{\boldsymbol{r}} \leftarrow (null, \ldots, null)$$
$$\boldsymbol{\alpha} \leftarrow (1, \ldots, 1)$$

2: **for** each episode e_k **do**

3: Select base learner L_k ▷ see Section 3.2

$$p_{\text{learners}} \sim Dir(\alpha)$$
$$L_k \sim p_{\text{learners}}$$

4: Generate trajectory τ with cumulative reward R using base learner L

5: Train all base learners on trajectory τ

6: $\hat{r}_{L_k} \leftarrow R$

7: **if** $\hat{r}_{L_k} > \max(\hat{\boldsymbol{r}} \setminus \hat{r_{L_k}})$ **then** ▷ see Section 3.3

8: $\alpha_{L_k} \leftarrow \alpha_{L_k} + 1$

9: **end if**

10: Decay concentration parameters ▷ see Section 3.4

$$\boldsymbol{\alpha} \leftarrow \max(\boldsymbol{\alpha} * (1 - \nu), 1)$$

11: **end for**

3.2 Learner selection

At the start of each episode e_k, the ensemble learner selects a base learner L_k by sampling the distribution over base learners, p_{learners}, obtained by sampling the Dirichlet distribution, as shown in Equation 3.2 and 3.3.

$$p_{\text{learners}} \sim Dir(\boldsymbol{\alpha}) \tag{3.2}$$

$$L_k \sim p_{\text{learners}} \tag{3.3}$$

Selecting base learners in this manner allows the ensemble learner to select good base learners with high probability (which addresses *property 1*), while still allowing the ensemble learner to periodically explore the other base learners (which addresses *property 2*).

3.3 Adjusting concentration parameters

The ensemble learner keeps track of the performance of each base learner separately, by storing the latest performance estimate for each learner, denoted by the vector $\hat{\boldsymbol{r}}$. At the end of each episode, the performance estimate for the currently selected learner $\hat{r}_{L_k}$ is stored and compared to the stored performance estimates of the other learners denoted by $\hat{\boldsymbol{r}} \setminus \hat{r_{L_k}}$. The concentration parameter corresponding to the selected base learner, α_{L_k}, is updated as shown in Equation 3.4 to increase the concentration if L_k outperformed all the other base learners. Increasing the

concentration parameters corresponding to good learners increases the probability of selecting these good learners and allows the ensemble learner to satisfy *property 1*.

$$\alpha_{L_k} = \begin{cases} \alpha_{L_k} + 1 & \text{if } \hat{r}_{L_k} > \max(\hat{\boldsymbol{r}} \setminus \hat{r_{L_k}}) \\ \alpha_{L_k} & \text{if } \hat{r}_{L_k} \leq \max(\hat{\boldsymbol{r}} \setminus \hat{r_{L_k}}) \end{cases} \tag{3.4}$$

The Dirichlet distribution's concentration parameters are not adjusted until all base learners have been selected at least once, to allow the ensemble learner to have an estimate of the performance of each base learner before it starts to adjust the probability of selecting learners. This is similar to multi-armed bandit algorithms, such as the UCB algorithm (Auer et al., 2002), where the algorithm is initialised by playing each arm once.

In this work we use the cumulative reward obtained by a base learner over an entire episode as the performance of the learner, but other metrics (such as episode length) could also be used.

3.4 Decaying concentration parameters

On tasks where the agent needs to train for a long time to reach optimal performance (most non-trivial tasks), the agent experiences a large number of episodes and this causes the concentration parameters of the Dirichlet distribution to grow without bound, which in turn causes the ensemble learner to become overly confident in a particular base learner (preventing it from still periodically exploring the other base learners). This is a form of overfitting, where the ensemble learner overfits to the current performance of the base learners. This can be problematic if the performance of the base learners changes over time.

The large concentration parameters and subsequent overconfidence is curbed by decaying the concentration parameters slightly, according to Equation 3.5, each time after selecting a new base learner (at the start of every episode), with $0 \leq \nu < 1$.

$$\alpha = \max(\alpha * (1 - \nu), 1) \tag{3.5}$$

Equation 3.5 also caps the concentration parameters at a minimum value of 1, to ensure that the probability of selecting any of the underperforming base learners does not become vanishingly small, thereby ensuring that these underperforming learners are still explored periodically.

Decaying the concentration parameters has multiple benefits. It limits the maximum value of the concentration parameters and also decreases the rate at which the concentration parameters grow slightly, thereby keeping the ensemble learner from becoming too confident too quickly in the base learner it has selected. The concentration parameter decay also allows the ensemble learner to "forget" old evidence, by essentially assigning more weight to more recent observations of base learner performance and less to older observations.

The concentration decay parameter, ν, can be used to trade off the speed at which the ensemble learner can adapt to changes in base learner performance or environment dynamics with the performance of the ensemble learner. A higher ν allows the ensemble learner to more quickly adapt to changes in base learner performance, while lowering the average performance of the ensemble learner (since suboptimal base learners are selected more frequently).

3.5 Evolution of concentration parameters

In this section we present a theoretical analysis of the evolution of the concentration parameters of APEL's Dirichlet distribution over time. For this analysis, we assume, without loss of generality, that there are n learners with only one optimal learner, $L_* = L_1$, which always receives the highest reward of all the base learners. We also assume that the environment's rewards are deterministic. L_*'s concentration parameter is incremented at each ensemble learner timestep e_n (since it always performs the best) and the concentration parameters of the other learners $(L_2, \cdots, L_n)$ are never incremented.

3.5.1 Incremental updates with decay

The recursive update rule for the concentration parameter of L_* is shown in Equation 3.6. Since the concentration parameter is constantly increasing in this case, we omit the $\max(\cdots, 1)$ operator of Equation 3.5 here.

$$\alpha_{k+1} = (\alpha_k + 1) \cdot (1 - \nu) \tag{3.6}$$

This equation can be written in explicit form as shown in Equation 3.7.

$$\alpha_k = (1 - \nu)^k \cdot (\alpha_0 + 1) + \sum_{i=1}^{k-1} (1 - \nu)^i \tag{3.7}$$

The steady-state (maximum) value of the concentration parameter α_{ss} is determined by evaluating Equation 3.7 as $k \to \infty$.

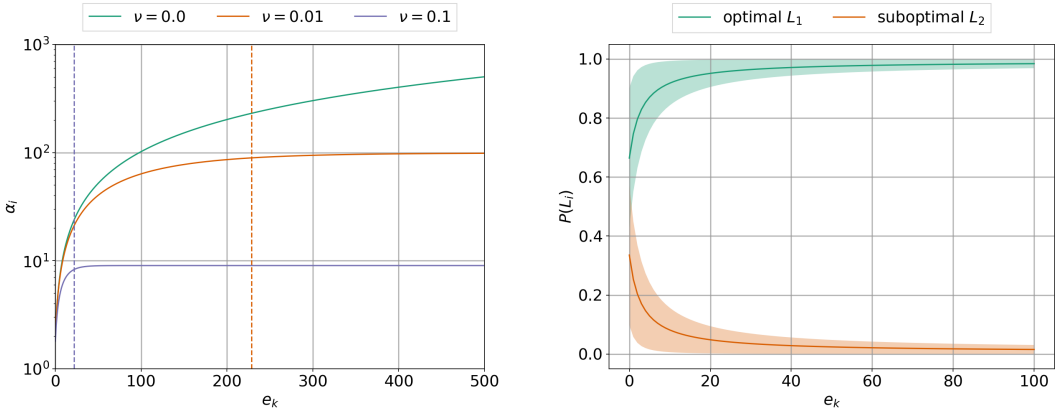
$$\begin{aligned} \alpha_{ss} &= \lim_{k \to \infty} \alpha_k = \lim_{k \to \infty} \left((1 - \nu)^k \cdot (\alpha_0 + 1) + \sum_{i=1}^{k-1} (1 - \nu)^i \right) \\ &= 0 + \sum_{i=1}^{\infty} (1 - \nu)^i \end{aligned} \tag{3.8}$$

Since $0 < (1 - \nu) < 1, \forall \nu \in (0, 1)$, the infinite series in Equation 3.8 converges, as shown in Equation 3.9, for any ν in the range $(0, 1)$.

$$\begin{aligned} &\sum_{i=0}^{\infty} r^i = \frac{1}{1 - r} \\ &\therefore \sum_{i=1}^{\infty} r^i = \frac{1}{1 - r} - 1 \\ &\therefore \sum_{i=1}^{\infty} (1 - \nu)^i = \frac{1}{1 - (1 - \nu)} - 1 \\ &\therefore \alpha_{ss} = \lim_{k \to \infty} \alpha_k = \frac{1}{\nu} - 1 \end{aligned} \tag{3.9}$$

Figure 3.2a shows how the concentration parameter grows until it reaches its steady-state value for different values of ν. The time to reach 90% of the steady-state value is shown by dashed

vertical lines. When $\nu = 0$, there is no decay and the concentration parameter's growth is unbounded. When $\nu \neq 0$, the concentration parameter's growth is bounded and the steady-state value, as well as the time it takes to reach this value, becomes smaller as ν becomes larger.



(a) Value of constantly increased concentration parameter as a function of episodes for different values of ν.

(b) Evolution of probability of selecting base learners for $\nu = 0.01$ with variance of distributions shown by shaded regions.

Figure 3.2: Progression of concentration parameters and the subsequently generated distributions.

The concentration parameters can be used to calculate the means and variances of the probability distributions generated by the Dirichlet distribution using Equation 3.1. The evolution of the means and variances of the generated distributions are shown in Figure 3.2b. The probability of selecting the optimal learner becomes larger as time progresses and the variance of the generated distribution also decreases. Conversely, the probability of selecting the suboptimal learner becomes smaller as time progresses, but the variance of the generated distribution also decreases.

3.5.2 Decay only

The recursive and explicit update rules for the concentration parameters of L_i where $L_i \neq L_*$ are shown in Equation 3.10

$$\alpha_{k+1} = \begin{cases} \alpha_k \cdot (1-\nu) & \text{if } \alpha_k \geq 1 \\ 1 & \text{if } \alpha_k < 1 \end{cases} \tag{3.10}$$
$$\alpha_k = \max\left(\alpha_0 \cdot (1-\nu)^k, 1\right)$$

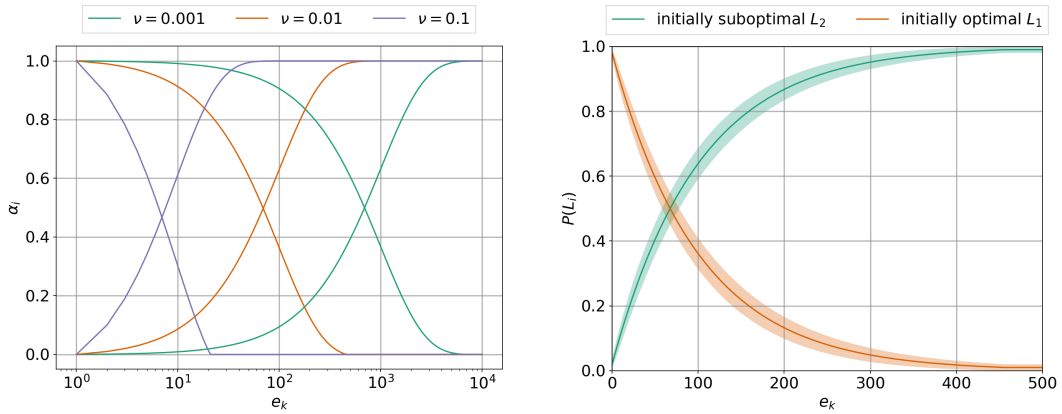
The steady-state value in this case is defined (by the max operator) as 1.

3.5.3 Adapting to changing base learner performance

In this section, we analyse the effect of changing the optimal learner part-way through training. In other words, after APEL has already settled on the optimal learner (the concentration parameters have reached their steady-state values), the optimal learner is changed from $L_* = L_1$ to $L_* = L_2$.

Since APEL keeps on exploring the suboptimal learners, it will observe that L_2 now performs better than L_1 and shift the probability mass to L_2.

In this case, L_1 uses the decay-only update rule (Section 3.5.2) and starts with α_0 equal to its steady-state value. L_2, on the other hand, starts with $\alpha_0 = 1$ and uses the increment-and-update update rule (Section 3.5.1). Figure 3.3a shows how the concentration parameters evolve after switching the optimal base learners and following the new update rules. The curves that start with $\alpha = 0$ correspond to L_2 and the curves that start with $\alpha = 1$ correspond to L_1. The evolution of the mean and variance of the distributions sampled according to the concentration parameters over time is shown in Figure 3.3b.



(a) Normalised concentration parameters. Each pair of curves with the same colour correspond to a specific value of ν.

(b) Mean and variance (shown by shaded regions) of distributions sampled according to concentration parameters.

Figure 3.3: Evolution of APEL's values after optimal base learner has been switched.

3.6 Selecting the decay parameter

The decay parameter is integral to APEL's behaviour and characteristics, and can be tweaked to achieve different results. Here we show four different criteria that can be used to choose the decay parameter's value:

1. The time it takes APEL to adapt to changes in base learner performance.
2. The rise time of the concentration parameters.
3. The steady-state exploration probability.
4. The steady-state performance.

3.6.1 Decay according to adaptation time

APEL's ability to adapt timeously to changes in base learner performance is governed by the decay parameter. Assuming that APEL's concentration parameters have reached their steady-state values (APEL has settled on the base learner it thinks is optimal), the time it takes the mean probability of selecting each base learner to become equal again is given by solving Equation 3.11

for k. The initially optimal learner's concentration parameter is given by α_{0_1} and the initially suboptimal learner's concentration parameter is given by α_{0_2}.

$$\begin{aligned} \alpha_{0_1} \cdot (1-\nu)^k &= (1-\nu)^k \cdot (\alpha_{0_2}+1) + \sum_{i=1}^{k-1}(1-\nu)^i \\ \left(\frac{1}{\nu}-1\right) \cdot (1-\nu)^k &= (1-\nu)^k \cdot (1+1) + \sum_{i=1}^{k-1}(1-\nu)^i \end{aligned} \tag{3.11}$$

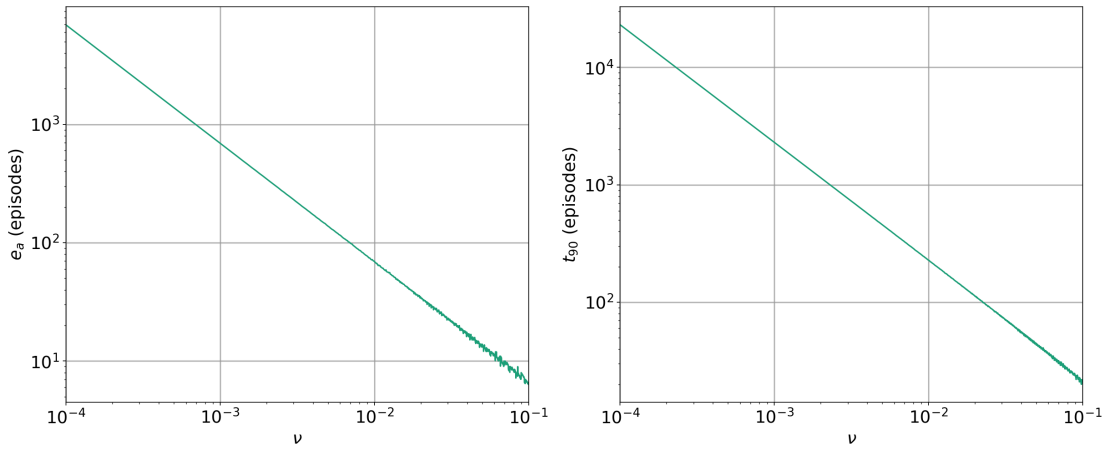
The decay parameter can be selected according to the desired adaptation time. Figure 3.4a shows the adaptation time as a function of ν.

3.6.2 Decay according to rise time

The rise time of a concentration parameter, t_{90}, is the number of episodes it takes the concentration parameter to reach a certain percentage of the steady-state value (90% in this case). The rise time gives an indication of how quickly APEL becomes confident that any given learner is the optimal base learner. The rise time of the concentration parameter can be determined for any value of $\nu \in (0, 1)$ by solving Equation 3.12 numerically for k.

$$\begin{aligned} 0.9 \cdot \alpha_{ss} &= (1-\nu)^k \cdot (\alpha_0+1) + \sum_{i=1}^{k}(1-\nu)^i \\ \therefore 0.9 \cdot \left(\frac{1}{\nu}-1\right) &= (1-\nu)^k \cdot (\alpha_0+1) + \sum_{i=1}^{k}(1-\nu)^i \end{aligned} \tag{3.12}$$

The decay parameter, ν, can be selected according to how quickly APEL should become confident that a given base learner is the optimal base learner. Figure 3.4b shows the rise time as a function of ν.



(a) Adaptation time.

(b) Rise time.

Figure 3.4: Concentration parameter metrics as a function of the decay parameter.

3.6.3 Decay according to steady-state exploration probability

Assuming that APEL eventually settles on one learner, that learner's concentration parameter will tend to its steady-state value in the limit (Equation 3.9) and the other learners' concentration parameters will tend to 1 in the limit (Equation 3.10). Substituting these steady-state values into the Dirichlet distribution's mean gives μ_{L_*} (the mean of the distribution generated for L_*) as a function of ν and rearranging gives ν as a function of μ_{L_*} (Equation 3.13), where n is the number of base learners.

$$\begin{aligned} \mu_{L_*} &= \frac{1/\nu - 1}{1/\nu - 1 + \sum_1^{n-1} 1} \\ \therefore \nu &= \frac{\mu_{L_*} - 1}{\mu_{L_*} \cdot (1 - \sum_1^{n-1} 1) - 1} \end{aligned} \tag{3.13}$$

The mean values of the distributions generated by the Dirichlet distribution govern how often each base learner is selected on average. Due to *property 2*, APEL should still select non-optimal learners in the steady-state case in order to explore the non-optimal learners. The decay parameter ν can be chosen, according to Equation 3.13, by defining the amount of exploration that APEL should still do after it has settled on the optimal base learner.

3.6.4 Decay according to steady-state performance

Due to APEL's continued periodic exploration of suboptimal base learners, the average steady-state performance is slightly lower than the performance of the optimal base learner. This section shows how ν can be chosen to give a minimum level of steady-state performance (on average).

Continuing with the case where there is one optimal base learner L_* and APEL settles on this base learner in the steady-state, we can write APEL's expected return $E[G]$ as the sum of the returns of the individual base learners, weighted by their probabilities of selection ($E[P(L_*)]$ and $E[P(L_i)]$) as shown in Equation 3.14. The probability of selecting L_* and $L_2, \cdots, L_n$ respectively can be determined by Equation 3.1 by noting that α_{L_*} will be equal to its steady-state value and the other concentration parameters will be equal to 1.

$$\begin{aligned} E[G] &= E[P(L_*)] \cdot G_{L_*} + \sum_{i=2}^{n} E[P(L_i)] \cdot G_{L_i} \\ &= \frac{\alpha_{L_*}}{\alpha_{L_*} + (n-1)} \cdot G_{L_*} + \frac{1}{\alpha_{L_*} + (n-1)} \cdot \sum_{i=2}^{n} G_{L_i} \\ &= \frac{1/\nu - 1}{1/\nu - 1 + (n-1)} \cdot G_{L_*} + \frac{1}{1/\nu - 1 + (n-1)} \cdot \sum_{i=2}^{n} G_{L_i} \end{aligned} \tag{3.14}$$

Re-arranging Equation 3.14 yields an expression for ν as a function of the returns of the base learners (G_{L_*} and G_{L_i}) and the desired minimum steady-state performance $E[G]$. This expression, shown in Equation 3.15, allows the selection of ν according to the desired minimum average

steady-state performance.

$$\nu = \frac{G_{L_*} - E[G]}{G_{L_*} + E[G] \cdot (n-2) - \sum_{i=2}^{n} G_{L_i}} \tag{3.15}$$

3.7 Summary

In this chapter we presented the APEL algorithm, an ensemble learner capable of using a set of different base learners and selecting the best performing learner with high probability, while remaining agnostic to the inner workings of the base learners. We use a Dirichlet distribution to allow APEL to select between the base learners probabilistically and show how the Dirichlet distribution's concentration parameters are updated and decayed to prevent APEL from overfitting to the current base learner performance.

Furthermore, we presented a theoretical analysis of our parameter decay method and showed how the concentration parameters (and subsequently the distributions sampled from the Dirichlet distribution) evolve over time as the concentration parameters are updated.

Lastly, we showed different metrics that can be used to choose the decay parameter and derived equations to allow a designer to calculate the value of the decay parameter given their specifications.

Chapter 4

Experiments

In this chapter, we present our experimental setup and analyse APEL qualitatively and quantitatively. We also compare the theoretical results of Section 3.5 and 3.6 to the experimental results in this section.

The qualitative analysis (Section 4.2) shows how APEL behaves and how its parameters change throughout training and compares the results to the theoretical analysis. We also demonstrate APEL's ability to leverage the strengths of multiple base learners online on the Atari Breakout domain.

In the quantitative analysis (Section 4.3) we evaluate APEL's ability to outperform any individual base learner on average when run on multiple tasks and its ability to adapt to changing base learner performance, and compare the results to the Sliding Stochastic Bandit Algorithm Selection (SSBAS) algorithm (Laroche and Feraud, 2018) (the most similar related method that also selects between different base learners while remaining agnostic to the internal mechanics of the base learners). We also analyse the effect that the decay parameter has on APEL's steady-state performance and adaptation time as well as the effect that adding many additional base learners has on APEL's performance. Finally, we compare APEL to majority voting ensembles and show the hyperparameter sweep used to choose values for APEL and SSBAS's hyperparameters. We perform the aforementioned evaluations on a maze domain and the Cartpole domain (described in Section 4.1).

4.1 Experimental setup

4.1.1 Maze domain

The maze domain is a 10×10 grid world that has randomly generated walls that form a perfect maze (a maze with only one path between any two points). The state space consists of the agent's position (x, y). The starting position and goal location of each maze is also randomised while ensuring that the distance between the two points is at least 80% of the distance between the maze's two diagonal corners. Episodes have a maximum length of 1000 timesteps and the randomly generated goal location is the only terminal state.

Examples of generated mazes, along with the generated starting point and goal location and the shortest path between them, are shown in Figure 4.1.

The experiments that use the maze domain evaluate train time (the number of episodes it takes an agent to reach optimal performance) denoted as e_T. Concretely, the agent is done training when the mean length of the last 50 episodes is below $1.1 \times$ optimal length, where optimal length is the minimum number of steps required by the agent to move from the start position to the goal location.

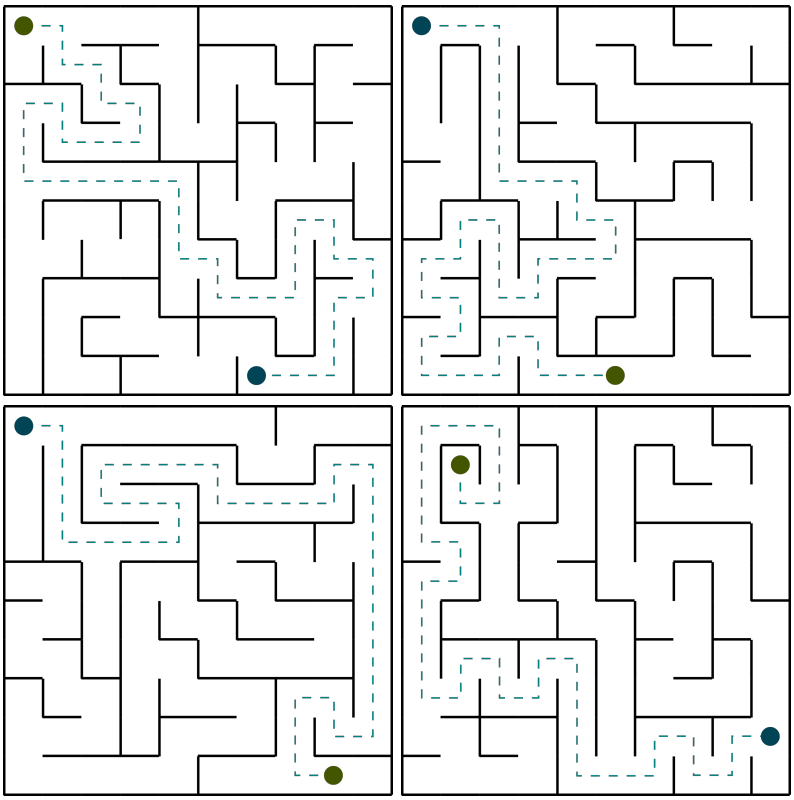


Figure 4.1: Examples of mazes generated by the maze environment. The agent starting locations and goal locations are shown by blue and green dots respectively.

4.1.2 Cartpole domain

The Cartpole domain is a classic control problem described by Barto et al. (1983) that consists of a pole that is attached, by an un-actuated joint, to a cart that can slide horisontally on a rail, as shown in Figure 4.2. The agent's goal is to keep the pole in an upright position without moving the cart too far to either side of the starting position.

The agent can observe the following continuous variables:

- Cart position
- Cart velocity
- Pole angle
- Pole angular velocity

The agent can apply one of two discrete actions at each time step: push cart to the left or push cart to the right.

The agent receives +1 reward for each timestep that the pole remains upright and the episode ends when the cart is more than 2.4 units away from the center or the pole falls to more than 15 degrees from vertical. The domain is considered solved when the agent obtains an average reward of at least 195.0 over 100 consecutive episodes.



Figure 4.2: Screenshots of the Cartpole environment in different states.

4.1.3 Breakout domain

The standard OpenAI Gym Atari Breakout environment (Brockman et al., 2016) is used, as well as a version of the environment with a handcrafted state space and reward to allow a tabular Q-learner to be able to learn to play Breakout (refer to Section 4.2.3 for a more detailed description).

In Breakout, the player (or agent) controls a paddle at the bottom of the screen and there are bricks that can be broken at the top of the screen. When the game is started, a ball is placed somewhere above the paddle and starts falling. The player (or agent) then needs to manoeuvre the paddle in order for the ball to bounce on the paddle. The goal of the game is to get points by bouncing the ball in such a way that it breaks as many bricks at the top of the screen as possible. Screenshots of the Atari Breakout environment are shown in Figure 4.3.

4.1.4 Random seeds

The results of each experiment are averaged over a number of different runs. These different runs each use a different seed. Different seeds are obtained by seeding Python's random number generator with the seed 1234 and then generating seeds using the random package's randint function.

4.2 Qualitative experiments

4.2.1 Selection behaviour with two base learners

In this experiment, we investigate the way APEL's parameters change over time as it trains. We use the maze domain in this experiment. In each run, we initialise two Q-learners: Q_{good} which is a normal Q-learner with the full state space and Q_{bad} which is a Q-learner with a reduced state

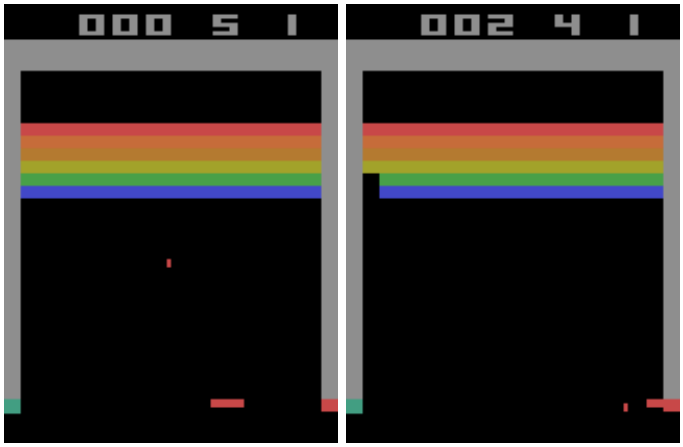


Figure 4.3: Screenshots of the Atari Breakout environment in different states.

space that excludes the y state variable, rendering it unable to learn to perform the navigation task. The concentration parameters of Q_{good} and Q_{bad} are denoted as α_1 and α_2 respectively. This setup closely mimics the setup in the theoretical analysis of Section 3.5.1 where there is one optimal base learner that always performs better than the other base learner(s).

Figure 4.4 shows the reward obtained by the APEL learner as well as the concentration parameters α_i of the Dirichlet distribution and the probability $P(L_i)$ of selecting each base learner sampled from the Dirichlet distribution. The results are averaged over 100 runs.

The concentration parameter α_1 rises monotonically, as in the theoretical analysis, since Q_{good} always performs better than Q_{bad}. The rate at which α_1 increases is, however, smaller than in the theoretical case since, unlike the theoretical case that increases α_1 after every episode, the practical implementation of the algorithm only increases the parameter of the learner that was selected for each episode. In other words, when Q_{bad} is selected (due to random exploration) and APEL observes that it performs worse than Q_{good}, α_1 is not incremented. Despite the lower rate of increase of α_1, the sampled probabilities quickly become polarised, with high probability assigned to Q_{good} and low probability assigned to Q_{bad}. The curves show moderately high variance at the start of training because APEL sometimes selects Q_{bad} a few times consecutively before correcting itself.

4.2.2 Adaptation behaviour with two base learners

In this experiment, we investigate the way APEL reacts when the optimal base learner changes partway through training. We use the maze domain with two Q-learners as base learners. We simulate a change in optimal base learner by varying the amount of exploration done by the two learners by changing their exploration parameters, ε, at the switch point, e_s, and keeping them fixed otherwise. The base learners, Q_1 and Q_2, start with $\varepsilon_1 = 0.4, \varepsilon_2 = 0.8$ and then at e_s the second learner's exploration parameter changes to $\varepsilon_2 = 0.01$ with the expected effect that Q_2's performance will increase and exceed Q_1's performance due the reduction in the amount of exploration it performs. This setup closely mimics the theoretical analysis of Section 3.5.3.

Figure 4.5 shows the concentration parameters α_i of the Dirichlet distribution along with the probability $P(L_i)$ of selecting each base learner sampled from the Dirichlet distribution. The

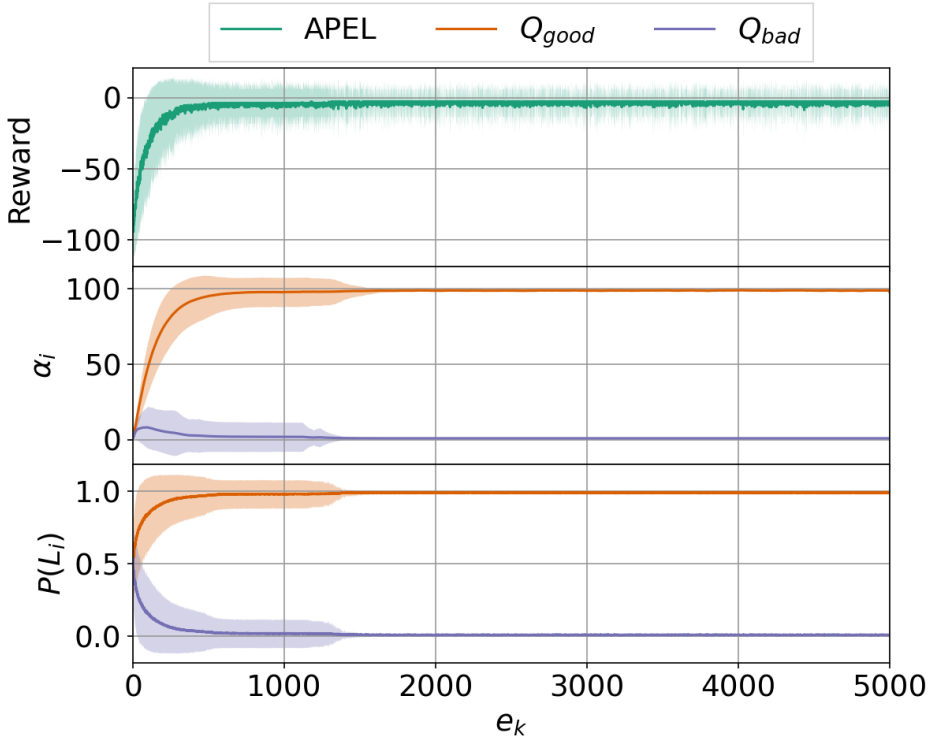


Figure 4.4: Curves showing the change in APEL's parameters over time for $\nu = 0.01$ with variances shown by shaded regions.

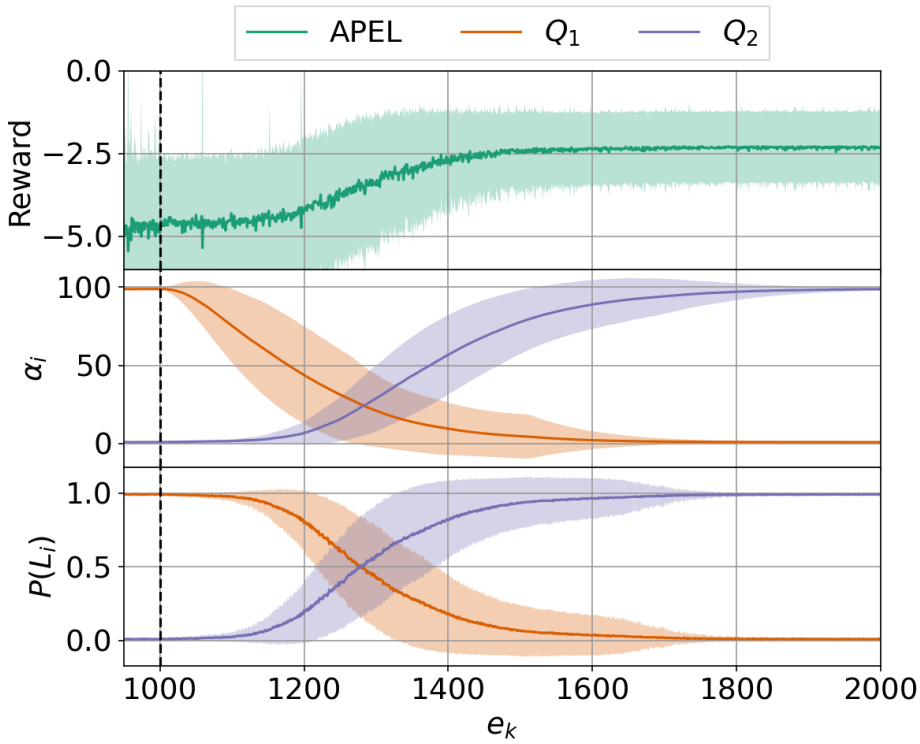


Figure 4.5: Curves showing the change in APEL's parameters after switching the optimal base learner for $\nu = 0.01$. The switch point is indicated by the dashed vertical line and the variances of the curves are shown by shaded regions.

figure also shows the reward obtained by the APEL learner. The switch point was set as $e_s = 1000$. The results are averaged over 100 runs.

The reward curve shows the initial steady-state reached due to APEL's selection of the initially optimal (but actually suboptimal) learner that later increases to a higher level once APEL starts selecting Q_2 after it becomes the optimal learner. The concentration parameters start at their steady-state values. Once the switch takes place, Q_2 will have superior performance to Q_1, but APEL remains ignorant of this fact and keeps on selecting Q_1 until it randomly selects Q_2 through exploration and discovers that its performance is now better than Q_1's performance. At this point, α_1 starts decaying quickly (since it is no longer incremented at all but is still decayed after every episode), but α_2 only starts to increase slowly because it is only increased when Q_2 is selected for an episode and, since Q_2's probability of being selected is initially low, it is not selected often until later when α_2 has grown somewhat. The fact that Q_2 initially has a low probability of being selected causes α_2 to increase slower than in the theoretical case. Section 4.3.2 presents a more in-depth investigation of the decay parameter's effect on the adaptation time.

4.2.3 Leveraging the strengths of multiple learners online

Different learning algorithms have different learning characteristics. Some might learn quickly, but not reach high steady-state performance, while others might take longer to learn but learn better behaviour eventually. An ensemble learner should not only select the best base learner in the limit, but also in an online fashion, combining the strengths of different base learners.

In this experiment, we evaluate APEL's ability to optimise online performance by selecting the best base learner throughout training, not just in the limit. APEL's performance is compared to its base learners' performance throughout training to determine if APEL achieves the same performance as the best base learner throughout training.

We use the Breakout domain with two base learners: a Q-learner (a learner that reaches mediocre performance quickly) and a DQN-learner (a learner that reaches good performance, but takes longer to reach that performance). The DQN-learner uses the normal (pixel) representation of Breakout and the Q-learner uses a handcrafted representation. The handcrafted reward that the agent receives is +1 when the paddle hits the ball and 0 otherwise (as opposed to Breakout's normal reward of +1 when a brick is broken and 0 otherwise). The handcrafted state space has the following state variables: ball position relative to the paddle (left, above left, directly above, above right, and right), ball direction (each of the diagonal directions), and whether the ball is in the upper or lower half of the screen. This handcrafted representation, along with hyperparameters that are tweaked specifically for the Breakout domain, is used to simplify learning for the Q-learner.

The aforementioned handcrafted representation simplifies learning for the Q-learner by drastically reducing the state space of the problem, while still being informative enough for the task to be solvable.

There is a relatively large delay between the agent performing the correct actions (moving the paddle in such a way that it hits the ball) and the agent receiving reward (due to a brick breaking after the ball has moved all the way up from the paddle to the brick). This makes it difficult for the agent to learn which actions are the correct actions, since the actions that it takes just before receiving the reward are actually irrelevant, because the ball has already been hit and moving the paddle after the ball has been hit has no effect on whether a brick will be broken or not. Giving the Q-learner the reward when the ball hits the paddle, instead of when a brick is broken, eliminates the delay and makes it much easier for the Q-learner to learn which actions are good, since the good actions now directly precede the reward.

Importantly, we show here that APEL is able to select between base learners that are completely different, i.e. not just two instantiations of the same algorithm using different hyperparameters, but two different algorithms with different state spaces and learning mechanisms. We also compare APEL's learning curves to the learning curves of SSBAS (the most similar related method that also selects between different base learners while remaining agnostic to the internal mechanics of the base learners) when using the same base learners.

The following hyperparameters are used (see Section 4.3.6 for details on how the hyperparameters were selected):

- APEL: $\nu = 0.01$
- SSBAS: $\xi = 4.921$
- DQN: The OpenAI baselines (Dhariwal et al., 2017) DQN implementation is used and the

default hyperparameters (for Atari) are used except for the exploration time and exploration fraction, which are 15e6 and 0.5 respectively.

- Q-learner: $\varepsilon = 0.9$, $\varepsilon_{decay} = 7500$, $\alpha = 0.05$, $\gamma = 0.9$.

The performance (mean 100 episode reward) of both base learners, APEL and SSBAS is shown throughout training in Figure 4.6. This figure also shows APEL's concentration parameters (α_Q and α_{DQN}) and the probability of selecting each of its base learners throughout training. The results are averaged over 10 runs.

The learning curves show that the handcrafted Q-learner reaches mediocre performance very quickly, but then never learns to perform better. The DQN-learner, on the other hand, takes some time to reach the same level of performance as the handcrafted Q-learner, but its performance eventually greatly exceeds that mediocre level of performance.

APEL's learning curve follows the handcrafted Q-learner for the first part of training (where the handcrafted Q-learner performs better) and then follows the DQN-learner for the second part of training (where the DQN-learner performs better), showing that APEL is able to leverage the strengths of multiple base learners online.

There is a small delay between the DQN-learner starting to perform better than the handcrafted Q-learner and APEL selecting the DQN-learner instead of the handcrafted Q-learner. This delay is caused by APEL being relatively certain that the handcrafted Q-learner is the best base learner, as can be seen by the high value of α_Q and its high probability of being selected. Before the probability of selecting the DQN-learner can increase, α_Q first needs to decrease and α_{DQN} needs to increase, and this is slightly delayed because Q_{DQN} first needs to be randomly explored by APEL before APEL will start increasing α_{DQN}. The delay is relatively small, however, and confirms Section 4.3.2's results: APEL is able to adapt to changes in base learner performance relatively quickly even after having selected one base learner for a long time.

SSBAS's performance is similar to APEL: it also starts out by selecting the handcrafted Q-learner and then later switches to the DQN-learner, but it takes much longer to switch to the DQN-learner once it starts outperforming the handcrafted Q-learner. This additional delay before SSBAS switches to the DQN-learner indicates a phenomenon that we describe in more detail in Section 4.3.2.

4.3 Quantitative experiments

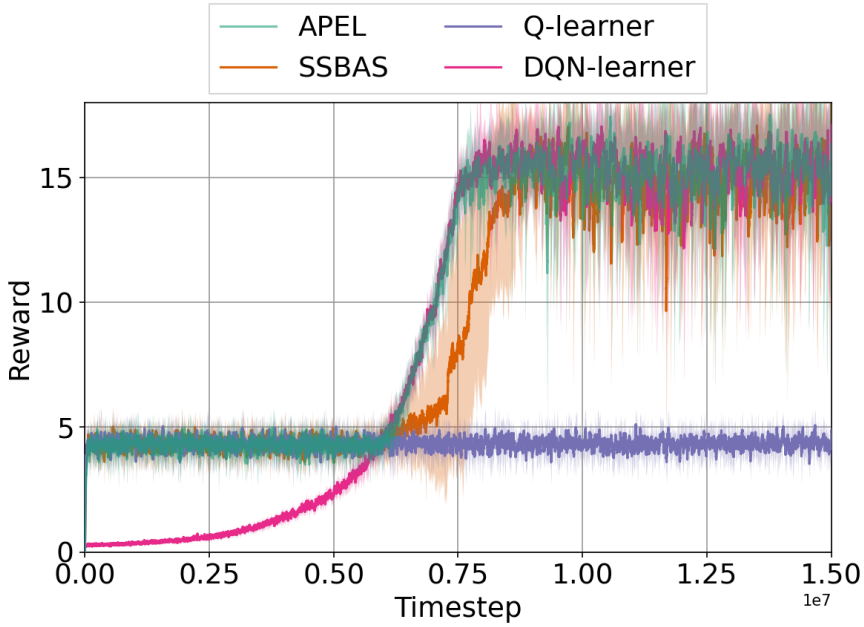
4.3.1 Outperforming individual learners on average

Many tasks can be solved, to varying degrees, with different algorithms, each of which have different benefits and drawbacks. A simpler algorithm might be able to solve a task faster in some cases, but might fail to solve the task in other cases, where a more complex model might always be able to solve the task, but take longer to do so.

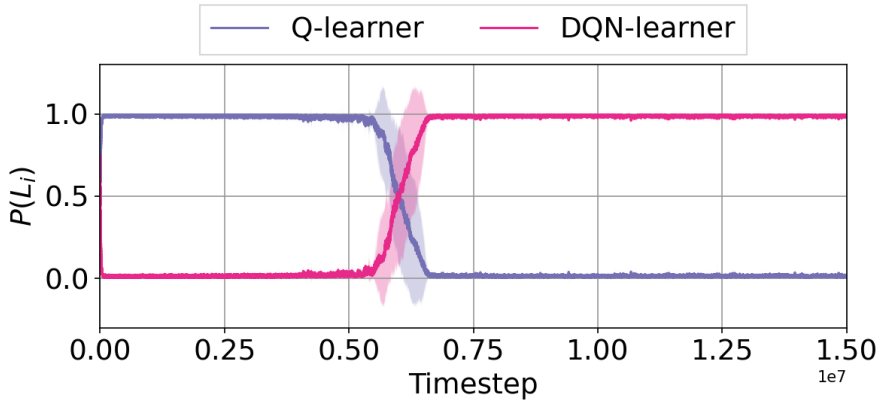
Average performance with a learner that randomly performs well or fails

In this experiment, we evaluate APEL's ability to outperform each individual base learner, on average, when one of the base learners can either solve the task quickly or not at all and the other base learner can always solve the task, but takes longer to converge.

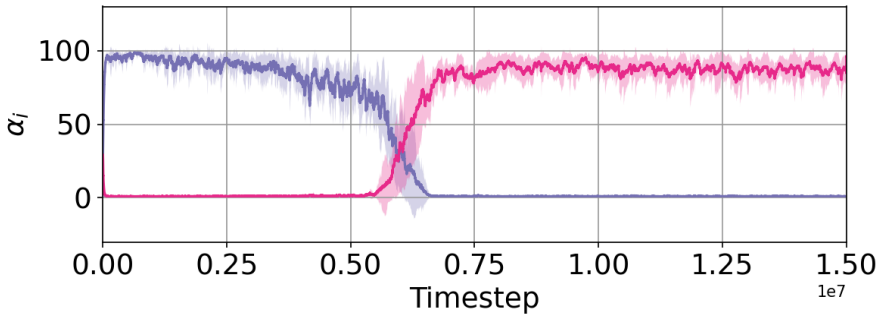
We use the maze domain, augmented with one 10 dimensional random distractor variable. In each run, we initialise two Q-learners: Q_{full} with the full state space and Q_{red} with a reduced state



(a) Reward curves for different learners



(b) Probability of APEL selecting each base learner



(c) APEL's concentration parameter for each base learner

Figure 4.6: Learning curves of different learners on Breakout domain.

space which excludes one of the state variables. If the excluded variable is a distractor variable, then Q_{red} will solve the task faster than Q_{full}, but will otherwise be unable to solve the task. We constrain the generated mazes to have optimal trajectory lengths of between 33 and 35 to reduce the variance introduced by differences in the randomly generated optimal trajectories.

We also compare against SSBAS, a related method that, like APEL, also selects between different base learners online while also remaining agnostic to the inner workings of the base learners. SSBAS frames the learner selection problem as a bandit problem and uses the UCB method to trade off the exploration and exploitation of base learners. Refer to Section 5.1 for a more detailed description of the SSBAS algorithm.

The following hyperparameters are used (see Section 4.3.6 for details on how the hyperparameters were selected):

- Q-learners: $\alpha = \varepsilon = 0.1$, $\gamma = 1$
- APEL: $\nu = 0.01$
- SSBAS: $\xi = 4.921$

The average training times for the different learners (reduced Q-learner, normal Q-learner, APEL and SSBAS) are shown in Figure 4.7 (refer to Section 4.1.1 for details on how train time and optimal performance are defined). The results are averaged over 100 runs. In this experiment, the learners are only allowed to train for a maximum of $5,000$ episodes. If optimal performance is not reached in this time, the train time is set to the maximum train time of $5,000$ episodes.

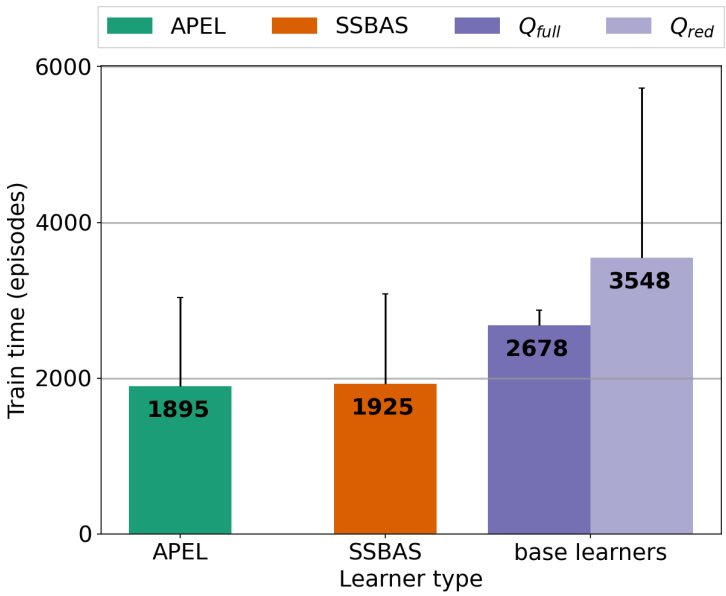


Figure 4.7: Mean train time for different learners, with error bars showing standard deviation.

Figure 4.7 shows that both ensemble learners (APEL and SSBAS) are able to outperform either of the base learners on average. The fact that APEL outperforms both of the base learners on average proves that it selects the best learner on average (selecting Q_{red} when it is able to solve the task and otherwise selecting Q_{full}). The APEL learner also performs better (on average) than the SSBAS learner.

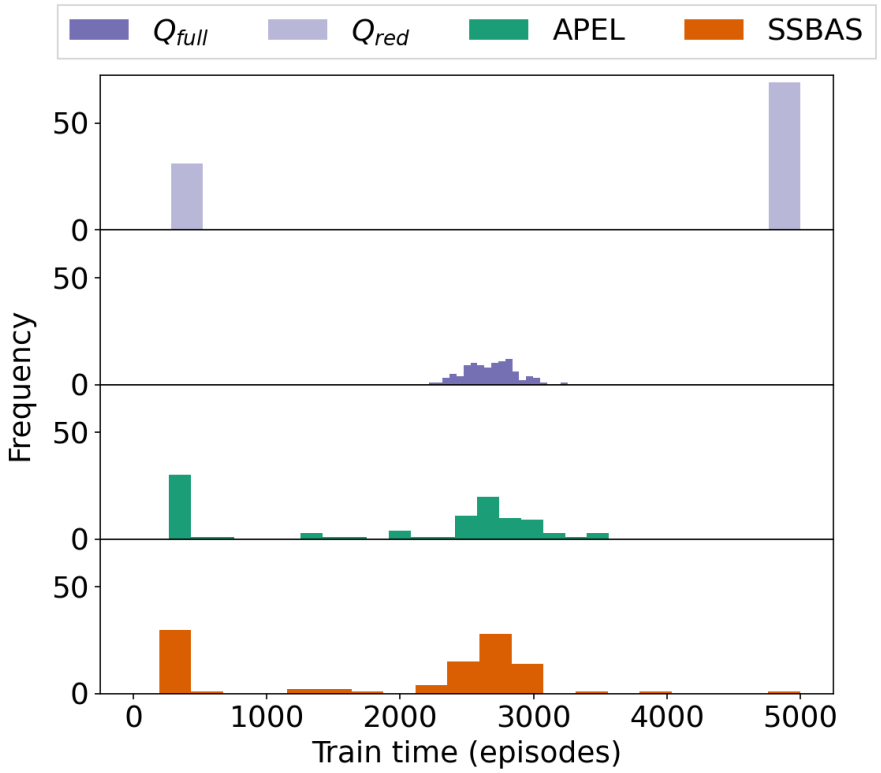


Figure 4.8: Distribution of train times for different learners.

The distributions of the train times of the different learners are shown in Figure 4.8. The figure clearly shows that Q_{red} either learns to complete the task very quickly, or not at all. On the other hand, Q_{full} always learns to complete the task, but never as fast as Q_{red} (when it succeeds). The fact that none of the ensemble learners have any runs that were $5,000$ episodes long shows that they all select Q_{red} when it is able to complete the task and select Q_{full} when Q_{red} cannot complete the task.

Average performance with multiple learners over different environments

In this experiment, we show that APEL reduces an RL practitioner's design burden when training RL algorithms on new problems. Without prior knowledge about the different RL algorithms and the problem to be solved, an RL practitioner would not know which algorithm to use on a new problem. This experiment evaluates APEL's ability to select the best algorithm while training when given a set of different base learners and faced with different problems.

APEL is given three very different learners as base learners: a Q-learner (which uses a tabular value function representation), a DQN-learner (which uses a neural network to approximate the value function) and an ACER-learner (which uses the actor-critic framework with function approximation). In each run of the experiment, the agent is randomly placed in one of two domains: the maze domain or the Cartpole domain.

The following hyperparameters are used (see Section 4.3.6 for details on how the hyperparameters were selected):

- Q-learner: $\alpha = \varepsilon = 0.1, \gamma = 1$.
- DQN-learner: default hyperparameters used by OpenAI baselines implementation.
- ACER-learner: $\alpha = 2e-4, n_{hidden} = 256, \gamma = 0.98$.
- APEL: $\nu = 0.01$.

The average training times for the different learners (Q-learner, DQN, ACER and APEL) for both domains are shown in Figure 4.9 (refer to Section 4.1.1 and 4.1.2 for details on how train time and optimal performance are defined).

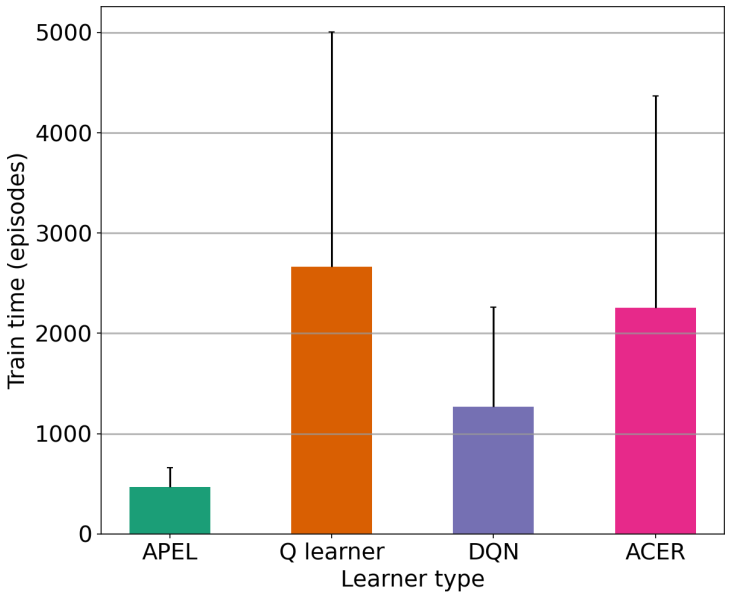


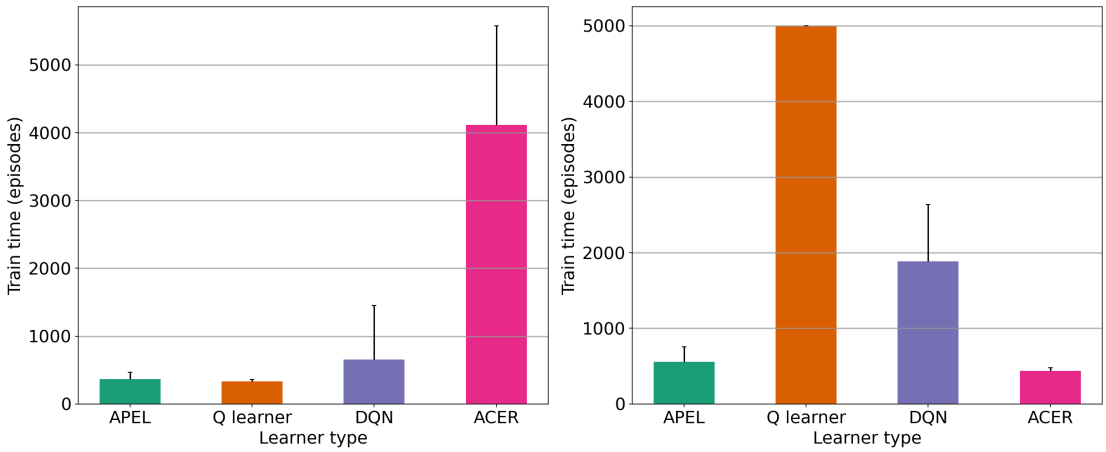
Figure 4.9: Mean train time for learners, averaged over both domains.

Figure 4.9 shows that APEL outperforms all the base learners, when averaging over all runs from both domains. This confirms that APEL can be provided with a set of different types of base learners and it will select the most appropriate learner for a given task while training, allowing it to outperform the base learners on average over different tasks and removing the need for an RL practitioner to select the best algorithm beforehand.

The average train times for the maze domain and the Cartpole domain are shown in Figure 4.10a and 4.10b respectively and show that APEL performs approximately as well as the best performing base learner on each domain. Since different types of learners perform well on the different domains, APEL leverages the strengths of multiple base learners to outperform the base learners on average over the different domains by selecting the appropriate type of learners for each domain while training.

4.3.2 Adapting to changing learner performance

Given a set of base learners being trained on a task, the best base learner to select at any given timestep is likely to change throughout the course of training, because of different learner characteristics or even non-stationary environment dynamics. Subsequently, an ensemble learner should be able to adapt if the best base learner changes.



(a) Maze

(b) Cartpole

Figure 4.10: Mean train time for learners for each domain.

This experiment evaluates APEL's ability to adapt to changing base learner performance by providing two base learners, one which performs better initially and one that performs better after episode e_s, at which point the environment dynamics change. We examine the time it takes APEL to adapt after the dynamics have shifted.

We use the maze domain with two Q-learners as base learners. We simulate the switching environment dynamics by varying the amount of exploration done by the two learners by changing their exploration parameters, ε, at the switch point, e_s, and keeping them fixed otherwise. The base learners start with $\varepsilon_1 = 0.4, \varepsilon_2 = 0.8$ and then at e_s the second learner changes to $\varepsilon_2 = 0.01$. The time (number of episodes) it takes the ensemble learner to adapt (switch to the better learner), $e_a = e_T - e_s$, is analysed.

The following hyperparameters are used (see Section 4.3.6 for details on how the hyperparameters were selected):

- Q-learner: $\alpha = 0.1, \gamma = 1$.
- APEL: $\nu = 0.01$.
- SSBAS: $\xi = 4.921$.

The time the ensemble learner takes to adapt, e_a, is shown for different switch points, e_s, in Figure 4.11. The results are averaged over 100 runs.

Figure 4.11 shows that APEL is able to adapt to changes in base learner performance relatively quickly (less than 500 episodes) and adapts to changes more quickly on average than SSBAS and with significantly less variance in the time it takes to adapt. Another interesting property of APEL can be seen in Figure 4.11: the time it takes APEL to adapt stays relatively constant, regardless of the switch point. This shows that APEL does not become overconfident in its selection of the base learner that it thinks is the best and still explores the other base learner periodically and is therefore able to adapt relatively quickly, even if it has been selecting one base learner for a long time.

The distribution of the adaptation times for $e_s = 1000$ and $e_s = 20000$ are shown in Figure 4.12.

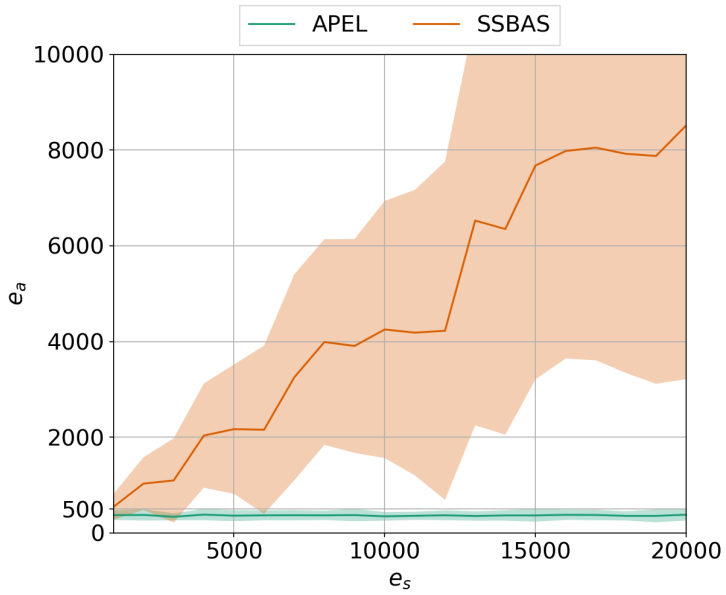
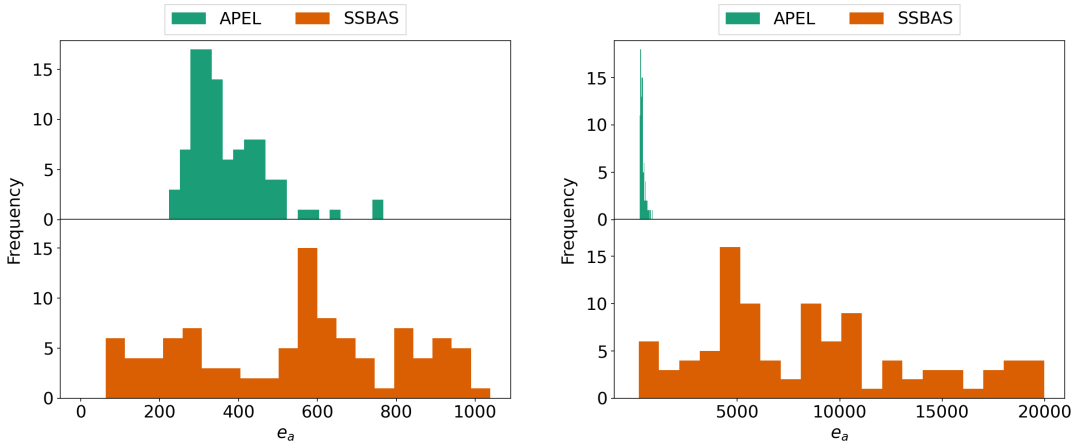


Figure 4.11: Time to adapt to base learner changes for different switch points.

APEL adapts in a relatively consistent manner, with little variation in the amount of time taken to adapt, whereas the time it takes SSBAS to adapt is highly variable. APEL's adaptation also stays relatively consistent between $e_s = 1000$ and $e_s = 20000$, whereas SSBAS's adaptation times become even more spread out when the switch takes place later.

APEL's decay parameter directly affects its ability to adapt to changing base learner performance. Figure 4.13 shows e_a as a function of e_s (similar to Figure 4.11 and using the same parameters) for different values of ν. The results are averaged over 100 runs.

Figure 4.13 shows that the time it takes APEL to adapt stays relatively constant regardless of e_s, as long as there is some decay ($\nu \neq 0$). The time it takes to adapt also scales inversely with the value of ν, with more decay resulting in faster adaptation times. Similar to the theoretical analysis of Section 3.6.1, Figure 4.13 shows that an order of magnitude increase in ν results in roughly an order of magnitude decrease in e_a, although the actual to time adapt is longer than in the theoretical analysis due to the random exploration that needs to take place before the concentration parameters start getting adjusted.



(a) $e_s = 1000$. (b) $e_s = 20000$.

Figure 4.12: Distribution of adaptation times for different switch points.

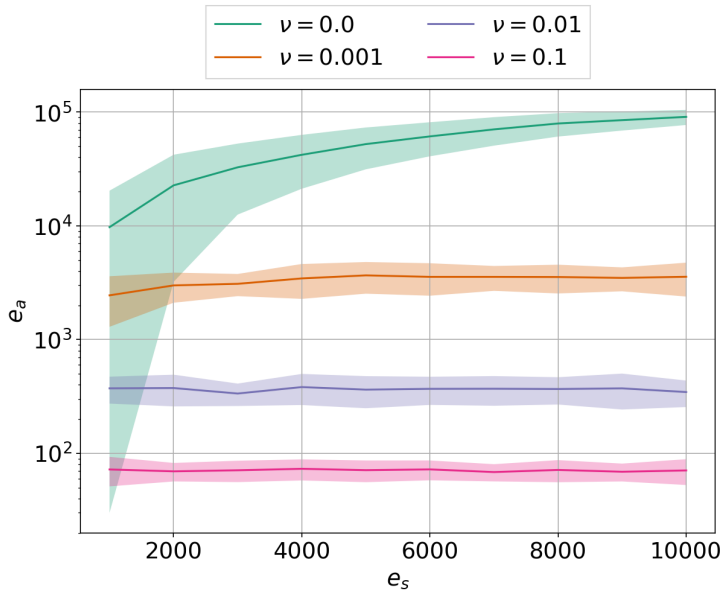


Figure 4.13: Effect of ν on the time it takes APEL to adapt.

4.3.3 Effect of decay parameter on steady-state performance

The steady-state exploration performed by APEL can degrade the learner's average performance if some learners do not reach optimal performance in the steady-state. The amount of steady-state exploration is influenced by ν and therefore this experiment analyses the decay parameter's effect on APEL's steady-state performance.

We use the maze domain and give APEL two base learners. The first base learner is a normal Q-learner with the full state space, Q_{norm}. The second base learner is either a bad Q-learner, Q_{bad}, that has a reduced state space that excludes the y state variable (preventing it from learning the optimal policy), or a slow Q-learner, Q_{slow}, that is identical to Q_{norm} except that its learning rate is $1/10th$ of Q_{norm}'s. APEL is allowed to train for much longer than necessary for its base learners to reach optimal performance – it trains for $20,000$ episodes and reaches optimal performance in $2,000 - 5,000$ episodes. The steady-state performance (its mean return per episode, averaged over the last 500 episodes) is then analysed.

The Maze domain gives -0.1 reward for each step except when the agent reaches the terminal state where it receives $+1$ reward. It also limits episodes to 1000 timesteps. We constrain the optimal path length of the generated mazes to be exactly 34 in this experiment. The expected steady-state return of Q_{norm} and Q_{slow} is therefore $G_L = 1 - 33 \cdot 0.1 = -2.3$. The expected steady-state return of Q_{bad} is $G_L = -1000 \cdot 0.1 = -100$.

APEL's steady-state performance, for different values of ν, is shown in Figure 4.14. The theoretical performance in each case is given by $\text{APEL}_{\text{Q-bad}}$ and $\text{APEL}_{\text{Q-slow}}$ and the actual performance (determined in the experiment) is given by $\hat{\text{APEL}}_{\text{Q-bad}}$ and $\hat{\text{APEL}}_{\text{Q-slow}}$ respectively. Since Q_{slow} reaches the same steady-state performance as Q_{norm}, $\text{APEL}_{\text{Q-slow}} = Q_{norm} = Q_{slow}$.

Similar to the theoretical analysis presented in Section 3.6.4, Figure 4.14 shows that the steady-state performance scales linearly with increases in ν. In the case of the slow learner, the linear scaling results in a relatively constant steady-state performance, since Q_{slow} reaches the same steady-state performance as Q_{norm}. On the other hand, in the case of the bad learner, the linear scaling results in a linear degradation of steady-state performance since Q_{bad} performs much worse than Q_{norm}.

The theoretical steady-state performance, calculated by substituting the different values of ν and the aforementioned values of G_L into Equation 3.14, is also shown in Figure 4.14. The steady-state performance degradation of Q_{bad} is very similar to the theoretical degradation, except that it grows slightly more quickly as ν increases.

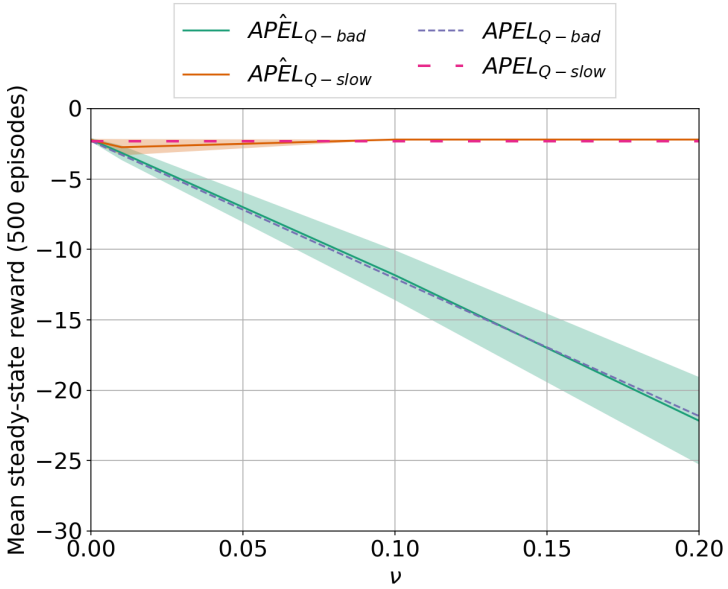


Figure 4.14: Effect of ν on APEL's steady-state performance when using a bad learner or a slow learner. Q_{bad} is a straight line at -1000 that is left out to keep the other lines visible.

4.3.4 Effect of additional learners on performance

This experiment analyses the effect that adding many redundant base learners has on the performance of the ensemble learner by using Q_{full} and Q_{red} again, but this time Q_{red} excludes the y state variable and therefore Q_{red} is never able to learn to finish the task.

We use the maze domain once again, with two cases: many Q_{full}, where the ensemble learner has one Q_{red} learner and n Q_{full} learners, and many Q_{red}, which is the inverse. n is shown on the x-axis.

The train time of the ensemble learner for different amounts of base learners and different values of ν is shown in Figure 4.15. The results are averaged over 100 runs.

Figure 4.15 shows that APEL's performance does not degrade substantially when no decay is used. When the concentration parameters are decayed, there is a definite degradation in performance, especially when $n > 5$. The degradation seems to scale approximately linearly with the number of learners, which is a small price to pay for the benefits provided by APEL (as shown in the previous sections). In both cases ($\nu = 0.0$ and $\nu = 0.01$), the performance degradation is less severe when there are many good learners (many Q_{full}) than when there are many learners that cannot complete the task (many Q_{red}).

The results shown in Figure 4.13 and 4.15 show that ν gives us the ability to trade off between APEL's speed of adaptation to changes in base learner performance and its robustness to performance degradation when many (possibly redundant) base learners are used.

Next, we show the effect of additional learners on performance, when using different types of learners on different environments.

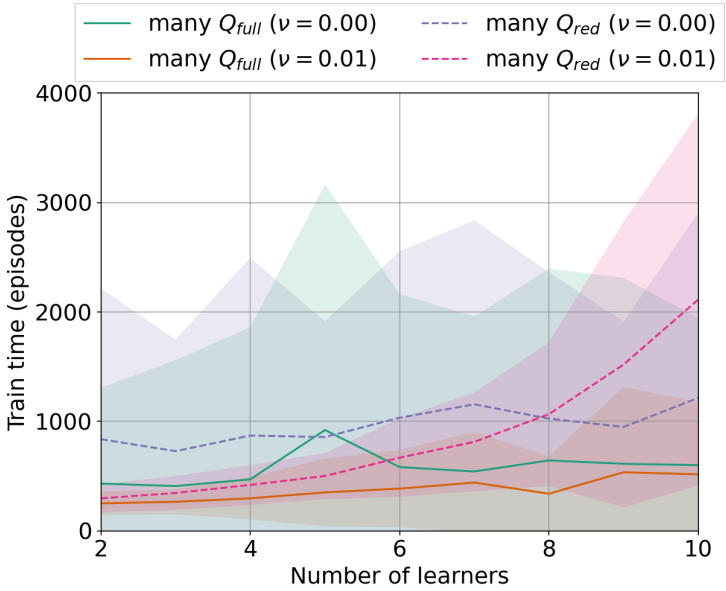


Figure 4.15: Effect of additional (redundant) base learners on the train time of the ensemble learner.

Q-learners and ACER-learners on the maze environment

In this case, we use Q-learners as "good" learners (that can easily solve the maze) and ACER-learners with very small neural networks (16 hidden units) as "bad" learners (that cannot solve the maze). We use the same two cases for the base learners: *many good learners*, where the ensemble learner has one bad learner (ACER-learner) and n good learners (Q-learners), and *many bad learners*, which is the inverse. APEL's performance is shown for different values of n in Figure 4.16. n is shown on the x-axis.

Similar to the case with Q_{full} and Q_{red}, there is only marginal degradation in performance when many good learners are used, but more degradation when many bad learners are used. The degradation in performance when using many bad learners is roughly linear.

Q-learners and ACER-learners on the Cartpole environment

In this case, we use ACER-learners as "good" learners and Q-learners as "bad" learners. The Q-learners use a very coarse quantisation of the continuous state space of Cartpole which renders them unable to solve the task. We once again use the same two cases for the base learners: *many good learners*, where the ensemble learner has one bad learner (Q-learner) and n good learners (ACER-learners), and *many bad learners*, which is the inverse. APEL's performance is shown for different values of n in Figure 4.17. n is shown on the x-axis.

APEL's degradation characteristics stay the same in this case, with barely any degradation in performance when using many good learners and approximately linear degradation in performance when using many bad learners.

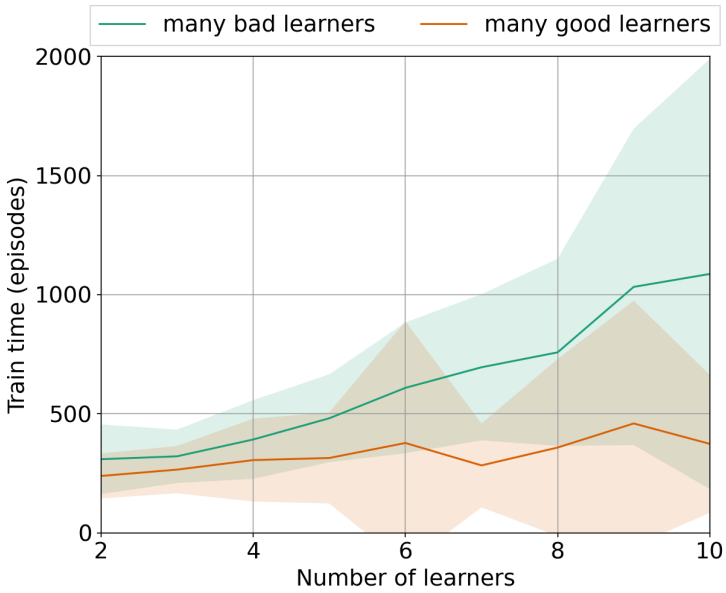


Figure 4.16: Effect of additional base learners on the train time of the ensemble learner when using Q-learners and ACER-learners on the maze environment.



Figure 4.17: Effect of additional base learners on the train time of the ensemble learner when using Q-learners and ACER-learners on the Cartpole environment.

4.3.5 Comparison with majority voting ensemble learners

In this section, we compare APEL to the ensemble RL algorithms proposed by Wiering and Van Hasselt (2008) (see Section 2.5.1 for more detail). Of the four algorithms proposed by Wiering and Van Hasselt (2008), the majority vote algorithm performed best in their paper and therefore we only show the results of the majority vote algorithm here. Due to the limitations of the majority voting ensemble learning algorithm shown in this section, we do not include it as a comparison in the other experiments and instead compare to SSBAS, which does not have these limitations, and therefore makes for a better comparison.

Similar to Section 4.3.1, we once again use the maze domain, augmented with one 10 dimensional random distractor variable. We also use the same two Q-learners: Q_{full} with the full state space and Q_{red} with a reduced state space that excludes one of the state variables. Q_{red} will solve the task faster than Q_{full} if the excluded variable is a distractor variable, otherwise it will be unable to solve the task (if it excludes the x or y state variable instead of the distractor variable).

The following hyperparameters were used:

- APEL: $\nu = 0.01$
- Majority voting ensemble learner: Boltzmann temperature $\tau = 0.1$. This value was chosen after performing a sweep of the τ parameter in the range $[0.01, 10]$ and finding 0.1 to be the best value.

The average training times for the different learners (reduced Q-learner, normal Q-learner, APEL and the majority voting (MV) ensemble learner) are shown in Figure 4.18. The results are averaged over 100 runs. In this experiment, the learners are only allowed to train for a maximum of $3,000$ episodes. If optimal performance is not reached in this time, the train time is set to the maximum train time of $3,000$ episodes.

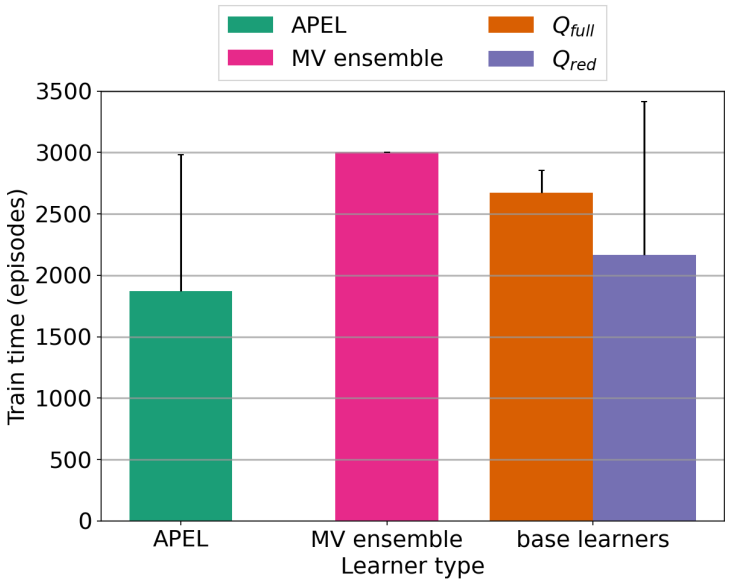


Figure 4.18: Mean train time for base learners and ensemble learners.

Figure 4.18 shows that APEL outperforms all the other learners on average. The majority

voting ensemble learner, on the other hand, never reaches optimal performance and therefore never finishes training. We argue that the majority voting ensemble learner is hindered by the differences in the two base learners (especially when Q_{red} accidentally excludes the wrong state variable) and is therefore not able to reach optimal performance.

The learning curves of the four learners are shown in Figure 4.19, which clearly shows that the majority voting ensemble learner takes longer to learn than the other learners and never reaches optimal performance. The learning curves are averaged over 20 runs.

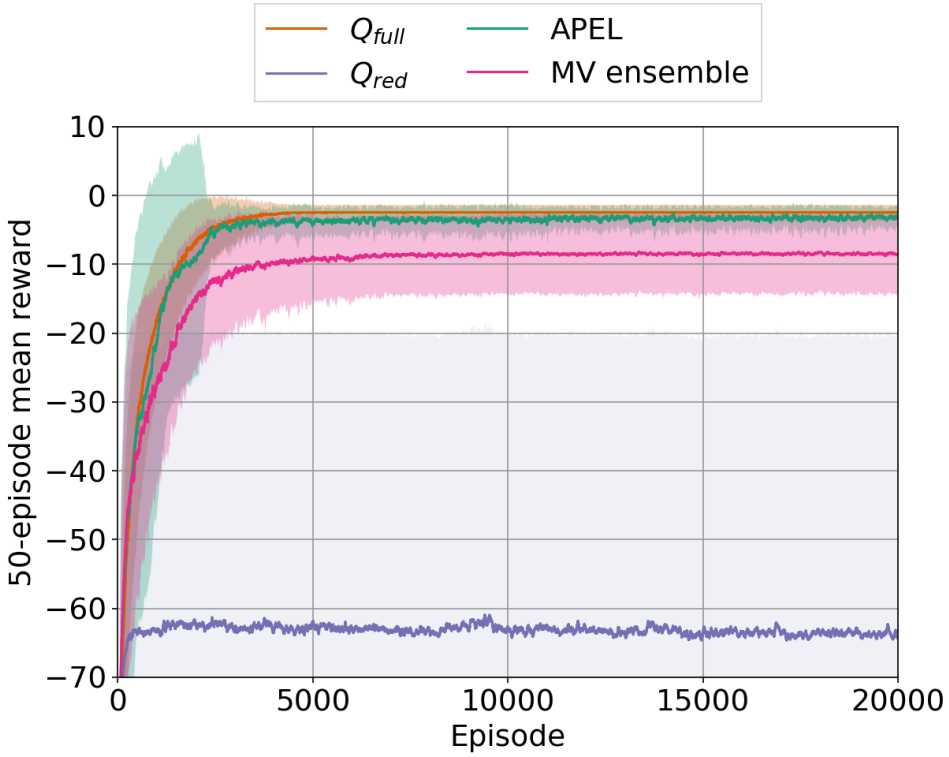
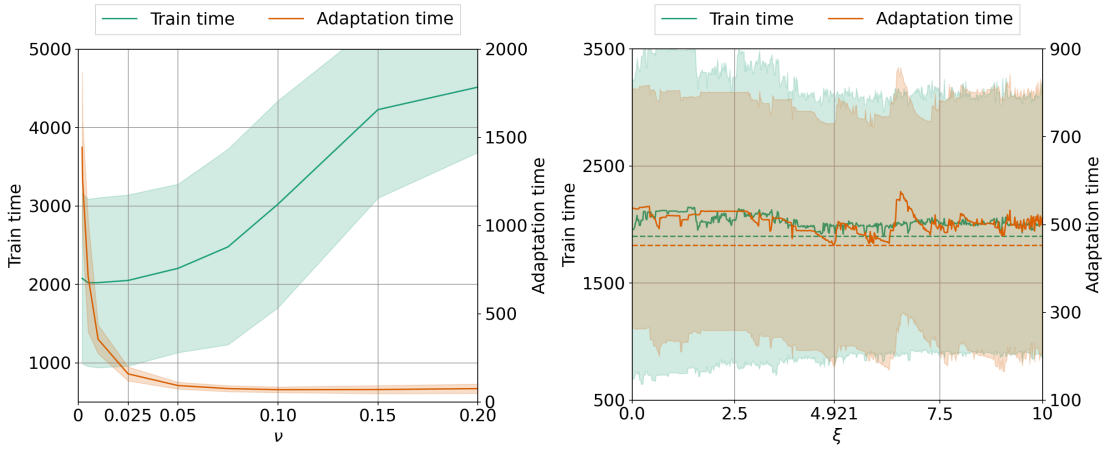


Figure 4.19: Learning curves for base learners and ensemble learners.

We conclude that, in the event that one of the base learners learns suboptimal behaviour (which is the case when Q_{red} excludes the wrong variable), the majority voting ensemble learner is hindered by the suboptimal actions generated by the suboptimal learner and therefore never reaches optimal performance, whereas APEL is able to leverage the strengths of its base learners and ignore their weaknesses in order to obtain optimal performance.

4.3.6 Hyperparameter selection

The hyperparameters used for APEL and SSBAS in the experiments were chosen by evaluating each learner's train time and adaptation time (see Section 4.3.1 and 4.3.2 for the experimental setup) for different values of each learner's hyperparameter and selecting a value for the hyperparameter that results in a low train time as well as a low adaptation time. The results for ν (APEL) and ξ (SSBAS) are shown in Figure 4.20. The lowest value for train time and adaptation time are shown by dashed horizontal lines and the results are averaged over 30 runs.



(a) APEL: ν (b) SSBAS: ξ

Figure 4.20: Performance of APEL and SSBAS as their hyperparameters are varied.

The hyperparameter sweep for ν confirms Section 3.6's results by showing that the selection of ν presents a trade-off between APEL's train time and its adaptation time. We use $\nu = 0.01$ for all the experiments, to strike a balance between the train time and the adaptation time.

The hyperparameter sweep for ξ shows that ξ does not have an intuitive effect on SSBAS's train time and adaptation time, increasing the difficulty of selecting a hyperparameter before training (i.e. without first running a hyperparameter sweep). We use $\xi = 4.921$, since that results in the lowest adaptation time and still yields a reasonably low train time.

Another advantage that APEL has when compared to SSBAS is the ability to intuitively select the hyperparameter before doing training. A designer can select a reasonable value for ν before doing any training by selecting ν according to their requirements: selecting a smaller ν if steady-state performance and train time are more important than adaptation performance, or selecting a larger ν if adaptation performance is more important than steady-state performance and train time (see Figure 4.20a). The designer can further improve their initial value for ν by using the design equations presented in Section 3.6. SSBAS's ξ parameter, on the other hand, cannot be set intuitively before training since SSBAS's train time and adaptation time varies seemingly randomly as ξ is changed (see Figure 4.20b). It is therefore necessary to perform a sweep or search in hyperparameter space in order to find a good value for ξ.

4.4 Summary

In this chapter we showed qualitatively that APEL is able to select the best base learner online and combine the strengths of multiple base learners. The qualitative results are similar to the results of the theoretical analysis, with small discrepancies due to practical implementation details and the assumptions made in the theoretical analysis.

The qualitative analysis showed that APEL can outperform any individual base learner on average when run on multiple tasks, and adapt to changing base learner performance. We showed that APEL performs better than SSBAS (a related method) on these tasks. Additionally, we showed that APEL also outperforms majority voting ensemble learners when the ensemble contains a

suboptimal base learner. We also showed that the decay parameter controls APEL's steady-state performance and adaptation time, with a larger decay value leading to a shorter adaptation time but also resulting in worse steady-state performance. The degradation in APEL's performance when adding redundant base learners was quantified and we showed that APEL's performance only really degrades when many suboptimal base learners are added to its set of base learners, while almost no degradation is observed when many optimal base learners are added to its set of base learners. Lastly, we showed how the hyperparameters used in all the experiments were chosen and showed that a good first value for APEL's hyperparameter can be selected intuitively before performing any training.

Throughout the experiments, the decay parameter's effect was shown in different scenarios which showed that the decay parameter can be used to trade off APEL's steady-state performance and robustness to redundant base learners with the speed at which it can adapt to changes in base learner performance.

Chapter 5

Related Work

There are different kinds of methods related to our problem of using a set of learners to improve learning. In Section 5.1 we discuss a method that frames the same problem as a bandit problem, which is the most closely related to our work. We also discuss methods that reuse previously learnt policies (Section 5.2), methods that subdivide a single problem into multiple problems (Section 5.3). Finally, we present methods that automate the design of machine learning systems by automatically selecting the appropriate algorithms and hyperparameters for any given problem in Section 5.4.

5.1 Framing algorithm selection as a bandit problem

The only other work that is directly related to our work and also considers the same problem of selecting between different learners during training while remaining agnostic to the inner workings of the learners (i.e. the type of learning algorithm used and whether the state and action spaces are discrete or continuous), is the work by Laroche and Feraud (2018) which proposes two algorithms: Epochal Stochastic Bandit Algorithm Selection (ESBAS) and Sliding Stochastic Bandit Algorithm Selection (SSBAS).

The Epochal Stochastic Bandit Algorithm Selection (ESBAS) and SSBAS algorithms tackle the problem of improving the sample complexity of RL algorithms, and decrease the amount of guesswork involved in choosing RL algorithms and parameters, by selecting the best learner from a set of learners online to optimise an objective function (like the RL return). Similar to our work, ESBAS and SSBAS are also agnostic to the inner workings of each of their learners and SSBAS is also able to select between the different learners online.

ESBAS considers the learner (or algorithm) selection problem as a bandit problem and uses the UCB algorithm to balance the exploration and exploitation of the different arms of the bandit (different base learners in this case). The ESBAS algorithm is shown in Algorithm 6, which operates on a set of trajectories ($\mathcal{D}$), a portfolio of different algorithms ($\mathcal{P}$) according to a given objective function (μ). The learners' policies are frozen during each epoch (line 6-12) of ESBAS in order for the bandit framing to be theoretically valid.

The ESBAS algorithm is extended to the online RL setting by allowing policies to be updated after each transition and training the stochastic bandit on a sliding window of the last $\tau/2$ selections. This modified algorithm is dubbed SSBAS. We compare APEL to SSBAS since it

Algorithm 6 ESBAS with UCB1

Require: $\mathcal{D}$: trajectory set, $\mathcal{P}$: algorithm portfolio, μ: objective function, the online RL algorithm selection setting.

for $\beta \leftarrow 0$ to ∞ **do**
  for $\alpha^k \in \mathcal{P}$ **do**
    $\pi^k_{\mathcal{D}_{2^\beta-1}}$: policy learnt by α^k on $\mathcal{D}_{2^\beta-1}$
  end for
  $n \leftarrow 0, \forall \alpha^k \in \mathcal{P}, n^k \leftarrow 0$, and $x^k \leftarrow 0$
  for $\tau \leftarrow 2^\beta$ to $2^{\beta+1}-1$ **do**
    $\alpha^{k_{max}} = \underset{\alpha^k \in \mathcal{P}}{\text{argmax}} \left(x^k + \sqrt{\xi \frac{\log(n)}{n^k}} \right)$
    Generate trajectory ε_τ with policy $\pi^{k_{max}}_{\mathcal{D}_{2^\beta-1}}$
    Get return $\mu(\varepsilon_\tau), \mathcal{D}_\tau \leftarrow \mathcal{D}_{\tau-1} \cup \{\varepsilon_\tau\}$
    $x^{k_{max}} \leftarrow \frac{n^{k_{max}} x^{k_{max}} + \mu(\varepsilon_\tau)}{n^{k_{max}}+1}$
    $n^{k_{max}} \leftarrow n^{k_{max}} + 1$ and $n \leftarrow n+1$
  end for
end for

is the only other algorithm that can select between different learners online while remaining agnostic to the learning mechanisms of the learners (e.g. value-based methods and actor-critic methods).

SSBAS's exploration mechanism is different from APEL's. Where APEL selects base learners probabilistically according to distributions sampled from its Dirichlet distribution, SSBAS selects learners deterministically according to the UCB method. The UCB method increases the value of an action (or learner in this case) according to the time that has elapsed since that learner was last selected, up to a maximum value (adjusted by the ξ parameter). Due to this deterministic learner selection, SSBAS is sensitive to the differences in rewards obtained by the base learners, as well as the value of the ξ parameter. Large differences in the rewards of the base learners, or a small value of ξ, cause SSBAS to be overly confident in the (supposed) optimal base learner and prevents it from still exploring the other learners. APEL, in contrast, only considers whether one base learner performs better than another base learner (not by how much) and is therefore not sensitive to the scale of the rewards.

SSBAS reduces this overconfidence by training the bandit algorithm on a sliding window of experience that grows as training progresses. The sliding window allows it to "forget" old evidence, but limits its ability to adapt to changes in base learner performance timeously, when compared to APEL (as shown in our experiments).

5.2 Policy reuse

Policy reuse methods use a set of previously learnt policies to improve the learning of a new agent by attempting to select the best previous policy to use at each timestep and in doing so bootstrap training on new tasks. The previous policies are frozen and are not allowed to learn further along with the new agent. Due to the requirement of having access to a set of previously trained policies, policy reuse methods are typically used in a transfer learning setting.

Policy Reuse in Q-learning (PRQL) (Fernández and Veloso, 2006), Context-Aware Policy Reuse

(CAPS) (Li et al., 2019) and Actor Critic with Teacher Ensembles (AC-Teach) (Kurenkov et al., 2019) are examples of algorithms that reuse previous policies probabilistically for exploration while training a new policy. An alternative method, Bayesian Policy Reuse (BPR) (Rosman et al., 2016), uses a Bayesian approach to select previous policies that maximise the acquisition of useful information about the current task, while minimising the amount of additional regret accumulated.

Similar to our work, policy reuse methods also consider the problem of choosing between different learners (policies) but require learners to be pre-trained and keep the policies of the learners they select between fixed during training of the new learner, as opposed to our work which considers the case where learners' policies are being learned online while also selecting between them. Our work can therefore be considered as a form of policy reuse, but extended to the online case.

5.3 Task subdivision

Instead of training an ensemble of learners on the full task, the original single-agent problem can instead be divided into different subtasks, with a different learner assigned to learning each subtask.

Multi-Advisor Reinforcement Learning (MAd-RL) (Laroche et al., 2017) divides a task into multiple subtasks by using multiple base learners (which they call advisors), which each receive a specific source of reward. The advice of the different learners are communicated to an aggregator which is in control of the system. The learners all learn in parallel from off-policy experience and the knowledge gained by the different learners are combined again by summing their action-value functions before deciding which action to use. Dividing the task into subtasks greatly reduces the state space of each individual learner and allows MAd-RL to train faster and get better scores than single learner baselines.

Divide-and-conquer RL (Ghosh et al., 2018) instead partitions the state space into slices and then trains an ensemble of policies on these different slices. Divide-and-conquer RL alternates between local policy optimisation (where the ensemble of policies are trained) and global policy optimisation (where the ensemble of policies are gradually unified into a single policy able to solve the task). Partitioning the state space in this way makes Divide-and-conquer RL robust to variation in the initial state of the agent.

Methods that subdivide tasks tend to require specific learning methods: Divide-and-conquer RL uses learners that learn with MDPs that are extended with context and MAd-RL aggregates over the different learners by summing their action-value functions, therefore requiring the use of algorithms that learn tabular representations of the action-value function. In contrast, our work only requires learners to use off-policy algorithms but allows any internal representation (tabular or function approximation, value function or policy) to be used. Task subdivision methods also typically use the same type of learner for all base learners, whereas our work is applicable to different types of base learners (e.g. a Q-learner and a DQN-learner). Since Divide-and-conquer RL and MAd-RL require the entire ensemble of learners to be of the same (specific) type, they are not directly comparable to APEL that allows different types of base learners to be used simultaneously.

5.4 AutoML

Automated machine learning (AutoML) considers a problem similar to ours: automatically selecting and parametrising machine learning algorithms, without human intervention, in order to achieve optimal performance on a task. APEL also considers the problem of selecting between different algorithms, but does so online (while training), whereas most AutoML methods focus on the offline case. AutoML methods also tune the hyperparameters of the algorithms and although APEL can be used with the multiple instantiations of a single algorithm each using different hyperparameters, its ability to tune hyperparameters in this way is limited when compared to AutoML methods.

There are various approaches to simultaneously selecting algorithms and hyperparameters in the supervised learning space. The selected algorithm can be considered to be another hyperparameter and then random forest models can be used to search the combined hyperparameters space to find combinations of algorithms and hyperparameters that outperform existing methods (Thornton et al., 2013). Alternatively, Bayesian optimisation can be used with a meta-learning warm-start step to boost the efficiency of the learning process (Feurer et al., 2015). Hierarchical planning can also be used to solve the AutoML problem by constructing hierarchical task networks and employing a two-phase search and select approach (Mohr et al., 2018).

AutoML can also be applied to RL by considering the RL reward function to be a hyperparameter and using AutoML methods to automatically search for a good reward function (Faust et al., 2019). Most AutoML methods are constrained to static data sets, but AutoML can also be used with slowly changing data sets by using concept drift detection techniques that determine when initial models should be adapted (Madrid et al., 2018). The setting we consider in this work is, however, not constrained to slowly changing dynamics (the performance of base learners or the environment dynamics could change quickly) and therefore AutoML with concept drift detection would not be able to adequately solve our problem.

5.5 Summary

In this chapter we presented methods related to APEL that attempt to solve the same problem, or similar problems. We showed that SSBAS is the only related method that also considers the problem of selecting between different base learners while staying agnostic to the inner workings of the base learners, but instead of using a probabilistic selection method like APEL, SSBAS uses the UCB method.

We gave an overview of policy reuse, which also uses different policies (learners) like APEL does, but uses the other policies to bootstrap training of a single new policy and does not allow the old policies to train online like APEL does. We showed that RL methods that use task subdivision are similar to APEL since these methods also use an ensemble of learners, but instead of focusing on using a variety of different types of learners like APEL, they instead focus on dividing a task into subtasks and training similar learners on these different (simpler) subtasks.

Finally, we presented methods from the AutoML literature, which focus on automatically selecting between different algorithms and selecting hyperparameters for these algorithms. Unlike APEL, AutoML methods generally use offline datasets when choosing between different algorithms and AutoML techniques have not been directly applied to the RL problem yet.

Chapter 6

Conclusions and Future Work

We have presented Adaptive Probabilistic Ensemble Learning (APEL), an ensemble learner that selects between base learners online by using a Dirichlet distribution to shift probability mass to base learners that perform well. APEL is capable of outperforming individual base learners on average, adapting to changing base learner performance and leveraging the strengths of multiple base learners online. Unlike previous methods in the ensemble RL space, APEL is adept at handling learners that have different inner workings (state representations and learning mechanisms).

APEL decays the concentration parameters of its Dirichlet distribution to allow it to quickly adapt to changes in base learner performance. This concentration parameter decay mechanism was thoroughly investigated through a theoretical analysis that was validated by empirical results. We provided guidelines in the selection of the decay parameter and showed that the decay parameter can be used to trade off APEL's steady-state performance and its robustness to redundant learners with the speed at which it adapts to changes in base learner performance.

We showed that, compared to SSBAS, a related method that is also agnostic to the inner workings of its base learners, APEL has similar performance when evaluated on its ability to outperform individual base learners on average (see Section 4.3.1), and better performance when evaluated on its ability to adapt to changes in base learner performance (see Section 4.3.2). In Section 4.3.5, we showed that APEL is robust to suboptimal learners in its ensemble and can still reach optimal performance despite these suboptimal learners, whereas existing ensemble RL techniques are hindered by such suboptimal learners and never reach optimal performance when their ensemble includes suboptimal learners. We also showed that APEL is able to leverage the strengths of multiple base learners online by evaluating the learning curves of two different base learners and APEL on the Atari Breakout domain (see Section 4.2.3).

In the presented formulation, the Dirichlet distribution's concentration parameters are updated by simply incrementing the concentration parameter of the selected base learner if it performed better than the other base learner (when they were last selected). An avenue for future work could be to improve the update rule by adding a factor to the update that takes into account the time that has elapsed since that base learner was last selected. If a base learner has not been selected for a long time (which implies that APEL has observed that it generally performs worse than the other base learners) and then performs better than the other base learners once it is selected again, it would make sense intuitively to update the concentration parameter more

than usual.

Future work could also consider adding a factor to the concentration parameter update, or scaling the update, according to the actual performance of the base learner. Instead of simply incrementing the concentration parameter when the selected base learner does better than the other base learners, the concentration parameter could be increased proportional to the difference between the selected base learner's performance and the performance of the other base learners. Alternatively, the update could be scaled, for example, according to the probability that the selected base learner could achieve the given performance. If a model trained on the selected base learner's previous performance estimates assigns a low probability to the selected base learner achieving a certain performance (i.e. the level of performance is surprising), the concentration parameter update could carry more weight.

Bibliography

P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-Time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.*, 47(2–3):235–256, May 2002.

A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. doi: 10.1109/TSMC.1983.6313077.

R. Bellman. *Dynamic programming*. Princeton University Press, 1957.

D. Berry and B. Fristedt. *Bandit problems*. Chapman and Hall, London, 1985.

L. Breiman. Bagging Predictors. *Machine Learning*, 24(2):123–140, Aug. 1996.

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv e-prints*, page arXiv:1606.01540, 2016.

R. Y. Chen, S. Sidor, P. Abbeel, and J. Schulman. UCB exploration via Q-ensembles. *arXiv e-prints*, page arXiv:1706.01502, 2017.

P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov. *OpenAI Baselines*. GitHub, 2017. URL `https://github.com/openai/baselines`.

D. Ernst, P. Geurts, and L. Wehenkel. Tree-Based Batch Mode Reinforcement Learning. *Journal of Machine Learning Research*, 6(18):503–556, 2005.

A. Faust, A. Francis, and D. Mehta. Evolving Rewards to Automate Reinforcement Learning. In *6th ICML Workshop on Automated Machine Learning*, 2019.

F. Fernández and M. Veloso. Probabilistic Policy Reuse in a Reinforcement Learning Agent. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '06, pages 720–727, New York, NY, USA, 2006. Association for Computing Machinery. event-place: Hakodate, Japan.

M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and Robust Automated Machine Learning. In *Advances in Neural Information Processing Systems*, volume 28, pages 2962–2970. Curran Associates, Inc., 2015.

Y. Freund and R. E. Schapire. Experiments with a New Boosting Algorithm. In *Proceedings of the Thirteenth International Conference on International Conference on Machine Learning*, ICML'96, pages 148–156, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

D. Ghosh, A. Singh, A. Rajeswaran, V. Kumar, and S. Levine. Divide-and-Conquer Reinforcement Learning. In *6th International Conference on Learning Representations, ICLR 2018,*

Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net, 2018.

T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1856–1865. PMLR, 2018.

R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive Mixtures of Local Experts. *Neural Computation*, 3(1):79–87, 1991.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement Learning: A Survey. *J. Artif. Int. Res.*, 4(1):237–285, May 1996.

S. Kotz, N. L. Johnson, and N. Balakrishnan. *Continuous multivariate distributions*. Wiley, 2 edition, 2000.

A. Kurenkov, A. Mandlekar, R. Martin-Martin, S. Savarese, and A. Garg. AC-Teach: A Bayesian Actor-Critic Method for Policy Learning with an Ensemble of Suboptimal Teachers. In *3rd Annual Conference on Robot Learning, CoRL 2019, Osaka, Japan, October 30 - November 1, 2019, Proceedings*, volume 100 of *Proceedings of Machine Learning Research*, pages 717–734. PMLR, Sept. 2019.

T. Kurutach, I. Clavera, Y. Duan, A. Tamar, and P. Abbeel. Model-Ensemble Trust-Region Policy Optimization. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

R. Laroche and R. Feraud. Reinforcement Learning Algorithm Selection. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

R. Laroche, M. Fatemi, H. van Seijen, and J. Romoff. Multi-Advisor Reinforcement Learning. *arXiv e-prints*, page arXiv:1704.00756, Apr. 2017.

S. Li, F. Gu, G. Zhu, and C. Zhang. Context-Aware Policy Reuse. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '19, pages 989–997, Richland, SC, 2019. International Foundation for Autonomous Agents and Multiagent Systems. event-place: Montreal QC, Canada.

J. G. Madrid, H. Jair Escalante, E. F. Morales, W.-W. Tu, Y. Yu, L. Sun-Hosoya, I. Guyon, and M. Sebag. Towards AutoML in the presence of Drift: first results. In *Workshop AutoML 2018 @ ICML/IJCAI-ECAI*, Stockholm, Sweden, July 2018. Pavel Brazdil and Christophe Giraud-Carrier and Isabelle Guyon.

V. Marivate and M. Littman. An Ensemble of Linearly Combined Reinforcement-Learning Agents. In *Proceedings of the 17th AAAI Conference on Late-Breaking Developments in the Field of Artificial Intelligence*, AAAIWS'13-17, pages 77–79. AAAI Press, 2013.

V. Mnih. Deep RL Bootcamp: Deep Q-Networks, 2017. URL: `https://drive.google.com/file/d/0BxXI_RttTZAhVUhpbDhiSUFFNjg/view`. Accessed on 2020/09/15.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou,

H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015.

F. Mohr, M. Wever, and E. Hüllermeier. ML-Plan: Automated machine learning via hierarchical planning. *Machine Learning*, 107(8):1495–1515, Sept. 2018.

A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018.

M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York, 1994.

B. Rosman, M. Hawasly, and S. Ramamoorthy. Bayesian policy reuse. *Machine Learning*, 104 (1):99–127, July 2016.

J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust Region Policy Optimization. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897, Lille, France, July 2015. PMLR.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal Policy Optimization Algorithms. *arXiv e-prints*, page arXiv:1707.06347, 2017.

S. P. Singh. The Efficient Learning of Multiple Task Sequences. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 251–258. Morgan-Kaufmann, 1992.

R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 2nd edition, 2018. ISBN 978-0-262-03924-6.

C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 847–855, New York, NY, USA, 2013. Association for Computing Machinery.

Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. d. Freitas. Sample Efficient Actor-Critic with Experience Replay. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, volume abs/1611.01224. OpenReview.net, 2017.

C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992. Publisher: Springer.

M. Wiering and H. Van Hasselt. Ensemble Algorithms in Reinforcement Learning. *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society*, 38:930–6, 2008.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992. Publisher: Springer.