university of groningen

faculty of science and engineering

Episodic Control with Drift Compensation

Master's thesis, Computing Science

January 14, 2019

Student: Michael L. LeKander

Primary supervisor: prof. dr. M. Biehl, Computing Science

Secondary supervisor: dr. M.A. Wiering, Artificial Intelligence

Abstract

The ability to learn to act in complex interactive environments is a vital component of human intelligence. Reinforcement Learning is a rapidly growing area of research which attempts to produce agents which interact in an environment (e.g. Atari games) to maximize reward (e.g. the final game score). While "deep" approaches have been highly successful in this domain, they have the drawback of requiring millions of frames of experience in order to learn. Model Free Episodic Control is a recently proposed algorithm that addresses this issue by using nearest-neighbors regression. This algorithm has the desirable property of “immediate one-shot learning”, allowing it to quickly latch onto successful strategies.

In this research, we make three primary additions to the existing work. First, we devise an efficient online approximate nearest neighbors algorithm, which is highly important for the overall efficiency of the algorithm. Second, we explore using a wider spectrum of local regression techniques, of which nearest-neighbors regression is just a single example. Finally, we explore the use of explicit drift compensation, to account for changes in the underlying return function.

We ultimately produce the reinforcement learning algorithm termed Episodic Control with Drift Compensation. Through a series of experiments on a suite of five classic Atari 2600 games, we demonstrate that this novel algorithm makes improvements above the state-of-the-art, particularly in expanding the long-term capacity of the agents.

Acknowledgements

To my advisors, (the other) Michael and Marco, whose advice and supervision have been truly invaluable.

To all my teachors and mentors, who instilled in me a deep love for learning, and eventually, a love for deep learning.

And most of all, to Babet and all my other loved ones, who never gave up on me, and who never passed up an opportunity to persistently ask me how my thesis was going.

Contents

Chapter 1

Introduction

Every day, humans interact with their environment in accordance to their needs and desires. The underlying algorithm guiding human behavior seems to have the important property of being highly adaptable: the same mental systems that one uses to learn how to use a new text editor are also employed when learning how to cook a new dish, implement a new algorithm, or play a new video game. To write a document, one activates muscle groups to cause their fingers to move in a coordinated fashion so as to press the desired keys on a keyboard. This behavior would be entirely foreign to a feral human, yet we have learned how to perform these actions to the point where we can do so without conscious thought dedicated to the process. It is clear that humans are able to learn how to act in environments which could not have been foreseen a priori. This ability to learn to act in complex interactive environments is a vital component of human intelligence.

The field of machine learning investigates formal models of learning through the development and application of algorithms capable of autonomously improving performance on a given task. These machine learning tasks generally fall into a handful of broad categories based on the availability of data and the type of desired output Sutton and Barto (1999):

1. supervised learning, which outputs labels according to a fixed rule from data which has been labeled by a teacher;
2. unsupervised learning, which learns latent structures and patterns from unlabelled data;
3. and reinforcement learning, which chooses actions to interact with an environment and only provides data in the form of states and rewards.

In this thesis, we primarily focus on algorithms which address the reinforcement learning problem, although we do briefly touch on supervised learning for some sub-problems which are relevant for reinforcement learning.

1.1 Reinforcement Learning

Reinforcement learning (Wiering and van Otterlo, 2012) defines a set of machine learning problems in which an agent interacts with an environment so as to maximize an external reward. Through a process of exploration and generalization, the agent learns to perform a sequence of actions which results in the highest cumulative reward (the *return*) for the agent. One example of a problem which fits the reinforcement learning paradigm is the task of playing the classic video game Ms. Pacman in a way that maximizes the score achieved by the agent (Bom et al., 2013). Indeed, this is an environment which we use as a testbed throughout this thesis. Examples of other reinforcement learning problems include 3D environment navigation (Blundell et al., 2016), single-player video games (Mnih et al., 2015; Hessel et al., 2017; Lillicrap et al., 2015), multiplayer board games (Tesauro, 1995; Silver et al., 2016, 2017), and autonomous helicopter control (Abbeel et al., 2007).

There are a variety of different approaches that can be used to solve this problem domain. Planning-based techniques have been successfully applied in some domains, typically employing search-based algorithms. For example, alpha-beta search has classically been used to produce professional-level players in some combinatoric board games, such as in the game of chess (Campbell et al., 2002; Tord Romstad, 2018). Planning-based approaches require an exact (or incredibly accurate) model of the environment to produce effective results. However, this is an additional requirement above the general reinforcement learning paradigm, which (in full generality) does not assume that the agent has a priori access to or knowledge of the environment. Additionally, it is intractable to produce a suitably accurate model for some (particularly real-world) environments. We note that model-based methods attempt to learn this environment model from experience, but this is still an open field of research.

Another technique for addressing reinforcement learning problems involves estimating the "value" (the expected return) of a state. If an agent can accurately estimate the cumulative reward it will receive after taking any given action from a given state, then it can optimally act by simply selecting the action with the highest estimated return from the current state.

In environments with discrete state and action spaces, it is possible to keep a large table to store an estimate of the future return after taking each action from each state, producing so-called tabular approaches. However, it is not feasible to create such a table for environments with large spaces (because of insufficient memory) or continuous spaces (because it is impossible to enumerate all possibilities). Another issue is that each state is considered entirely separate from each other: the value of one state is considered entirely independently of those from all other states, regardless of how much those two states have in common. Using tabular methods, a state from a game of Ms. Pacman would be considered entirely independently from a state with only a single pixel difference. If an agent discovered a useful strategy from one state, it would not be able to use this information to update its prediction about the other one. While this level of skepticism is perhaps warranted in some environments, almost all environments contain regularities and abstractions over which a skilled agent should learn to generalize.

Value approximation methods allow the agent to generalize across states through the use of a regression model (e.g. linear, neural network, or k-nearest neighbor regression) to produce real-valued outputs which estimate the value function. In particular, neural network-based value function approximators have recently been used to great success, achieving state-of-the-art performance across a wide variety of environments (Mnih et al., 2015; Silver et al., 2017; Lillicrap et al., 2015; Abbeel et al., 2007). However, although neural networks have been shown to be highly flexible as value function approximators, they have the unfortunate drawback of being data-inefficient due to their reliance upon slow gradient-based updates. Thus, agents employing such approaches are unable to quickly latch onto and exploit new strategies which are advantageous.

The recently-proposed Model-Free Episodic Control (MFEC) algorithm addresses this issue through an approach inspired by instance-based learning of episodic memory in the hippocampus (Blundell et al., 2016). Specifically, this algorithm uses a variant of k-nearest neighbor regression to estimate the value function. This regression model has the advantage of rapidly adapting to new strategies for two reasons: firstly it can recall exact instances stored in its memory upon encountering an identical state, and secondly it does not require slow gradient-based steps to interpolate predictions for previously-unseen states. We describe this algorithm in more detail in Section 2.5.

1.2 Research Questions

In this thesis, we investigate the MFEC algorithm and explore the consequences of various design choices and improvements to the core algorithm. We explore three primary areas: the usage of an approximate nearest neighbors algorithm, the usage of various local regression techniques, and the usage of drift compensation.

1.2.1 Approximate Nearest Neighbors

In order to obtain reasonable performance, MFEC agents employ an approximate nearest neighbors data structure, which is not guaranteed to return the exact closest items to the query point. However, the details of this data structure were not specified in the original paper, and neither were the consequences of various accuracy settings.

Furthermore, most popular approximate nearest neighbors data implementations assume a static dataset and have significant construction times. However, the MFEC algorithm incrementally adds new elements to the dataset as more experience is gathered. Thus, we would ideally want an online data structure capable of adding and removing items without having to fully rebuild the entire data structure.

In this direction, we formulate the following research questions:

- Can an efficient approximate nearest neighbors data structure be defined which supports online addition and deletion?
- How does the loss of accuracy incurred by using these approximate data structures impact the performance of the MFEC agent on the reinforcement learning problem?

These questions will be addressed in Chapter 3.

1.2.2 Local Regression

The MFEC agent uses simple k-nearest neighbor regression to interpolate unseen states, which simply returns the average label of the nearest neighbors to the state. However, this is just the simplest of a family of regression methods known as local regression.

Simple k-nearest neighbor regression considers the contribution of each neighbor equally. However, local regression methods can be extended to use a kernel function which weighs the contribution of a neighbor's label according to how close that neighbor is to the query point. Thus, such a kernel would cause closer neighbors to be considered more important than further neighbors.

Additionally, simple k-nearest neighbor regression simply takes the average of the (weighted) labels of its neighbors. This can be seen as fitting a constant model to the (weighted) local data. However, more complex local models (e.g. a linear model) could be employed to better match the local data distribution, thus making more accurate predictions.

In this direction, we formulate the following research questions:

- Can alternate kernel weightings improve the performance of the agent above that of the constant weighting of simple k-nearest neighbor regression?
- Can linear local models improve the performance of the agent above that of the constant local model of simple k-nearest neighbor regression?

These questions will be addressed in Chapter 4.

1.2.3 Drift Compensation

Although MFEC agents learn to quickly exploit new strategies, they seem to have problems generalizing in the long run. We hypothesize one cause of this phenomenon is due to MFEC agents having no ability to "forget" experiences which are no longer relevant: once a MFEC agent obtains experience from a state, it retains that exact information. Although there is a mechanism for deleting old items when the maximum memory size is exceeded, this is not responsive as is desired for sample-efficient agents.

Thus, we look towards the field of concept drift from online learning as inspiration to identify when memories should be deleted. Ideally, as the agent learns new strategies which maximize reward in the environment, the agent should be able to actively forget old experiences which no longer align with the agent's knowledge and abilities. However, just because an experience is old does not mean it should automatically be discarded: it is possible that some experiences should be retained while other newer experiences should be discarded.

In this direction, we formulate the following research questions:

- Can a drift compensation algorithm be designed to only “bad” evict experiences while retaining “good” ones, based on a metric other than age?
- Can this drift compensation improve the performance of the agent?
- Can this drift compensation serve as a form of targeted exploration?

These questions will be addressed in Chapters 5 and 6.

Chapter 2

Reinforcement Learning

Reinforcement learning is an important subfield of machine learning where an agent learns how to take actions in an environment so as to maximize its cumulative reward (known as its *return*) (Wiering and van Otterlo, 2012).

An agent is situated in an environment and chooses a single action at each time-step. After an action is chosen, the environment gives the agent a *reward* (which can be positive, zero, or negative) dependent on the agent's action and environment's current state. The environment also simultaneously transitions the agent to a new state, again dependent on the chosen action. The goal of the agent is to choose actions which maximize the total reward it receives.

The reinforcement learning paradigm can and has been applied across multiple domains. While we focus primarily on classic Atari video games in this thesis, reinforcement learning algorithms have been successful in 3D environment navigation (Blundell et al., 2016), single-player video games (Mnih et al., 2015, 2016; Hessel et al., 2017), multi-player board games (Tesauro, 1995; Silver et al., 2016, 2017) and autonomous helicopter control (Abbeel et al., 2007).

In the general reinforcement learning problem, agents should be able to act in environments with discrete or continuous actions, or any combination thereof. In this thesis, we only consider environments with discrete actions. For Atari games, agents choose the vertical and horizontal directions of the joystick (Up/Neutral/Down and Left/Neutral/Right, respectively). Some games also utilize the action button (Neutral/Pressed), for a total of 18 possible actions. However, some games do not respond to certain button presses, so the 18 possible actions is simply an upper bound.

An agent may not have perfect information about the current state, and must rely on *observations* of the current state. In the Atari environment, for example, the agent does not know the exact state of the game, but must rely only on the pixel values displayed on the screen. Observations in the Atari environment generally map one-to-one to the current state, as explained in the following section.

2.1 Formulation

2.1.1 Markov Decision Processes

Reinforcement learning problems are formally described in terms of a stochastic Markov Decision Process (MDP), which are defined as the 4-tuple (S, A, p, γ), where:

- S is the set of all possible states,
- A is the set of all possible actions,
- $p(s_{t+1}, r_t \mid s_t, a_t)$ defines the probability of transitioning to state $s_{t+1} \in S$ with reward $r_t \in \mathbb{R}$ after taking action $a_t \in A$ from state $s_t \in S$,
- and $\gamma \in [0, 1]$ is the discount factor, which determines how important immediate rewards are in comparison to distant ones.

We can slightly simplify this nation for deterministic environments, where the transitions and rewards are uniquely determined by the current state and actions. By using functional notation, we thus denote a deterministic MDP with the 5-tuple (S, A, T, r, γ), where:

- S, A, and γ are identical to that of the stochastic MDP definition,
- $T : S \times A \to S$ is the transition function,
- and $r : S \times A \to \mathbb{R}$ is the reward function.

Since we consider only deterministic environments for the remainder of this thesis, we will use the term MDP to refer strictly to deterministic MDPs.

A Partially Observable Markov Decision Process (POMDP) is a variant of an MDP where the agent does not have full knowledge of the exact state of the environment, but instead perceives the environment via *observations*. This notion is formalized by the 7-tuple $(S, A, T, r, O, \phi, \gamma)$, where:

- S, A, T, r, and γ are identical to that of the MDP definition,
- O is the set of possible *observations*,
- and $\phi : S \to O$ is the observation function.

For example, when playing an Atari game, the state of the game is uniquely determined by the contents of the machine's random access memory (RAM). However, an agent only has knowledge of the pixels on the screen, not the exact contents of the RAM. In Atari environments, the set of observations O is the set of all possible values for the $160 \times 210 \times 3$ RGB pixels, and the observation function ϕ maps RAM states to a single image.

As mentioned previously, the screen images of Atari games (which the agent can observe) generally map one-to-one to the contents of RAM (the true state of the environment). However, there are some elements which cannot be uniquely determined from a single screen image, such as the direction of movement and behavior of enemies in games such as Ms. Pacman, Qbert, and Frostbite.

Additionally, games where the player controls the camera are similarly prone to having multiple true states mapping to a single static image. This includes three-dimensional games, such as Battlezone or Robotank, as well as games which feature exploration through a series of rooms, such as Montezuma's Revenge or Pitfall. However, none of the environments we evaluate in this thesis use an agent-controlled camera.

2.1.2 Agent-Environment Interaction

At any given time step, the environment is situated in a single state, s_t. An agent interacts with the environment by selecting an action from the action set, $a_t \in A$. The environment provides the agent with reward $r_t = r(s_t, a_t)$, and transitions the current state to state $s_{t+1} = T(s_t, a_t)$.

Depending on the type of environment, this action-selection process is then repeated until a terminal state is reached (*episodic environments*) or ad infinitum (*continuous environments*[1]). An episodic environment can be thought of as a special case of a continuous environment, where the terminal state always transitions to itself with 0 reward.

2.1.3 Return and γ

The *return*, R_t, describes the discounted future reward during a run after a given time step:

$$R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i}. \tag{2.1}$$

For episodic environments, this is identical to:

$$R_t = \sum_{i=0}^{T-t} \gamma^i r_{t+i}, \tag{2.2}$$

where T is the maximum time step of the episode.

Care should be taken not to confuse the *return* with the *reward*. The reward (denoted by lower-case r) is the feedback given to the agent from a single time step, whereas the return (denoted by upper-case R) is the discounted result from all future time steps.

Larger values of the discount factor, γ, cause the agent to be more concerned with the value of future rewards, whereas lower values cause the agent to care more about immediate rewards. In the extreme case of $\gamma = 0$, the agent only attempts to maximize the reward for the very next time step.

The discount factor is necessary for continuous environments to be tractable, since it keeps future rewards finite. However, it is also commonly used for episodic environments, since limiting the reward horizon has proven to be beneficial in assigning credit to action choices.

2.1.4 Policy and Value

An agent's *policy* $\pi : S \to A$ describes what action an agent takes when it encounters a given state. The *state-value function* $V^\pi : S \to \mathbb{R}$ (also known as the *V-function*) describes the expected discounted return given a specific policy:

[1]Not to be confused with environments with continuous action- or state-spaces.

$$V^\pi(s) = E_\pi \left[R_t \mid s_t = s \right]. \tag{2.3}$$

Similarly, the *action-value function* $Q^\pi : S \times A \to \mathbb{R}$ (also known as the *Q-function*) describes the expected discounted return if an action is taken (and then the policy is followed thereafter):

$$Q^\pi(s,a) = E_\pi \left[R_t \mid s_t = s, a_t = a \right]. \tag{2.4}$$

Note that $V^\pi(s) = Q(s, \pi(s))$; that is, the V-function describes the *on-policy* behavior of the agent. The Q-function, on the other hand, can be used to describe *off-policy* behavior.

In reinforcement learning, the formal goal of the agent is to devise a policy which maximizes the value function for each state. The optimal policy (the one which maximizes the expected return) is typically denoted π^*, and the optimal value functions are denoted by V^* and Q^*.

2.2 Comparison with Other Machine Learning Disciplines

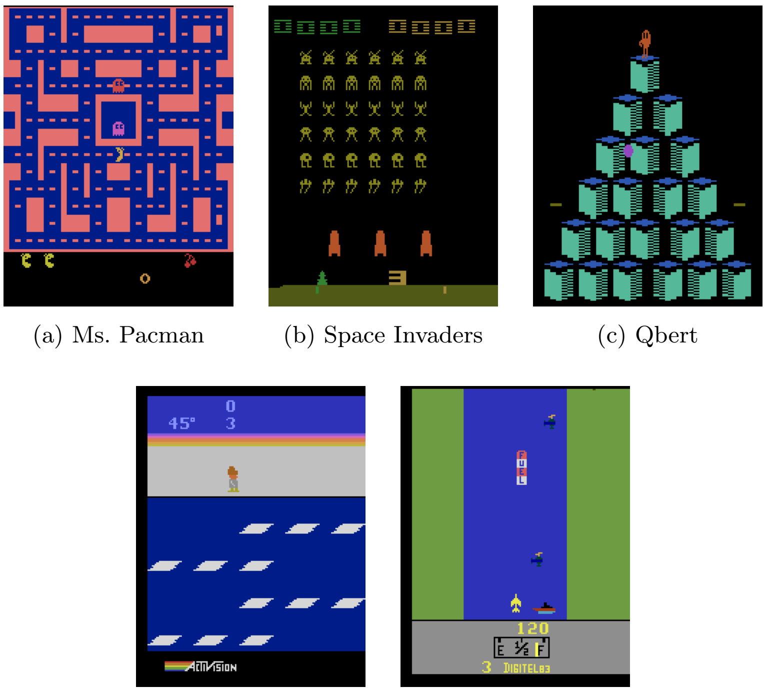
Reinforcement learning problems are distinct from other well-explored problem types in machine learning, such as unsupervised learning and supervised learning. Unsupervised learning problems ask the system to discover latent features of a dataset without explicitly-defined labels. Identifying clusters in or reducing the dimensionality of an unlabeled dataset are examples of unsupervised learning problems. Given that reinforcement learning does make use of a type of label (in the form of rewards from the environment), it is clearly a different type of problem.

In supervised learning problems, the goal is to learn from a set of labeled data so as to correctly predict future unlabeled data. For example, a supervised learning problem may ask to predict the breed of a dog given an image of the animal or the selling price of a house given its property and demographic information. In any supervised learning problem, the system has immediate access to the "correct" label for all inputs during training. In contrast, feedback for reinforcement learning problems is given only through the indirect mechanism of reward, and typically does not include access to demonstrations of expert behavior. Thus, to solve general reinforcement learning problems the agent must be able to learn from its own behavior.

Further, neither supervised nor unsupervised learning problems typically involve a trade-off between exploration and exploitation, which is an important consideration in reinforcement learning.

2.3 Environments

In this thesis, we evaluate our results on the Atari environment, using a collection of five different Atari games sampled from the Arcade Learning Environment (Bellemare et al., 2013), as implemented by the OpenAI Gym (Brockman et al., 2016). We chose these games to be the same as those evaluated in (Blundell et al., 2016), namely Ms. Pacman, Space Invaders, Frostbite, Qbert, and River Raid.



(a) Ms. Pacman (b) Space Invaders (c) Qbert

(d) Frostbite (e) River Raid

Figure 2.1: In-game screenshots from the Atari 2600 versions of environments used in this thesis.

Various environments challenge agents in different ways. One rough taxonomy of environments, proposed by (Bellemare et al., 2016), organizes the suite of games from the Arcade Learning Environment based on two criteria: the importance of exploration and the type of reward schedule. Of the environments we explore in this thesis, three fall under the category of "hard exploration" but "dense reward" (namely, Ms. Pacman, Frostbite, and Qbert), with the other two falling under "easy exploration" with "human-optimal" rewards (Space Invaders and River Raid).

We display screenshots from each of these games in Figure 2.1. We give a basic description and point out some salient features of each of these five environments in Appendix A.

2.4 Deep Q Networks

Mnih et al. (2013) introduced the Deep Q Networks (DQN) algorithm, which uses a neural network to estimate the Q-function. This algorithm showed promising results on a set of 7 classic Atari 2600 games. One defining characteristic of DQN is that it uses an experience replay buffer to store prior experience. This replay buffer stores transition tuples: the previous state, the action taken from that state, the reward received after taking that action, and the next state after taking that action. During each training phase, the network is trained on a minibatch of samples from this replay buffer.

Algorithm 1 Deep Q Networks (DQN)

```
1:  Initialize replay memory, $D$
2:  Initialize the Q-function, a CNN with random weights, $\theta$
3:  Initialize the target Q-function, a CNN such that initially $\theta^- = \theta$
4:  Let $\phi$ be some state preprocessing function
5:  stepCount $\leftarrow 0$
6:  for episode $\leftarrow 0, 1, 2, \ldots,$ numEpisodes $- 1$ do
7:      Observe the initial screen image, $o_0$
8:      $s_0 \leftarrow \phi(o_0)$
9:      for $t \leftarrow 0, 1, 2, \ldots, T - 1$ do
10:         if rand() $< \epsilon$ then
11:             $a_t \leftarrow$ a random action
12:         else
13:             $a_t \leftarrow \text{argmax}_a Q_\theta(s_t, a)$
14:         Take action $a_t$
15:         Observe immediate reward, $r_t$, and resulting screen image, $o_{t+1}$
16:         $s_{t+1} \leftarrow \phi(o_{t+1})$
17:         Store the tuple $(s_t, a_t, r_t, s_{t+1})$ in $D$
18:         Sample a random minibatch, $(s_j, a_j, r_j, s_{j+1})$, of transitions from $D$
19:         $Q_{\text{target},j} \leftarrow r_j + \gamma \max_{a'} Q_{\theta^-}(s_{j+1}, a')$
20:         $\text{error}_j \leftarrow \max(-1, \min(1, Q_{\text{target},j} - Q(s_j, a_j)))^2$
21:         Perform gradient descent on $\text{error}_j$ with respect to $\theta$
22:         stepCount $\leftarrow$ stepCount $+ 1$
23:         if stepCount$\%C == 0$ then
24:             $\theta^- \leftarrow \theta$
```

The experience replay buffer helps to avoid issues caused by training updates which contain highly correlated inputs, which is common in most reinforcement learning problems where each individual action usually only has a minor effect on the environment. This high degree of correlation breaks assumptions made by many regression models, causing naive neural network-based approaches to diverge and making the agent's policy unstable. Sampling from an experience replay buffer helps to avoid these correlations and makes agent training much more stable.

The DQN algorithm uses a convolutional neural network (CNN, (LeCun et al., 1995)) as a function approximator to predict Q-values based on the 160×210-pixel screen images. These images are first subjected to a preprocessing step, which scales the image such that the input to the network is a 84-by-84 greyscale image. Instead of using a separate network for each possible action, DQN uses a single network whose final layer has a separate output for each possible action.

At each time step, the agent completes a forward pass of the input through its network, obtaining a list of predicted Q-values. The agent uses these values to select an action according to an epsilon-greedy strategy. After each environment interaction, the agent stores the observed transition (the 4-tuple of previous state, action, reward, and next state) in its experience replay buffer. The agent then samples the replay buffer to obtain a minibatch on which to train. DQN uses one-step Q-learning as a target value for training:

$$Q_{\text{target}}(s,a) = r_t + \gamma \max_{a'} Q(s',a').$$

Finally, the RMSProp algorithm (Tieleman and Hinton, 2012) is used to update the network according to the gradient of the temporal difference errors with respect to the network weights:

$$\text{error} = Q_{\text{target}}(s,a) - Q(s,a).$$

It should be noted that the magnitude of the gradient is "clipped", restricting all positive rewards to be 1, and all negative rewards to be -1 (zero rewards remain at 0).

Another feature of DQN is that it uses a frame skip parameter to repeat an action N times before the agent decides on a new action. Mnih et al. (2013) state their usage of the frame-skip technique is motivated by computational concerns[2], some evidence suggests that this frameskip parameter itself may assist in learning Atari games by shrinking the effective reward horizon (Braylan et al., 2015).

In the follow-up paper (Mnih et al., 2015), the DQN algorithm is improved upon in two primary ways. Firstly, this paper introduces the "target network": a different set of network parameters which are updated at a slower rate than that of the active network. The target network is only used to compute the target value for the error term when training the network. This increases the stability of training by holding the target values steadier across iterations. The parameters of the active network are copied over to the target network every 10,000 iterations.

The second improvement found in this paper is that the error term is explicitly clipped, in addition to the aforementioned reward clipping. This clipped error term is thus bounded between the range of -1 and 1:

$$\text{error}' = \max(-1, \min(1, Q_{\text{target}}(s,a) - Q(s,a))).$$

Both of the reward and error clipping effectively limit the potential range of the gradients, thus avoiding large updates which risk causing the network to diverge.

Together, these two papers helped usher in the modern era of (deep) reinforcement learning. Since then, more recent improvements have been proposed, such as Double DQN (DDQN, van Hasselt et al. (2016)), Prioritized Experience Replay (Schaul et al., 2015), Dueling Networks (Wang et al., 2016), Asynchronous Advantage Actor-Critic (A3C, Mnih et al. (2016)), Distributional RL (Bellemare et al., 2017), and Noisy Networks (Fortunato et al., 2017). Hessel et al. (2017) integrated these improvements (sans A3C) into a single agent, considerably improving agent performance across a large selection of Atari games. The recent Deep Quality-Value (DQV) Learning algorithm simultaneously learns the state value function alongside the state-action value function to accelerate training (Sabatelli et al., 2018).

[2]"Since running the emulator forward for one step requires much less computation than having the agent select an action, this technique allows the agent to play roughly N times more games without significantly increasing the runtime."

2.5 Model-Free Episodic Control

Algorithm 2 Model-Free Episodic Control (MFEC)

```
1: Initialize KNN buffer for each action, $\text{buff}_a$
2: Let $\phi$ be some state preprocessing function
3: for episode $\leftarrow 0, 1, 2, \ldots, \text{numEpisodes} - 1$ do
4:     Observe the initial screen image, $o_0$
5:     $s_0 \leftarrow \phi(o_0)$
6:     for $t \leftarrow 0, 1, 2, \ldots, T - 1$ do
7:         if rand() $< \epsilon$ then
8:             $a_t \leftarrow$ a random action
9:         else
10:            $a_t \leftarrow \text{argmax}_a$ QUERY-MFEC$(s_t, a)$
11:        Take action $a_t$
12:        Observe immediate reward, $r_t$, and resulting screen image, $o_{t+1}$
13:        $s_{t+1} \leftarrow \phi(o_{t+1})$
14:    $R_T \leftarrow 0$
15:    for $t \leftarrow T - 1, T - 2, \ldots, 1$ do
16:        $R_t \leftarrow R_{t+1} + r_t$
17:        UPDATE-MFEC$(s_t, a_t, R_t)$
18: function QUERY-MFEC$(s, a)$
19:    if $s \in \text{buff}_a$ then
20:        return $\text{buff}_a(s)$
21:    else
22:        $s_1, s_2, \ldots, s_K \leftarrow \text{buff}_a$.KNEAREST$(s)$
23:        $R_j \leftarrow \text{buff}_a(s_j)$
24:        return $\frac{1}{K} \sum R_j$
25: function UPDATE-MFEC$(s, a, R)$
26:    if $s \in \text{buff}_a$ then
27:        $\text{buff}_a(s) \leftarrow \max(\text{buff}_a(s), R)$
28:    else
29:        $\text{buff}_a(s) \leftarrow R$
```

Although so-called "deep" reinforcement learning methods have produced impressive results, they often require very many environment interactions before human-level results can be achieved. Results are typically stated in terms of performance after 200 million frames of experience, which corresponds to over 38.5 days (926 hours) of experience when played at human speeds[3].

The Model-Free Episodic Control (MFEC) algorithm, proposed by Blundell et al. (2016), explicitly attempts to address the problem of sample inefficiency. The authors posit that the observed sample inefficiency in DQN is due to the neural network updates being slow to adapt to new information, preventing DQN-based agents from rapidly exploiting newly-discovered high-reward trajectories. The authors thus propose to replace the convolutional neural network with a neurologically-inspired *episodic controller*, motivated by the instance-based learning of episodic memory in the hippocampus.

[3]For human play, the Atari simulator produces 60 frames per second.

For states which have been previously visited, the episodic controller operates identically to tabular reinforcement learning methods. In particular, the episodic controller maintains a table of state-action tuples as keys which map onto a single real-valued output, representing the expected return after taking the specified action from the specified state.

However, the episodic controller differs from tabular methods in its ability to extrapolate to unvisited states, which is not a capability of (most) tabular reinforcement learning methods. The episodic controller extrapolates a Q-value for novel states by using the previously-stored items in the agent's memory as elements in a nearest neighbor model. This simple model utilizes the inductive bias that similar states contribute more to the prediction of a novel state than those which are dissimilar. When encountering a previously-unseen state, the episodic controller queries for the k items stored in its memory which are closest to the state (according to some distance metric) and returns the average of their associated returns.

It is notable that the regression model MFEC uses for its Q-value approximation is able to exactly recall a given example after seeing that example only once. This behavior, known as *immediate one-shot learning* (Mathy et al., 2015), does not occur in typical neural network approaches, which require gradually training over many examples in order to learn. As mentioned previously, this slowly-adapting learning of neural networks is hypothesized to cause the sample inefficiency of DQN-based approaches. Due to this property, we expect MFEC to be more sample-efficient, particularly in early episodes. However, we do note that MFEC's Q-value approximation does have limits in its generalization ability. Thus, we might expect MFEC's Q-value approximation to have reduced long-term capacity when compared to DQN approaches.

The episodic controller makes the assumption that its environment is deterministic. That is, taking an action from a state always transitions to the same state (or, alternatively, that the environment's transition probabilities for each state and each action are non-zero for only one output state). In a deterministic environment it is always possible to recreate an experienced trajectory by replaying the exact same sequence of actions. This assumption allows the agent to be highly optimistic in its updates: rather than taking a running average of the observed returns (i.e. $Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha R_t$, as is common in tabular reinforcement learning methods), it is possible to simply update the table with the maximum of the new and old values (i.e.: $Q(s_t, a_t) \leftarrow \max(Q(s_t, a_t), R_t)$, see UPDATE-MFEC of Algorithm 2). Doing this in a non-deterministic environment would encourage risky behaviors, such that the agent might over-pursue a high reward which the environment only rarely transitions to.

Modern computers do not have enough memory capacity to store the exact raw representations of environment observations in sufficient quantities (for high-dimensional observation spaces). Because of this, the episodic controller must employ some dimensionality reduction method to project the raw representation to a lower-dimensional space, thus reducing memory demands. Blundell et al. (2016) explores two different dimensionality reduction methods: random projection and variational autoencoder (VAE) latent space.

The random projection method maps the input according to the function $\phi(x) = \mathbf{A}x$, where x is a D-dimensional vector and $\mathbf{A} \in \mathbb{R}^{\mathbf{F} \times \mathbf{D}}$ such that $F < D$ and the elements of $\mathbf{A}$ are drawn from a standard Gaussian distribution. According to the Johnson-Lindenstrauss lemma, the distance in this reduced space approximates the relative distances in the original space: $d(x_1, x_2) \approx d(\phi(x_1), \phi(x_2))$ (Johnson and Lindenstrauss, 1984).

The VAE latent space method first takes 1 million random actions in the environment, and then uses this dataset of observations to train a VAE (Kingma and Welling, 2013) in an unsupervised manner. VAEs are related to standard autoencoders, both of which use neural networks to attempt to reproduce a given input as output while propagating the signal through a comparatively narrow "bottleneck" layer. The output of this bottleneck layer, known as the latent space of the autoencoder, is by definition of lower dimensionality than the input space. Standard autoencoders consist of an *encoder*, which transcribes an input to a latent vector, and a *decoder*, which transforms a latent vector back to the original input space. However, instead of the encoder producing a single deterministic latent vector, VAE encoders output a distribution over the latent space which is then in turn sampled to produce the input for the decoder.

Thus, this latent space presumably represents latent features in the environment, making it a reasonable candidate to be used for dimensionality reduction. However, it should be noted that this latent space is trained only in an unsupervised manner and ignores any information present from the known labels (i.e. the associated returns). It is likely that the VAE will capture extraneous information which is relevant for reproducing the image but not for predicting the future return (e.g. the previously-attained score, which is present in all Atari screens).

Upon empirical evaluation, Blundell et al. (2016) found that the random projection reduction method often outperformed the VAE latent space method, particularly for Atari games. Thus, for the remainder of this thesis, we focus only on the random projection method.

The authors of Blundell et al. (2016) use a discount rate of $\gamma = 1$, making future rewards entirely undiscounted. Additionally, the authors set the exploration rate for epsilon-greedy exploration to be $\epsilon = 0.005$, which is an order of magnitude lower than that used for DQN agents. They found that this lower exploration rate helped the agent exploit its knowledge better. Despite common wisdom that such reduced exploration would cause detrimental effects due to pursuing less route for exploration, their method performs significantly better than DQN approaches on Atari games and 3D maze navigation.

In practice, the episodic controller employs an approximate nearest neighbor data structure to reduce the computational burden of performing nearest neighbor lookups for novel states. As a further implementation detail, the controller actually maintains a separate nearest neighbor data structure per-action. We describe a novel approximate nearest neighbor data structure designed to accommodate MFEC's data access patterns in Chapter 3. In this chapter, we also investigate the effect of noise introduced by these approximate queries on the performance of MFEC.

The regression method employed in the episodic controller is a simple form of k-nearest neighbor (kNN) regression. This is the simplest of a family of regression methods known as *local regression*. In Chapter 4 we explore other regression algorithms in this family, and compare their results against the KNN regression used in the original MFEC model.

However, even after reducing the size of the observation, storing every frame encountered would still require too much memory. Thus, the storage buffer containing the previously-seen frames has a fixed memory limit. Once this limit is exceeded, old memories are evicted until the buffer is once again within the limit. The authors state that the removal of older, less-frequently accessed memories is a phenomenon also observed in human memory (Hardt et al., 2013). In Chapter 5, we expand upon this idea and propose a method of actively identifying and discarding memories which have become outdated in the face of new information.

2.6 Summary

In this chapter, we formally defined the reinforcement learning problem and gave descriptions of the environments which we use as testbeds throughout this thesis. We also gave a brief introduction to the state-of-the-art neural approaches in the form of DQN and its variants. Finally, we also introduced the MFEC algorithm, which we explore and expand upon throughout the remainder of this thesis.

Chapter 3

Efficient Nearest Neighbors Search

In many different domains, it is useful to know which data points are closest to some query point. Such local information can be used for purposes such as density estimation, clustering, classification, and regression. Of particular interest for this thesis, MFEC uses this local information to estimate the action-value function for reinforcement learning.

The k-nearest neighbors problem is to find the k elements in some dataset whose distance to some query point is the smallest. Nearest neighbor (NN[1]) data structures attempt to solve this problem efficiently. For algorithms such as MFEC, using an efficient NN search algorithm can greatly reduce the computational burden required to select actions.

For high-dimensional datasets, it is exceedingly difficult to find the *exact* nearest neighbors, a manifestation of the so-called "curse of dimensionality". An intuitive explanation of this phenomenon is that the volume of the hypersphere of constant radius grows exponentially as the dimensionality increases ($V \propto R^d$).

As a result, approximate nearest neighbor (ANN) algorithms are widely used in practice, since it is often sufficient to simply obtain a representative sampling of the local neighborhood. ANN algorithms make a trade-off between accuracy and computation time. In this thesis, we measure accuracy in terms of 10-nearest neighbor precision (of the actual 10 nearest neighbors, how many were found?), and we measure computation time in terms of queries per second.

Using NN search for MFEC introduces an additional requirement for the data structure: new data is constantly added. Many NN data structures assume that the data is entirely present during initial construction. Such "top-down" or "batch" algorithms can still be used for MFEC, but would require rebuilding after every epoch. In practice, we found that such rebuilding was prohibitively expensive.

An ideal NN data structure for MFEC should be able to incrementally add data without having to entirely re-index, thus avoiding such expensive rebuilding steps. Additionally, in Chapter 5 we propose an extension which requires the deletion of arbitrary data points, a requirement we keep in mind when designing

[1]In this thesis, the abbreviation NN will always refer to nearest neighbors. If neural networks are referenced, it will not be abbreviated.

our solution.

For the remainder of this chapter, D represents a d-dimensional dataset (currently) containing N elements. We denote the ith example from dataset D by the vector $\boldsymbol{D_i} \in \mathbb{R}^d$, and the query point by the vector $q \in \mathbb{R}^d$. We assume that the distance between two elements is given by the standard Euclidean distance, denoted as $d(\cdot, \cdot)$.

We present full algorithm descriptions for all algorithms mentioned in this chapter in Appendix B. In some algorithm descriptions we use C-style ternary statements of the form "a ? b : c", which should be interpreted as "if a is true, then use b, else c."

Finally, when we refer to a "max-heap of size k", we mean a binary tree data structure with the property each parent's key is larger than those of its children. This is sometimes referred to as a priority queue. Whenever the $k + 1$th element is added, the max-heap discards the largest item. Thus, the heap always contains the k elements with the smallest keys.

3.1 Batch Methods

Nearest-neighbor algorithms generally fall into one of four broad families:

- brute force,
- locality-sensitive hashing,
- neighbor graph search,
- and hierarchical (tree-based) space partitioning.

In the following subsections, we describe these various algorithm families. Since the solution proposed in this thesis takes a tree-based space partitioning approach, additional emphasis will be placed in its description relative to the other families.

3.1.1 Brute Force Search

The simplest NN algorithm is brute force search, where each element of the dataset is compared to the query point. Algorithm 4 gives pseudocode defining brute force search.

Brute Force does not require a rebuilding step, and thus is equally well suited for online datasets as it is for offline ones. However, it is extremely slow, with a complexity of $\mathcal{O}(N \times d)$.

Some implementations use low-level optimizations (in particular, the BLAS library) to improve the runtime of brute-force search. Additionally, since the problem is inherently parallel, GPUs have been used to dramatically improve performance (Garcia et al., 2008; Johnson et al., 2017).

3.1.2 Locality-Sensitive Hashing

Locality-sensitive hashing (LSH) operates by computing a *hash* for each element of the dataset. Algorithms 5 and 6 give pseudocode defining locality-sensitive hashing search.

Each element is then placed into a bucket, according to its hash, analogous to the classic hash table data structure. However, while hash tables typically use hashing algorithms which reduce the chance of collision, LSH methods use hashing algorithms which are similar for items which are close together (as the name suggests).

An example of such a locality-sensitive hashing algorithm is comprised of a set of N sub-hashes: $H = \{h_1, h_2, \ldots, h_N\}$. Each of these sub-hashes uses different random projections onto a single 1-D real value, and returns 1 if this projection is positive, and 0 otherwise. That is, $h_i = \mathbb{1}[\boldsymbol{a} \cdot \boldsymbol{x} > 0]$, where $\boldsymbol{a}$ is a vector with elements drawn from a standard Gaussian distribution. Concatenating each of the sub-hashes results in a N-bit hash (Wang et al., 2014).

At build-time, each element in the dataset is hashed and then added to the corresponding bucket in the hash table, as described in Algorithm 5. At query-time, the query is processed using the same hash function, and all the contents of the corresponding bucket are added to the output heap, NBR. To increase search coverage, additional hashes are also computed from the true hash (e.g. by random bit-flips), and the contents of those buckets are also added to NBR (Algorithm 6, lines 5-7). For greater coverage, some implementations use multiple hash tables with different random hashes, and then search through all those tables at query time.

3.1.3 Neighbor Graph Search

Neighbor graph search algorithms first iterate through the data to build a directed graph data structure. Algorithm 7 gives pseudocode defining neighbor graph search.

After this construction step, each data point has edges leading from itself to its (approximate) nearest neighbors within the dataset. Finding the inter-dataset nearest neighbors for each element can be accomplished through the use of an external ANN algorithm. Various implementations also propose alternate algorithms for incrementally building the graph via the previously-constructed graph itself (Malkov et al., 2014; Malkov and Yashunin, 2016; Fu and Cai, 2016).

Searching requires a set of initial points from which to start the search. Selecting these initial points can either be done randomly, or through an external (low-accuracy) ANN algorithm. The search itself proceeds similarly to Dijkstra's shortest path algorithm, where at each step the minimum element (in this domain, the one closest to the query) is visited and expanded. The search stops once the kth nearest found point is closer than the nearest point in the frontier.

3.1.4 Hierarchical Space Partitioning

Hierarchical space partitioning algorithms all recursively divide the search space into one or more contiguous regions at each level, forming a tree structure. The exact construction and partitioning strategies are what distinguish the various algorithms from each other.

3.1.4.1 k-d Tree

k-d trees (short for k-dimensional trees) attempt to solve the k-nearest neighbor problem by hierarchically partitioning the search space. Specifically, k-d trees use axis-aligned splitting hyperplanes: each inner node splits the data according to a single feature. Since the split point is the median of the dataset, each level in the tree exactly splits the dataset in two, resulting in trees which are height-balanced. The k-d tree construction algorithm presented in Algorithm 8 cycles through the axis of splitting incrementally, but alternate implementations could choose the splitting axis dynamically (e.g. by selecting the axis of greatest variance).

The k-d tree search algorithm presented in Algorithm 9 reduces the computational cost of exact searches by pruning branches of the tree which cannot have closer neighbors than the current contents of NBR. The key observation is that if the distance from the query point to the splitting hyperplane is d_{split}, then all elements on the other side of the hyperplane must be at least d_{split} units away from the query point. Thus, at any given point in the search, if the distance to the kth nearest neighbor is less than that to the splitting hyperplane, it is guaranteed that none of the elements in that branch will be closer.

The ε parameter allows for approximate searches while still preserving some theoretical guarantees. If the true distance to the kth nearest neighbor is d_k, then all returned elements are guaranteed to be at most $d_k * (1 + \varepsilon)$ from the query point.

Since each level of the tree compares just a single dimension, the performance of k-d trees degrade as the dimensionality grows, as it fails to adapt to the intrinsic dimensionality of the dataset (Dasgupta and Freund, 2008).

3.1.4.2 Random Projection Tree

Random projection trees (RP trees, Dasgupta and Freund (2008)) are another type of binary space partitioning tree. Instead of only comparing with a single dimension at each level, a random splitting hyperplane is used. The direction of this hyperplane is defined by a random unit vector in d-dimensions, then the dot products between each element and the hyperplane direction is computed. The median value of these dot products is chosen to be the bias, shifting the hyperplane such that it evenly splits the data. Finally, each datapoint is assigned a child node according to which side of the hyperplane it falls in. Algorithm 10 gives pseudocode defining the construction of a RP tree.

Search in a RP tree is identical to that in a k-d tree, except the distance calculation to the separating hyperplane is adjusted to account for non-axis-aligned hyperplanes.

3.1.4.3 Mean Tree

Mean trees (Nister and Stewenius, 2006) maintain a set of cluster centroids at each level. During construction, a simple clustering algorithm (such as k-means) is used to divide the space into a predefined number of clusters. The hyperplane of all points equidistant to two cluster means defines the splitting hyperplane between those two centers. Each point in the dataset is assigned to the cluster whose center is closest to that point.

Search in mean trees works similarly to that in k-d trees and RP trees, except for a slight modification to accommodate for the arbitrary branch factor. The child nodes are first sorted in order of increasing distance from the query point, and the closest cluster is always searched through first. Then, the remaining clusters are searched in sorted order, since neighbor points are more likely to be in close clusters than far away ones. Once the distance to a separating hyperplane is greater than the distance to the kth nearest neighbor, that cluster can be safely pruned, for the same reason described in Section 3.1.4.1. Moreover, further away clusters don't need to be checked at this point, since it's guaranteed that the distance to their hyperplanes will be further away, so we can safely terminate the search at this point (line 14 of Algorithm 12).[2]

Mean trees have also been used in the context of compression, where it is known under the names of "hierarchical vector quantization" (Gersho and Shoham, 1984; Vishwanath and Chou, 1997) and "tree-structured vector quantization" (Geva, 2000; Wei and Levoy, 2000). Mean trees have been found to be more adaptive to intrinsic dimension relative to other spatial partitioning trees, which cause them to produce lower quantization errors and more accurate nearest neighbor lookups (Ram and Gray, 2013; Verma et al., 2009).

3.2 Online Methods

3.2.1 Online Mean Trees

Encouraged by the results of Ram and Gray (2013) and Verma et al. (2009), we first attempted to adapt mean trees to be used in an online context. Our first attempt (which we refer to as online mean trees) uses a simple splitting behavior: when a leaf node exceeded capacity, it converts itself to an inner node and creates p new children leaf nodes via k-means. Algorithm 13 describes this behavior in pseudocode.[3]

It should be noted that on line 14 of Algorithm 13, for clarity the pseudocode does not include the exact steps for keeping track of the cluster centers in an online setting. The implementation has two auxiliary variables for each cluster: a vector containing the total sum of the positions of all that node's children, as well as a count for the number of children contained within the node. By just adding the input vector to the running sum and incrementing the counter, the mean position for the node can be computed without having to re-evaluate every child leaf item.[4]

One problem with this approach is that the resulting tree is not guaranteed to be balanced. In practice, we found that online mean trees produced dramatically lopsided trees, especially for higher dimensions. In some cases, the online trees devolved to near-linked lists, such that at most levels, all children except one were leaves. This drastically reduces the performance of the data structure, in terms of build time, query time, and memory usage.

Batch mean trees also don't have any guarantees about tree balance, but in practice the batch trees tend to be rather balanced. This difference between observed behavior of the batch and non-batch versions is presumably due to the

[2]This behavior is implemented in lines 623 to 664 of vqtree.cpp of LeKander (2017b).
[3]This behavior is implemented in lines 835 to 1345 of vqtree.cpp of LeKander (2017b).
[4]This behavior is implemented in onlineaverage.cpp of LeKander (2017b).

lack of data (initially) during each split. Since batch mean trees can split the entire dataset, they can better match the true distribution of the data. However, online batch trees have significantly less data at the time of splitting (exactly *maxLeafSize* datapoints), and as such they risk not capturing the underlying distribution of the entire dataset.

As a result of these lopsided trees, we investigated alternate construction algorithms with stronger balancing guarantees.

3.2.2 K-trees

The K-tree algorithm[5] (Geva, 2000) is a Hierarchical Vector Quantization construction algorithm which incorporates B-tree splitting behavior to produce trees which are height-balanced. The K-tree algorithm is a bottom-up algorithm, incrementally building the data structure by adding each element one at a time. After each insertion, the tree is guaranteed to have the following properties:

- All leaf nodes are located at the same level of the tree.
- Each leaf node contains at most *maxLeafSize* (a hyperparameter) elements.
- Each inner node contains at least 1 child, and at most *branchFactor* (another hyperparameter) children.
- Each inner node maintains the average location of all items in eventual successor leaf nodes (the cluster centers).

When inserting a new element into an existing K-tree, first a greedy search (lines 3-6 of Algorithm 14) is performed to find the closest leaf node. The new element is then appended to the contents of that leaf node. If the leaf node size is greater than *maxLeafSize*, then it splits into two new nodes using k-means, and both are added to the parent node. Similarly, if the parent node contains more than *branchFactor* children, it splits, forming clusters based on the centers of its children.

This process repeats up the tree until either there is enough space to accommodate the new node, or until the leaf node is reached. If the root node is full, then a new root node is created, to which the newly-split nodes are added.

Pseudocode describing the K-tree construction algorithm is given in Algorithm 14.[6] In our ad-hoc experimentation, we found that the trees constructed via this algorithm were far superior to those constructed by the online mean trees algorithm.

3.3 KForest

As a result of the success of our initial experiments with K-trees, we decided to expand upon this data structure to use for Episodic Control. We propose a series of extensions (some inspired by other works, some our own invention) to increase the quality of search with K-trees. We use the name “KForest” to refer to our implementation which includes all extensions below.

[5]Not to be confused with k-d trees.

[6]This behavior is implemented in lines 978 to 1344 of vqtree.cpp of LeKander (2017b).

3.3.1 Forest

Instead of using a single tree, our first extension is to maintain multiple trees. This collection of trees collectively form a forest (hence, the chosen name of our implementation). The benefit of using a forest over a single tree is that the individual trees differ in how they partition the search space. This concept is similar to the idea of random forests in decision trees (Breiman, 2001).

Adding elements to a forest is a straightforward extension of the single-tree case: for each tree in the forest, add the new element to that tree. Search in a forest is likewise similar, except the neighbor queue is shared between search of successive trees. This slightly benefits search types which compare against the current kth nearest neighbor (such as Algorithm 12) since later trees have a better estimate to begin with.

For reasons described earlier, it is necessary that the trees are sufficiently different from each other for forests to be beneficial. One source of differentiation is achieved through the k-means splitting. In our implementation (described in Algorithm 15), the k-means initialization is performed according to the following heuristic:

1. Both centers are initially placed at the mean of the data.
2. The ordering of the data is randomized (the data is shuffled).
3. Each datapoint is evaluated one at a time, adding itself to the closest cluster center. In the case of a tie, one center is chosen arbitrarily.

Since the k-means implementation processes each datapoint one at a time, the initial ordering of the points impacts the final result of the clustering. Thus, using different random shuffles for each tree causes the structure of the trees to diverge, even though the data is identical.

In our implementation, the *numTrees* hyperparameter determines the number of trees in the forest.

3.3.2 Consistency

One property of any means tree is something we call *consistency*. A tree is considered to be consistent if for all $\boldsymbol{D_i} \in d$, a *greedy search* (also known as a *defeatist search*) for $\boldsymbol{D_i}$results in a leaf node containing $\boldsymbol{D_i}$. A greedy search is the simplest kind of search: starting from the root, the search progresses down the tree by recursively selecting the closest child node to $\boldsymbol{D_i}$until a leaf node is encountered. Algorithm 17 gives pseudocode defining greedy search in mean trees.

Thus, a tree is consistent if all of the leaf data is located in the correct "bin". Formally, a tree is consistent if and only if $\boldsymbol{D_i} \in \textsc{greedySearch}(\text{root}, \boldsymbol{D_i})$.

While batch mean trees are guaranteed to be consistent, K-trees have no such guarantees. In inner nodes, online K-tree additions can be thought of as a form of online k-means with only a single iteration. The cluster centers "drift" as new data is added, changing the separating hyperplane in the process. This is done without consulting the data in the leaf nodes, which could potentially cause them to be placed into the wrong bin.

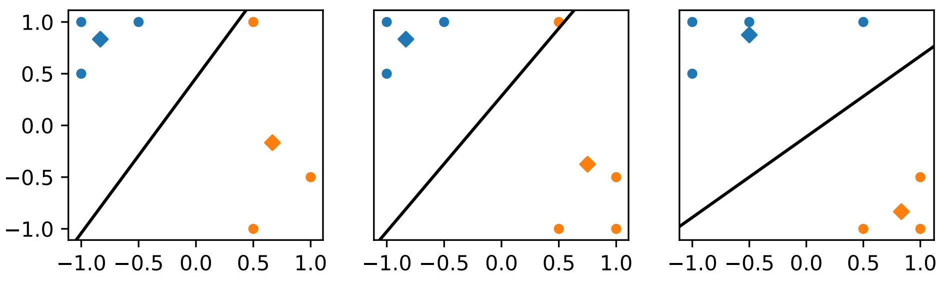


Figure 3.1: Visualization of a 2-dimensional K-tree with 1 level of inner nodes. The node centers are depicted as diamonds, the data points are labeled by color according to which node they have been assigned, and the separating hyperplane is depicted as a black line. (left) Initial consistent tree. (middle) A new point is added at (1, -1) into the orange cluster. Note that the top-most point in this cluster now lies on the opposite side of the separating hyperplane from the cluster center, causing the tree to be inconsistent. (right) The tree is made consistent again by removing the inconsistent point and then re-adding it to the tree.

To illustrate this, consider the example in Figure 3.1. In the left side, two simple clusters are separated according to their respective cluster centers. However, when a point at (1, -1) is added, the orange cluster center is moved down and to the right. This causes the top-most point in the orange cluster to no longer be on the correct side of the separating hyper plane. Thus, we say that the left and right figures represent consistent trees, whereas the middle figure represents an inconsistent tree.

To address this issue, we propose a strategy for enforcing consistency in K-trees. After each addition, we choose previously-added elements from the dataset and see if that data point is still placed into the correct bin by performing a greedy search. If that data point is not found by the greedy search, it is removed from the tree and then re-added. Figure 3.1 (right) depicts a K-tree after such consistency enforcement.

It is possible to ensure full tree consistency by iteratively checking each data point according to the above procedure until the tree is entirely consistent. We refer to this approach as *full consistency*. However, this approach is prohibitively expensive because it requires iterating over the dataset potentially many times after each addition. The second approach uses a limited number of tests per addition to reduce this computational burden. This does not have the same theoretical guarantees that the tree will be consistent, but it does assist with making the tree more accurate while not being an excessive burden. We refer to this approach as *online consistency*.

Our KForest does not implement these consistency methods directly. Instead, we expose methods which perform consistency checks for a single element (namely, `enforceTreeConsistencyFull`, `enforceTreeConsistencyAt`, and `enforceTreeConsistencyRandom`).[7] However, the ECDC agent, which

[7]This behavior is implemented in lines 284 to 310 and lines 425 to 461 of vqtree.cpp of LeKander (2017b)

uses our KForest implementation, has a *tree_consistency_iters* hyperparameter which determines the number of online consistency iterations to perform.[8]

3.3.3 Spill

Thus far, all the trees discussed have assigned each data point to a single leaf node. However, if a data point is close to a separating hyper plane, it may be beneficial for these points to be added to both subtrees. This is a concept known as *spill trees* (Liu et al., 2005), since the spatial partitionings slightly "spill over" into each other.

It is not immediately apparent how the amount of spill should be determined. Liu et al. (2005) use a fixed *overlapping size* which determines the width of the region of overlap. If the distance between a new data point and the separating hyperplane is less than τ (a hyperparameter), then that point is added to both subtrees.

However, this approach causes inner nodes deeper in the tree to have a higher percentage of split items than those on the first layers. This is caused by the nature of the hierarchical space partitioning: by design, later nodes capture smaller regions of the search space. Each level in the tree progressively "narrows" the search space more at each level. In practice, the authors of Liu et al. (2005) use a hybrid tree structure wherein some nodes do not spill if using spill would make any child contain a large percentage of the dataset. Although not mentioned explicitly, this hybrid approach does mitigate this narrowing effect.

The approach used by Liu et al. (2005) uses the usual metric of distance from a plane. Given two cluster centers $\boldsymbol{w}_a$ and $\boldsymbol{w}_b$, the distance from $\boldsymbol{D_i}$to the separating hyperplane between $\boldsymbol{w}_a$ and $\boldsymbol{w}_b$ is given by:

$$\text{planeDist}(\boldsymbol{D_i}, \boldsymbol{w}_a, \boldsymbol{w}_b) = \frac{\left|d(\boldsymbol{D_i}, \boldsymbol{w}_a)^2 - d(\boldsymbol{D_i}, \boldsymbol{w}_b)^2\right|}{2 * d(\boldsymbol{w}_a, \boldsymbol{w}_b)}. \tag{3.1}$$

As depicted in the top row of Figure 3.2, this metric gives different results across different scales. A fixed threshold τ would capture more of the search space for the left figure (with cluster centers at $(-1, -1)$ and $(1, 1)$) than for the right (with cluster centers at $(-5, -5)$ and $(5, 5)$).

We would instead like the width of the spill regions to be effected by the spread of the data at each node. In particular, the position of the cluster centers can be used as one estimate for the variance of the data in that node. We thus propose an alternate metric: instead of using the absolute distance to the separating hyperplane, we propose using a metric which scales according to the distance between two cluster centers. We propose using the relative distance between the closest center and each sibling center:

$$\text{relPlaneDist}(\boldsymbol{D_i}, \boldsymbol{w}_a, \boldsymbol{w}_b) = \frac{\left|d(\boldsymbol{D_i}, \boldsymbol{w}_a)^2 - d(\boldsymbol{D_i}, \boldsymbol{w}_b)^2\right|}{d(\boldsymbol{w}_a, \boldsymbol{w}_b)^2}. \tag{3.2}$$

Intuitively, Equation 3.2 scales Equation 3.1 proportionally to the distance from each cluster center to the plane. One nice property of this metric is that the value at the cluster center is always 1: $\text{relPlaneDist}(\boldsymbol{w}_a, \boldsymbol{w}_a, \boldsymbol{w}_b) = \text{relPlaneDist}(\boldsymbol{w}_b, \boldsymbol{w}_a, \boldsymbol{w}_b) = 1$.

[8]This behavior is implemented in lines 57 to 58 of localreg.py of LeKander (2017a).

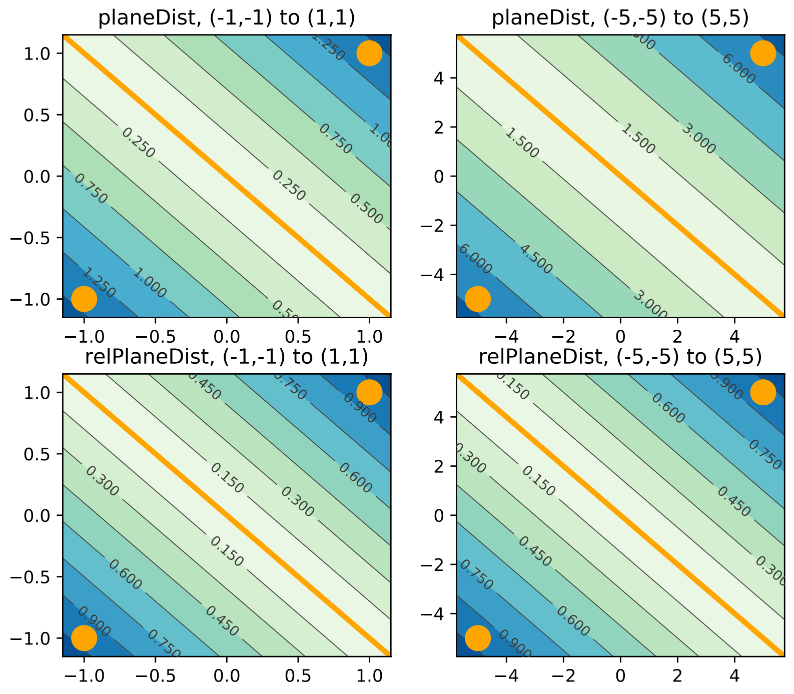


Figure 3.2: Visualization of the planeDist and relPlaneDist metrics. Cluster centers (depicted as orange circles) are located at the corners of the figures. The separating hyperplane between these two centers is given by the line $y = -x$ (depicted in orange). Each contour line represents an increment of 0.1 units.

relPlaneDist can actually be defined in terms of planeDist. If d_a and d_b denote the distances from the separating hyperplane to cluster centers $\boldsymbol{w}_a$ and $\boldsymbol{w}_b$, respectively, then $d_a = d_b = \frac{1}{2}d(\boldsymbol{w}_a, \boldsymbol{w}_b)$. Thus:

$$\begin{aligned}\frac{\text{planeDist}(\boldsymbol{D_i}, \boldsymbol{w}_a, \boldsymbol{w}_b)}{d_a} &= \frac{\left|d(\boldsymbol{D_i}, \boldsymbol{w}_a)^2 - d(\boldsymbol{D_i}, \boldsymbol{w}_b)^2\right|}{2 * d(\boldsymbol{w}_a, \boldsymbol{w}_b) * d_a} \\ &= \frac{\left|d(\boldsymbol{D_i}, \boldsymbol{w}_a)^2 - d(\boldsymbol{D_i}, \boldsymbol{w}_b)^2\right|}{2 * d(\boldsymbol{w}_a, \boldsymbol{w}_b) * \frac{1}{2}d(\boldsymbol{w}_a, \boldsymbol{w}_b)} \\ &= \frac{\left|d(\boldsymbol{D_i}, \boldsymbol{w}_a)^2 - d(\boldsymbol{D_i}, \boldsymbol{w}_b)^2\right|}{d(\boldsymbol{w}_a, \boldsymbol{w}_b)^2} \\ &= \text{relPlaneDist}(\boldsymbol{D_i}, \boldsymbol{w}_a, \boldsymbol{w}_b).\end{aligned}$$

The relPlaneDist metric is depicted in the bottom row of Figure 3.2. Note that the metric scales identically to the scaling of the cluster centers. Namely, the scale of the axes of the left image is at 1/5th the scale of the one on the right, but the contour lines are identical.

These trees introduce a new hyperparameter, the *spill* parameter. This parameter determines how close to the separating hyperplane you have to be in order to be added to both subtrees. The setting of this hyperparameter highly affects the performance of the tree, since it affects the number of duplicates that are present.

It should be noted that some implementation details must be slightly tweaked in order to accommodate for spill trees. One notable consideration is that the search procedure must be tweaked slightly to account for duplicates. In our implementation we use the `unordered_set` data structure to ensure that no duplicates are added to the max-heap.[9]

3.3.4 Labels and Data Storage

Since we are specifically interested in using the KForest implementation for local regression purposes, we decided to have each datapoint associated with a real-valued label. For MFEC, this label is the return achieved by the agent after encountering the observation represented by the data.

It should also be noted that the KForest implementation keeps a copy of each input datapoint which is shared across trees. The forest maintains two circular buffers: one which holds the datapoints and another for their associated labels. The *maxSize* hyperparameter determines the maximum size of this circular buffer. The maximum and minimum indices in the circular buffers are tracked by the `headNdx` and `tailNdx` variables.

Instead of storing the full data vector, each leaf node holds just a single index, indicating where in the forest's internal storage the associated element can be found. This reduces the memory requirements, since it would be redundant for the leaf data to be duplicated for each tree in the forest (and potentially multiple leaf nodes in a single tree, if spill is used).

Currently, the exact indices for each element can be tracked via the return values from the various methods which modify the elements contained in the forest. The `add` method returns the index of the most-recently added element (which is guaranteed to be the value of *headNdx* after the addition). The `nearestNeighborNdxes` method returns a list of indices representing the result of the nearest neighbor search. The values associated with an index can be obtained via the `getData` method, and the value can be obtained via that `getLabel` method. The `clearAndReplace` method (explained in the next section) returns the index that was promoted as a result of deleting that element.

An example of using this information to maintain additional information for each item is given in the `add` and `clear` methods of localreg.py of LeKander (2017a). However, this logic is quite tenuous and can be prone to errors. Thus, a future version of the KForest implementation will likely provide machinery to abstract away this implementation detail such that client users should not have to maintain any indices even if they wish to track metadata associated with each item. Indeed, this future version will support labels beyond simple real values.

3.3.5 Deletion

Our KForest implementation contains a `clearAndReplace` method, which provides the functionality to remove previously-added items from the forest one at a time.[10] This is motivated by an extension to the MFEC algorithm which we

[9]This behavior is implemented in lines 258 to 266 of vqtree.cpp of LeKander (2017b).
[10]This behavior is implemented in lines 196 to 239 of vqtree.cpp of LeKander (2017b).

propose in Chapter 5, which requires the removal of arbitrary elements at any time.

Removing an element from the tree requires first finding the leaf nodes from which the datapoint should be deleted. If the tree were guaranteed to be consistent and no spill was used, then a simple greedy search would suffice to find these leaf nodes. To accommodate for this, the forest maintains a `leafLookup`, an auxiliary array of `unordered_set`s which maps previously-added items to the leaf nodes which contain that element. When each tree addition procedure (Algorithm 14) reaches a leaf node, that node registers itself at the corresponding data index.

Once the leaf nodes are found, the to-be-deleted element is removed from the contents of that node. Then the algorithm traverses up the tree until it reaches the root, correcting the center positions of each node along the way by subtracting itself from them. Note that these center updates can be done in constant time since the centers are stored internally as a sum and a count, as explained in Section 3.2.1.

Two possible strategies for handling deletion from the circular buffer would be to either leave unused "holes" in the buffer or to linearly shift all succeeding elements back by 1. This former strategy would be prohibitively memory-inefficient, whereas the latter could continually shift half of the elements in the dataset in the worst case. We thus use a strategy which ensures the used memory is contiguous and deletions occur in constant-time, at the cost of maintaining strict order: when an item is deleted, the minimum element in the buffer is promoted to the deleted index and the tail pointer is incremented by one.

This has minor implications on the behavior of the MFEC algorithm. Namely, the MFEC algorithm specifies that if a new observation is added when the experience buffer is full (that is, if *maxSize* unique experiences are currently stored), then the oldest experience in the buffer is removed. In our KForest implementation, the experience located at the lowest index is removed. Note that, due to the buffer not strictly maintaining the order of the experiences, this is not necessarily the oldest item in the buffer (although it is guaranteed to have been in the buffer for at least *maxSize* additions).

It should be noted that when duplicate detection is enforced (see the section below), the KForest implementation handles this by deleting the old event and then adding the new one. Thus, removals technically occur quite often, changing the order of items in the buffer even when the end user does not use this method specifically.

3.3.6 Duplicates

The MFEC algorithm treats experiences which exactly match previous experiences as a special case. Thus, we provide machinery in the KForest implementation to perform searches for exact matches. Additionally, special logic is added to handle collisions when duplicates are added. This behavior is controlled by the boolean *removeDups* hyperparameter.

KForset optionally maintains an `unordered_map` to keep an exact index of previously inserted data. When adding each element, a lookup is first performed in the map. If that item exactly matches an item in the map, the matching item is removed and the label is set to the maximum value of the previous label and

Table 3.1: Search algorithm constants

Algorithm Name	**C++ Constant**	**Python Constant**
BRUTEFORCESEARCH	`VQSEARCH_BRUTE`	`vqtree.SEARCH_BRUTE`
MEANTREESEARCH	`VQSEARCH_EXACT`	`vqtree.SEARCH_EXACT`
GREEDYSEARCH	`VQSEARCH_DEFEATIST`	`vqtree.SEARCH_DEFEATIST`
PROTOTYPEDISTSEARCH	`VQSEARCH_PROT_DIST`	`vqtree.SEARCH_PROT_DIST`
PLANEDISTSEARCH	`VQSEARCH_PLANE_DIST`	`vqtree.SEARCH_PLANE_DIST`
LEAFGRAPHSEARCH	`VQSEARCH_LEAFGRAPH`	`vqtree.SEARCH_LEAFGRAPH`

the new label.[11] If the new datapoint does not match any item in the map, then the item is added to all trees as normal.

As noted above, if an exact match is found then the resulting label is taken as the maximum of the new and old labels. The maximum combinator is chosen in accordance to the MFEC algorithm, which keeps an "optimistic" estimate for the estimated return from a state. Future versions of the KForest library will allow the use of custom combinators, allowing more exotic use-cases.

3.3.7 KForest Search

In our KForest implementation, the search is controlled by three hyperparameters: *searchType*, *exactEps*, and *minLeaves*.

The *searchType* hyperparameter determines which of the 6 available search methods should be used. Our KForest implementation allows the user to specify this value in two different ways. The constructor contains a `searchType` parameter, which sets the `defaultSearchType` variable of the forest. This `defaultSearchType` is used by default if no value is given for the optional `searchType` parameter to the `nearestNeighbors` method. On the other hand, if a value is given to this optional parameter at search-time, then that value will be used for that search (but the `defaultSearchType` variable will not be changed for subsequent searches).

The *exactEps* and *minLeaves* hyperparameters both control how accurate the results from the search will be (making a trade-off between accuracy and runtime). The real-valued *exactEps* hyperparameter corresponds to the ε parameter of Algorithm 12, specifying the upper bound on the distance between the k furthest returned element and the actual kth furthest element from the dataset. The integer *minLeaves* hyperparameter specifies the number of leaf nodes that will be evaluated during the search (as explained below).

In addition to BRUTEFORCESEARCH, MEANTREESEARCH, and GREEDYSEARCH (Algorithms 4, 12, and 17, respectively), we implemented three additional search strategies. The first two algorithms, PROTOTYPEDISTSEARCH and PLANEDIST-SEARCH both traverse through the tree in an order specified by some distance metric. LEAFGRAPHSEARCH instead uses spill information to jump between leaf nodes which share at least one element due to spillage.

Our KForest implementation uses predefined constants to switch between each of the implemented search methods. The mapping between algorithms and implementation constants is given in Table 3.1.

[11]This behavior is implemented in lines 141 to 149 of vqtree.cpp of LeKander (2017b).

3.3.7.1 Prototype Distance Search and Plane Distance Search

Greedy search is perhaps the simplest search algorithm that one could imagine for a K-tree, in which only the closest leaf node to the query is evaluated. Since only a single leaf node is evaluated by this, this produces rather inaccurate results. One might wish to extend this idea such that multiple leaf nodes are evaluated.

We thus propose two methods to iterate through the search tree such that multiple leaf nodes are evaluated. These methods have a fixed budget (controlled by the *minLeaves* hyperparameter) of leaf nodes to evaluate, and are thus greatly impacted by the order in which the tree is iterated. Both these methods use some sort of distance metric to concentrate on "close" nodes while excluding "distant" ones. These methods differ in which metric they use.

These search methods begin by descending from the root, following the closest cluster center until a leaf node is reached. Along the way, the cluster centers which were not explored (all those except the closest center) are added to the frontier min-heap, sorted according to the distance from the query. For the next iteration, the minimum (closest) element from the frontier heap is chosen, recursively evaluating all its sub-nodes until a leaf is found. Each iteration results in the evaluation of a single leaf node. Thus, by repeating this process *minLeaves* times, the data from exactly *minLeaves* leaf nodes will be added to the NBR max-heap.

Note that the frontier min-heap is bounded to contain at most *minLeaves* items. This gives a slight speed benefit, since nodes greater than the *minLeaves*-th entry in the frontier are guaranteed to never be evaluated. To accommodate for this functionality, we actually use a sorted `std::multimap` to support removals from both the head and the tail of the list.

Prototype Distance Search[12] (Algorithm 18) ranks the frontier according to the distance from the query point to the cluster center.[13] In contrast, Plane Distance Search (Algorithm 19) ranks the frontier according to the distance from the query to the separating hyperplanes.[14]

For each individual node, these two methods produce the same ranking for a single node. The differentiation comes when comparing between different levels of the tree. Prototype Distance Search is more likely to expand lower nodes than higher nodes, since the cluster centers at these levels are more likely to be closer together due to the progressively shrinking search space. Plane Distance Search, on the other hand, may be more likely to identify situations where a point is at the outlier of a cluster, indicating evaluating a higher-level node would be more beneficial to evaluate.

3.3.7.2 Leaf Graph Search

Leaf Graph Search[15] (Algorithm 20) instead uses the spill property to search through the tree. The leaf nodes can be viewed as nodes in a graph, such that two leaf nodes are considered to be adjacent if they share at least one datapoint. This method does not explicitly traverse the tree during query time (besides the

[12]This terminology is motivated from the vector quantization literature, which commonly refers to what we call cluster centers as "prototypes".

[13]This behavior is implemented in lines 681 to 711 of vqtree.cpp of LeKander (2017b).

[14]This behavior is implemented in lines 714 to 748 of vqtree.cpp of LeKander (2017b).

[15]This behavior is implemented in lines 751 to 778 of vqtree.cpp of LeKander (2017b).

initial greedy search for the closest leaf node), but instead uses properties from tree construction to represent a graph directly between leaf nodes.

One important caveat is that this method requires that the spill hyperparameter is used in order to provide any benefit above greedy search. In our current implementation, the tree spill is the only mechanism which can cause the same datapoint end to up in multiple leaf nodes.

The algorithm operates by first finding the nearest leaf node to the query. For each datapoint in that leaf, the `leafLookup` mapping is consulted to discover which other leaf nodes contain that datapoint. As discussed in Section 3.3.5, the auxiliary variable `leafLookup` already exists for the purposes of deleting items from the forest, which means that no additional auxiliary structures are necessary to support this search. Those adjacent nodes which haven't been evaluated yet are added to the frontier, sorted by distance from the query. Note that an auxiliary `visited` set is used to determine if the proposed leaf node has been evaluated previously, to avoid duplicates in the frontier queue. Once the initial leaf node is exhausted, the next item from the frontier is evaluated.

Similar to the previous section, each iteration evaluates a single leaf node. Thus, by performing this process *minLeaves* times, exactly *minLeaves* distinct leaf nodes will be evaluated.

3.4 SIFT Benchmark

To test the performance of our KForest implementation, we compared its performance against other standard nearest neighbor libraries. Namely, we compared our KTree implementation against the following algorithms:

- Annoy (Bernhardsson, 2017b),
- BallTree (provided by nmslib) (Boytsov and Naidan, 2013),
- Flann (Muja, 2017),
- HNSW Graph (Malkov and Yashunin, 2016),
- KGraph (Dong, 2017), and
- SW Graph (Malkov et al., 2014).

We compare these algorithms against three versions of our KForest implementation, to examine the effects of the three proposed consistency strategies on the algorithm performance (see Section 3.3.2). We used the following three consistency settings:

- KForest none (no consistency enforcement),
- KForest online (10 online consistency enforcement iterations after each addition), and
- KForest full (full consistency enforcement).

We expect trees with higher consistency to produce better results, particularly for lower-accuracy settings where less leaf nodes are evaluated. We are thus interested in whether or not consistency enforcement improves the algorithm performance at all. Additionally, although we expect full consistency to produce the most efficient trees at query-time, it has prohibitively expensive construction costs for an online environment. Since online consistency requires less computation at construction-time, we are particularly interested to see whether or not the reduced query time of full consistency is worth the construction costs in an online environment.

We compared the performance of all these algorithms on the SIFT dataset originally proposed by Jegou et al. (2011), which contains one million 128-dimensional SIFT features extracted from Flickr images. We split this dataset into a training and test set with a random 99/1 split. Thus, the training set consisted of 990,000 elements, and the test set consisted of 10,000 elements.

For each trial, we recorded the average query time (the time required to query the 10 nearest neighbors) across the 10000 test queries. We likewise recorded the average 10-nearest neighbor precision (the overlap between the query result and the actual nearest neighbors) as a measure of the trial's accuracy.

It's possible that the performance of the algorithms may differ at various accuracies: one algorithm may have a comparative advantage for lower accuracies while being worse at higher accuracies. To assess the accuracy-computation trade-off for these algorithms, we tested each algorithm using a variety of hyperparameter settings. If a trial produced results which were strictly worse than any other trial, it was discarded. Note that a trial is considered to be strictly worse than another trial if both its query time is higher and its precision is lower. Thus, we retained the Pareto frontier for each tested algorithm.

For the pre-existing algorithms (Annoy, BallTree, Flann, HNSW Graph, KGraph, and SW Graph), we use the same set of hyperparameter settings as Bernhardsson (2017a). All experiments were performed on an Intel Core i7-4720HQ CPU running at 2.60GHz, with 8 gigabytes of RAM. No multithreading was used for these experiments.

In between KForest trials where only the query parameters differed (not construction parameters), the previously-constructed data structure was reused. For example, if the one KForest trial used planeDistSearch and the following trial used leafGraphSearch (with all other parameters identical), the tree would not be re-built. For KForest, the *searchType*, *exactEps*, and *minLeaves* parameters do not affect the tree construction, and changing these parameters will not force the tree to be rebuilt. On the other hand, changing the *maxLeafSize*, *branchFactor*, *numTrees*, *consistencyType*, *spill*, and *maxSize* parameters all triggered a tree reconstruction.

3.4.1 SIFT Benchmark Results

The results from this experiment are displayed in Figure 3.3. Note that we plot the queries per second (the inverse of the query time). As such, results which are further up and to the right are better.

The most striking result is the performance gap between the graph-based algorithms (HNSW Graph, KGraph, and SW Graph) and the tree-based approaches. HNSW Graph performed the best of all tested algorithms, often producing ten times more queries per second for the same level of accuracy as our KForest algorithm. Unfortunately, these graph-based libraries do not support incrementally adding items in an online environment, which disqualifies them from being used with MFEC. Future work should investigate ways

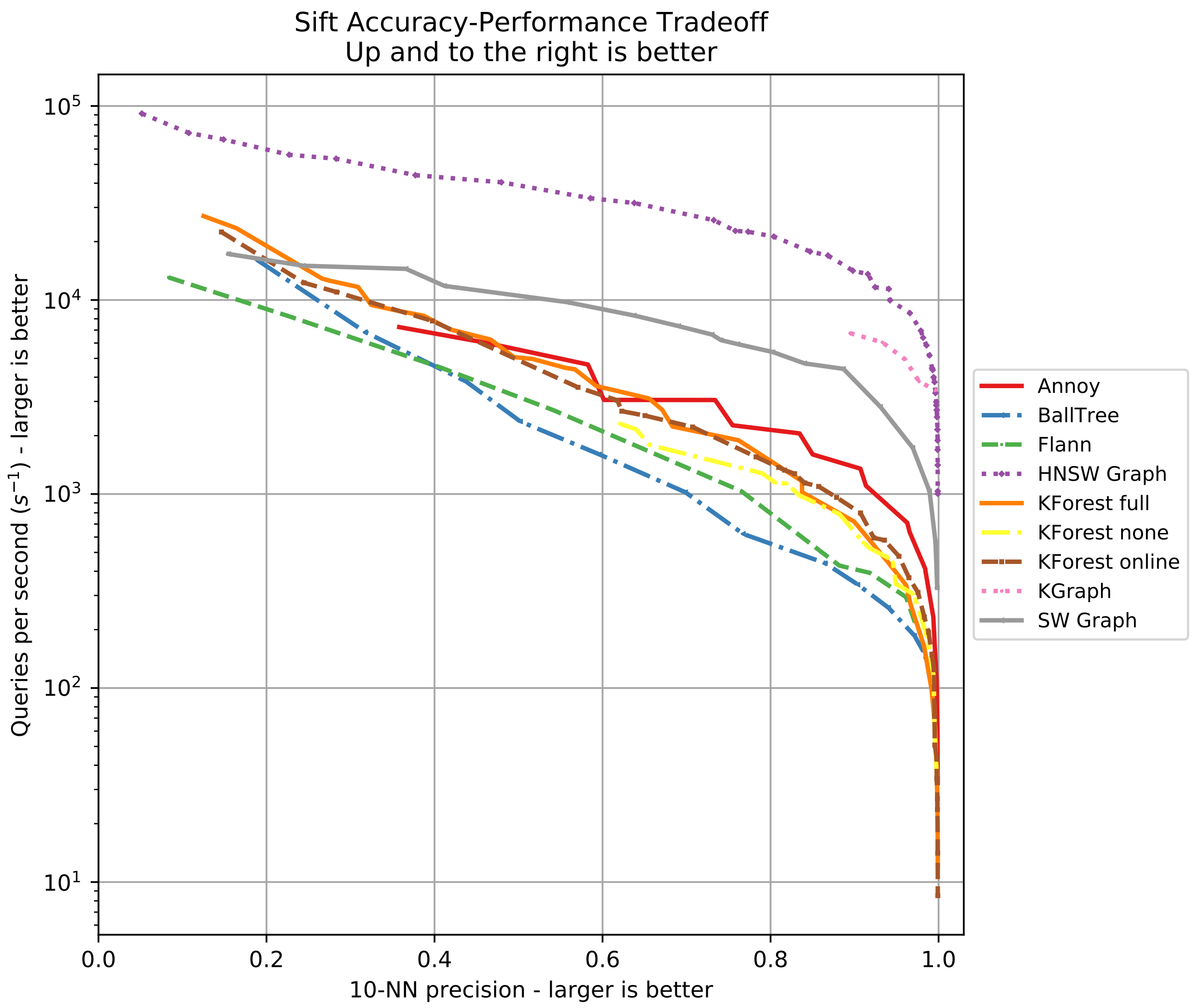


Figure 3.3: Trade-off between accuracy and computation time for various nearest neighbor algorithms.

to incrementally adapt these algorithms to take advantage of these impressive results.

All versions of our KForest algorithm produced better results than the BallTree or Flann algorithms. KForest produced better results than Annoy for low accuracy levels, but worse for high accuracies. We again note that KForest is the only algorithm investigated which supports online additions: in contrast, the BallTree, Flann, and Annoy algorithms all build their data structures in a top-down manner.

We also investigated the differences between the various KForest consistency settings (full, none, and online). As expected, no consistency (KForest none) generally performed worse than both online consistency and full consistency. Thus, we conclude that consistency enforcement does indeed improve the performance of the KForest algorithm.

The discrepancy between full and online consistency enforcement is comparable for the majority of the accuracy levels, with full consistency performing slightly better at extremely low accuracy levels, as predicted. Surprisingly, online consistency enforcement appears to perform slightly better than full consistency for higher accuracy levels. One possible explanation might be that, due to the iterative nature of continuously adding and removing inconsistent datapoints, full consistency enforcement creates trees which are slightly deeper than online enforcement. Another possible explanation might be that full consistency enforcement might cause data points to appear in more leaf nodes, causing wide searches to be less efficient. Regardless, for searches which evaluate many leaf nodes (such as high-accuracy searches), there seems to be diminishing returns in enforcing exact consistency. However, since the observed effect seen here is so small, further experiments are necessary to confirm this.

Overall, we conclude that, for our purposes, the slight query-time performance loss associated with online consistency enforcement is outweighed by the construction-time performance gain. Thus, we decide to use online consistency enforcement for KForest structures in our MFEC implementations throughout the rest of this thesis.

3.4.2 Ms. Pacman Benchmark Results

For the best representative comparisons, we wanted to use KForest settings which were somewhere on or near the Pareto frontier for our MFEC experiments on the Ms. Pacman environment. We thus first needed to gather a base dataset of frames from the Ms. Pacman environment with which we could test various KForest settings. However, we found that random exploration generally produced datasets which were very different from those actually encountered by the agent. We thus chose a set of KForest parameters which corresponded to 99% accuracy on the Sift experiment from the previous section, and used those to generate this dataset. We trained our agent with these parameters for 24 hours, and extracted the frames stored in the first action buffer.

This resulted in a dataset of 554,363 64-dimensional vectors. This was split into a 548,820-element training set, and a 5,543-element test set. The actual nearest neighbors from the training set to the items of the test set were determined using brute force search. Using this dataset, we tested a variety of different KForest parameters using the same experimental setup as the previous section. We tested the following sets of parameters:

accuracy	searchType	branchFactor	maxLeafSize	spill	numTrees	minLeaves
12.69%	GREEDYSEARCH	8	32	-1	1	32
47.58%	PLANEDISTSEARCH	64	48	-1	1	5
76.82%	PROTOTYPEDISTSEARCH	64	48	-1	1	53
79.74%	PROTOTYPEDISTSEARCH	32	128	-1	1	40
85.10%	PROTOTYPEDISTSEARCH	32	128	-1	1	60
89.64%	PROTOTYPEDISTSEARCH	48	64	0.1	1	160
95.17%	PROTOTYPEDISTSEARCH	48	64	0.1	1	320
98.89%	PROTOTYPEDISTSEARCH	32	96	-1	1	480
99.88%	PROTOTYPEDISTSEARCH	24	256	-1	1	640

Table 3.2: Pareto-optimal settings used for MFEC Pacman experiment.

- $searchType \in \{\text{PROTOTYPEDISTSEARCH}, \text{PLANEDISTSEARCH}\}$[16],
- $numTrees \in \{1\}$,
- $branchFactor \in \{24, 32, 48, 64\}$,
- $maxLeafSize \in \{64, 96, 128, 192, 256\}$,
- $spill \in \{-1, 0.1\}$[17],
- $enforceType =$ online, and
- $minLeaves \in \left\{ \left\lfloor \frac{b*2^i}{maxLeafSize} \right\rfloor \mid i \in [0, 14), b \in \{10, 15\} \right\}$[18].

After conducting this grid search, we chose the Pareto-optimal KForest settings with accuracy levels closest to 50%, 75%, 80%, 85%, 90%, 95%, 99%, and 99.9% accuracy. We also used the overall quickest setting, which corresponded to 12.69% accuracy. The exact settings used are listed in Table 3.2.

3.5 MFEC Pacman

As mentioned previously, neither the original MFEC article (Blundell et al., 2016) nor the follow-up article (Pritzel et al., 2017) investigated the impact of the nearest neighbor approximation. Although Blundell et al. (2016) does not mention the specifics of which nearest neighbor algorithm was used, Pritzel et al. (2017) makes explicit mention of k-d trees, which we describe in Section 3.1.4.1. We thus explored various nearest neighbor settings and see how different accuracy levels impact the performance of the MFEC agent.

We used the Ms. Pacman environment (namely, the `MsPacmanNoFrameskip-v4` from the OpenAI Gym Reinforcement Learning library (Brockman et al., 2016)) as a testbed for this experiment. For each run, we ran the algorithm for 16 hours or 100 million frames, whichever came first. After each episode, we recorded statistics such as the number of frames, current walltime, cumulative return,

[16] We found that LEAFGRAPHSEARCH was prohibitively expensive for this exhaustive search, consistently producing suboptimal results.

[17] A *spill* setting of -1 corresponds to no spill.

[18] The theoretical maximum number of elements to be compared against is $b * 2^i$.

amount of time spent acting during the episode (primarily query lookups) and amount of time spent wrapping up the episode (primarily adding elements to the respective KForests).

The MFEC agents used experience buffers which held up to 1,000,000 experiences for each possible action. The agents used a transformed version of the screen pixels as input: the screen was first converted to greyscale and then rescaled to a 84-by-84 image. These rescaled screen images were then transformed via a sparse Gaussian random projection, which produced a 64-element vector. These 64-dimensional vectors served as the state representations which were ultimately stored in the experience buffers.

Although not mentioned in the paper, in private correspondence it was discovered that Blundell et al. (2016) and Pritzel et al. (2017) used max-pooling between subsequent frames in addition to using a “stack" of the previous 4 frames. Our experiments did not use these specific preprocessing steps.

The KForests underlying the experience buffers used online consistency enforcement with 10 iterations. The agents used the $K = 11$ nearest neighbors for approximating the values of unseen frames. They used an epsilon-greedy exploration strategy ($\epsilon = 0.005$), and the agents repeated their actions for 4 frames between action decisions. We chose to test the Pareto-optimal KForest settings with accuracy levels closest to 50%, 75%, 80%, 85%, 90%, 95%, 99%, and 99.9% accuracy, as described in Section 3.4.2.

We conducted three runs for each of these settings, differentiated by the seeds used for epsilon-greedy exploration, random projections, and KForest splitting behavior. We compared the results of our experiments using two methods for time keeping: walltime and frame count. The walltime results were based on the actual time that elapsed between successive epochs, whereas the frame count only considered the number of frames processed by the agent.

The frame count metric gives an estimate of the agent's sample efficiency, which is especially relevant in environments where environment interaction has a higher latency, as is the case for real-world environments or complex simulated environments. Given the Atari environment's relative simplicity, the time spent on agent decision-making is almost always greater than that spent on environment simulation.

The walltime metric is perhaps the more practical metric for researchers interested primarily in the Atari environment, as it gives an insight on how long a researcher must run a simulation in order to obtain a result. However, this metric is heavily affected by the specific hardware of the host running the experiments, while also producing noisier estimates due to noise from other software running on the system. We note that a large majority of the literature only gives results in terms of frame count (likely due to these concerns). Nonetheless, we decided to include walltime results here, due to our focus on practically comparing the computational efficiency between these methods.

As mentioned previously, all approximate nearest neighbor algorithms make a trade-off between computational efficiency and accuracy. We would expect lower-accuracy nearest neighbor searches to require less computation time to chose actions, at the cost of producing less-accurate Q estimates. Thus, we expect that agents with high-accuracy KNN settings would perform better on the frame count metric than those with lower accuracy settings.

It is worth reiterating that, for MFEC agents, the KNN search is only performed when looking up the Q-value of a state which is not in the agent's memory (either due to it being unvisited, or it having been evicted to make room for other states). If the exact state is already present, then the maximum observed return from that state is given as a result (this is made efficient by means of an exact hash table). Thus, the noise induced by inaccurate KNN settings only effects the Q-function estimates of unseen states.

3.5.1 MFEC Pacman Results

Figure 3.4 displays the results from this experiment. To normalize across trials and to demonstrate the speed/accuracy trade-off, we show results for all frames, for just the first 10 million frames, and for the first 16 hours. Each trial is depicted as a partially-transparent line marking the smoothened return across episodes, as smoothed by the `gam` function from the R package `mgcv`. The bold lines depict the trails which obtained the median maximum average return of each type. That is, if we sort the three trials according to the maximum value of its smoothed return function, the one which produced the middle value is the one shown in bold.

As expected, agents with lower KNN accuracies encountered more frames than those with higher accuracies. The fastest agents (those with settings which produced a 12.69% accuracy on the SIFT benchmark) completed all 100 million frames in less time than it took the slowest agents (which used brute force search) to encounter 8 million frames. In this time, the 12.69% agents completed 20,000 epochs, compared to the 1,500 epochs of the brute force agents.

It is very surprising, however, that agents with high-accuracy ANN settings produced consistently worse results than those with low-accuracy settings across both metrics. While this may be unsurprising for the walltime metric (since low-accuracy agents effectively obtain more experience in the same amount of walltime), this was true even on the framecount metric, which we expected to be favorable to high-accuracy agents (given that they perform more work per frame). The median highest-accuracy (brute force) agent averaged scores around 1700 after 5 million frames, whereas the median lowest-accuracy (12.69%) agent averaged nearly 3100 at the same framecount. Furthermore, the lowest three accuracy settings produced the best median agents after 10 million frames.

One possible explanation for this counter-intuitive result might be that MFEC agents benefit from noisy Q-function imputation. As noted previously, visited states are always perfectly recalled, and the KNN search is only used for novel states. One would expect a noisy KNN search to produce results which are too high for some novel states and too low for others. Novel states whose predicted Q-values are too high are thus more likely to be explored. This has similarities to technique of "optimistic initialization" which encourages exploration by intentionally overvaluing unseen states (Singh and Sutton, 1996). Thus, it is possible that the negative impact of inaccurate imputations for unvisited states is ameliorated by the benefit of additional (somewhat) guided exploration.

Future research is necessary to fully explain this phenomenon. In particular, it would be possible to directly test if the "optimistic initialization"-esque behavior can explain the performance gap by using a Q-function estimator with

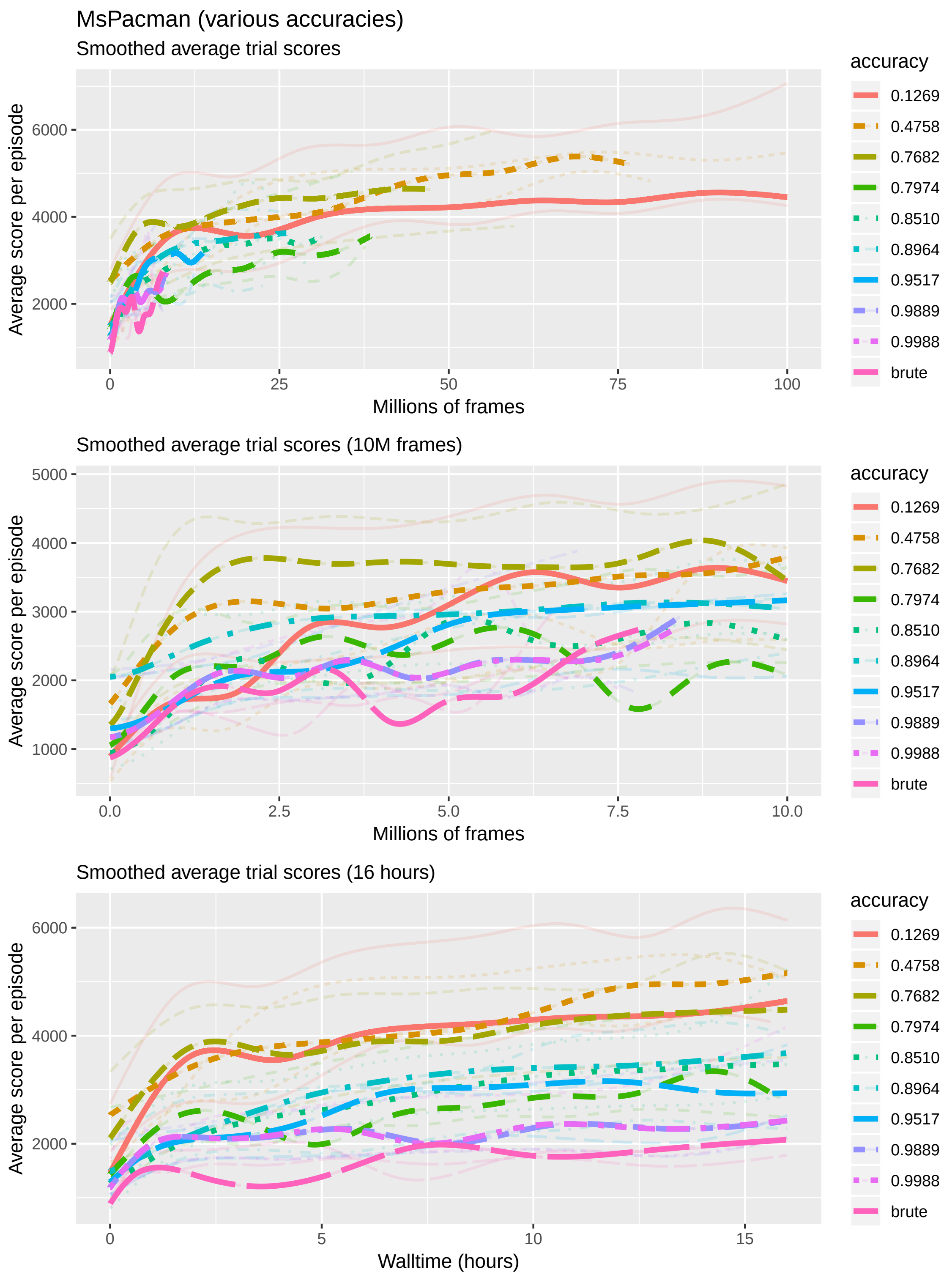


Figure 3.4: Ms. Pacman results for various accuracies.

an artificial reward for visiting novel states:

$$\mathrm{Q'(s,a)} = \begin{cases} \mathrm{Q(s,a)} & \text{if } \mathrm{(s,a)} \text{ visited} \\ \mathrm{Q(s,a)} + \alpha & \text{otherwise} \end{cases}$$

It should also be investigated how the performance of MFEC agents is impacted by the addition of artificial noise to accurate KNN queries, e.g. by adding Gaussian noise to the Q-function estimate.

3.6 Summary

In this chapter, we described the state-of-the-art in approximate nearest neighbor data structures. We then described our own online data structure, which we dubbed KForest, and showed that its performance on the SIFT benchmark is competitive with commonly-used tree-based batch methods which do not support additions or deletions. However, we think it is important to note that the graph-based approaches had significantly better performance on the SIFT benchmark, and that further research should be focused on an implementation of these algorithms which support online additions and deletions.

We then tested the performance of the MFEC agent on the Ms. Pacman environment across various accuracy settings. Interestingly, we found that the agents with the most accurate settings (ranging between 98.9% to 100%) performed worst on our tests. Even when looking at performance with respect to frame count (discarding the effect of less frames encountered due to increased processing time), these agents still performed among the worst for the first 5 million frames.

We found that the KForest settings which resulted in 76.82% nearest neighbor accuracy produced the best results on average for this experiment. Thus, we decide to use these KForest settings throughout all following experiments for the remainder of this thesis.

Chapter 4

Local Regression

One important task in machine learning is regression, where an algorithm predicts a real-valued label given an input. Regression datasets consist of P labeled examples, $D = \{(\boldsymbol{x}_i, y_i) \mid i \in [0, P)\}$, such that each example is a d-dimensional real-valued vector, $\boldsymbol{x}_i \in \mathbb{R}^d$, and each target is a real value, $y_i \in \mathbb{R}$.

At test-time, a regression algorithm is given an unlabeled example as input, and predicts an output which matches the rule underlying the dataset as closely as possible. We denote this prediction by the function $\hat{y}(\boldsymbol{x}_i) : \mathbb{R}^d \to \mathbb{R}$.

Many regression algorithms are trained to adapt a *global* model to predict the rule which produced the dataset. That is, a single set of parameters are used no matter what input is given. One example of an algorithm with such a global model is linear regression, which has the prediction function:

$$\hat{y}(\boldsymbol{x}) = a \cdot \boldsymbol{x} + b, \tag{4.1}$$

where $a \in \mathbb{R}^d$ and $b \in \mathbb{R}$ are trainable parameters which are used globally.

In contrast, in this thesis we explore a class of regression algorithms which use *local* models. The defining characteristic of these algorithms is that they use a different model for each query. This is often done by assigning less importance to distant examples in the dataset than to close examples. This relies on the assumption that examples which are close to the query are more contextually relevant than far away ones.

A simple version of a local regression algorithm is k-nearest neighbors (kNN) regression. In this scheme, the k training examples closest to the query are selected and all other examples are given a weight of 0. The prediction function for kNN regression is thus:

$$\hat{y}_{\boldsymbol{q}}(\boldsymbol{x}) = \frac{1}{k} \sum_{\boldsymbol{x}_i \in \mathrm{NBR}_k(\boldsymbol{q})} y_i, \tag{4.2}$$

where $\mathrm{NBR}_k(\boldsymbol{q}) : \mathbb{R}^d \to \mathcal{P}(\mathbb{R}^d)$ is the function which returns the set of k closest examples (the neighbors) to the query point, $\boldsymbol{q}$. We visually showcase the difference between a global linear and various local models in Figure 4.1.

For local regression algorithms, we use the notation $\hat{y}_{\boldsymbol{q}}(\boldsymbol{x})$ to denote the result of models trained around the query point $\boldsymbol{q}$ evaluated at the point $\boldsymbol{x}$. For these algorithms, the *local regression prediction* (the predicted result for a given

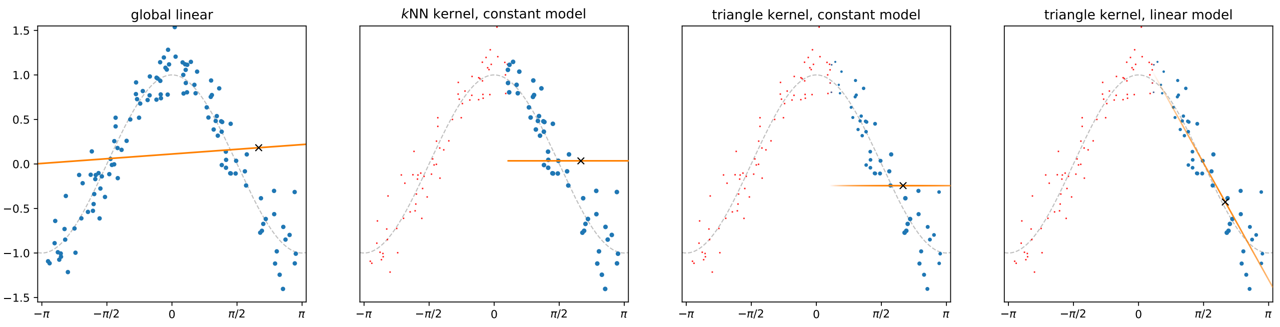


Figure 4.1: Global linear model (left) vs various local models (rightmost three) for estimating a noisy cosine function (noise sampled from the normal distribution with mean 0 and standard deviation 0.25). Filled blue circles mark examples that were used by the model, with their size indicating the relative strength of their contribution. The orange lines indicate the result of the fitted local model, and the black "X" indicates the local model's prediction for the query point, which was $\mathbf{q} = \frac{2\pi}{3}$. The local models used $k = 50$. The global linear model estimates a value of 0.18, the kNN kernel estimates 0.035, the triangle kernel estimates -0.24, and the triangle kernel with local linear model estimates -0.43. The target value was $\cos(\frac{2\pi}{3}) = -0.5$.

input) is given by:

$$\hat{y}(\boldsymbol{x}) = \hat{y}_{\boldsymbol{x}}(\boldsymbol{x}). \tag{4.3}$$

Making a prediction using these algorithms requires constructing a new model for each query. As a result, simple models are typically used as a local model. We will discuss this in more detail in the following section.

Note that the result of Equation 4.2 is dependent only on $\boldsymbol{q}$, not $\boldsymbol{x}$. From this example, it may not be totally clear why we distinguish between $\boldsymbol{q}$ and $\boldsymbol{x}$. To motivate this notation, we first give a formal definition for local regression.

4.1 Weighted Square Error

Most machine learning problems are solved by posing the problem as some differentiable cost function, which is then minimized. For example, regression algorithms often minimize the mean squared error between the predicted and actual targets:

$$\text{MSE} = \frac{1}{P}\sum_{i=0}^{P-1}(y_i - \hat{y}(\boldsymbol{x}_i))^2. \tag{4.4}$$

In this thesis we consider *local* models which minimize a *weighted* square error:

$$\text{WSE}_{\boldsymbol{q}} = \sum_{i=0}^{P-1} w(\boldsymbol{x}_i, \boldsymbol{q})(y_i - \hat{y}_{\boldsymbol{q}}(\boldsymbol{x}_i))^2, \tag{4.5}$$

where $\boldsymbol{q} \in \mathbb{R}^d$ is the *query point* around which the local model is centered, and $w(\boldsymbol{x}_i, \boldsymbol{q}) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is the *weighting function* which determines the relevance of example i to the query point (Loader, 1999). The weighting function monotonically decreases (does not increase, but might remain constant) as the distance between $\boldsymbol{q}$ and $\boldsymbol{x}_i$ increases. As a result, errors on distant examples are penalized less than those local to $\boldsymbol{q}$.

4.2 Weighting Function

The choice of weighting function plays a large role in the performance of local regressors. Indeed, we can see that MSE is actually just a special case of WSE where the weighting function is uniform ($w(\boldsymbol{x}_i, \boldsymbol{q}) = \frac{1}{P}$).

An important class of weighting functions are defined in terms of a monotonic *kernel* $K : \mathbb{R}^+ \to \mathbb{R}^+$.[1] Formally, these kernel weighting functions are defined as:

$$w_{h(\boldsymbol{q})}(\boldsymbol{x}_i, \boldsymbol{q}) = K\left(\frac{\|\boldsymbol{x}_i - \boldsymbol{q}\|}{h(\boldsymbol{q})}\right), \tag{4.6}$$

where $h(\boldsymbol{q}) : \mathbb{R}^d \to \mathbb{R}$ is the *bandwidth* hyperparameter which determines the "width" of the underlying kernel. Although any measure of dissimilarity can be used in the numerator of Equation 4.6, the Euclidean norm is most commonly used.

In general, $h(\boldsymbol{q})$ can either be a fixed constant ($h(\boldsymbol{q}) = h$), or can vary based on the query. Smaller bandwidth values produce spikier predictions (higher variance), whereas larger values risk simply reproducing the mean (higher bias).

One possible bandwidth function, known as the nearest neighbor bandwidth, is given by:

$$h(\boldsymbol{q}) = \underset{\boldsymbol{x}_i \in \mathrm{NBR}_k(\boldsymbol{q})}{\operatorname{argmax}} \|\boldsymbol{x}_i - \boldsymbol{q}\|. \tag{4.7}$$

This sets the bandwidth of the kernel to be equal to the distance from the query point to the kth closest example.

Multiple types of kernels can be used. Some popular choices for kernels are given in Table 4.1. However, the bandwidth generally has a larger impact on the quality of the result than the exact shape of the kernel (Tibshirani, 2014). A visual comparison between different kernels and bandwidths can be found in Figure 4.2.

4.3 Local Models

The WSE cost function is differentiable, so it is theoretically possible to use this cost function to train arbitrarily complex models via gradient descent (e.g. multilayer neural networks). However, this is prohibitively computationally expensive, as it would require an intensive training phase for each query. As a result, simple models with analytic solutions are typically used, such as a constant estimator (Section 4.3.1) or a linear model (Section 4.3.2).

In particular, two popular choices for local models are the constant local model (known as *Kernel Regression*) and the linear local model (known as *Locally Linear Regression*).

[1]"Kernel" is an overloaded term in machine learning. For example, in Support Vector Machines, "kernels" define an implicit inner product in some (potentially unknown) Hilbert space. Throughout this thesis, we use the term "kernel" only to describe monotonic functions from one positive real value to another.

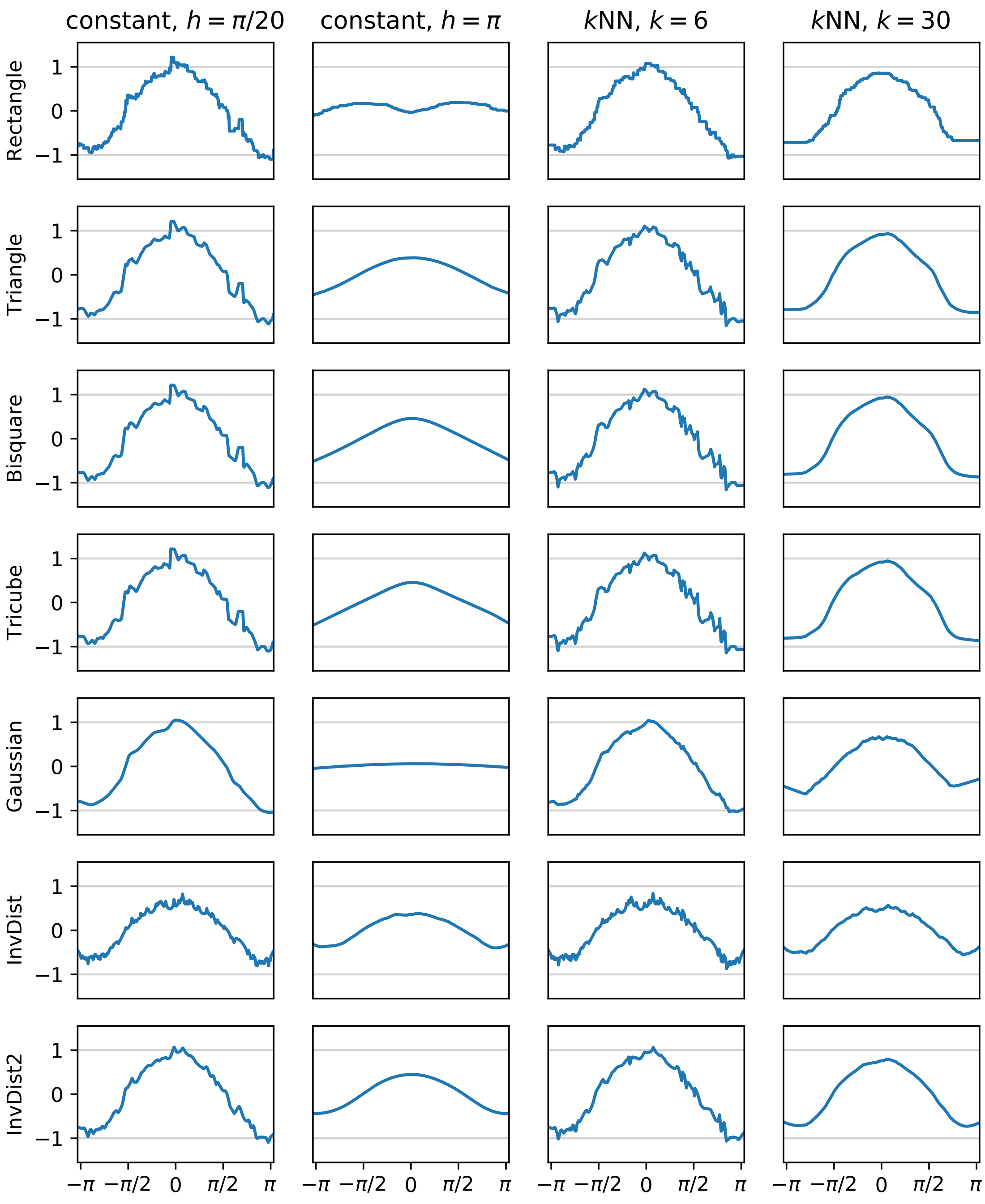


Figure 4.2: Comparison of local regression predictions between kernel regressors with various kernel weighting functions and bandwidths.

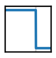
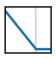
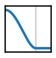
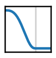
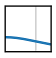
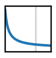
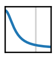
Rectangle		$K(x) = \begin{cases} 1 & \text{if } x \leq 1 \\ 0 & \text{otherwise} \end{cases}$
Triangle		$K(x) = \begin{cases} 1 - \lvert x \rvert & \text{if } x \leq 1 \\ 0 & \text{otherwise} \end{cases}$
Bisquare		$K(x) = \begin{cases} (1 - x^2)^2 & \text{if } x \leq 1 \\ 0 & \text{otherwise} \end{cases}$
Tricube		$K(x) = \begin{cases} (1 - \lvert x \rvert^3)^3 & \text{if } x \leq 1 \\ 0 & \text{otherwise} \end{cases}$
Gaussian		$K(x) = \frac{\exp(-x^2/2)}{2\sqrt{\pi}}$
Inverse Dist		$K(x) = \epsilon/(x + \epsilon)$
Inverse SqDist		$K(x) = \epsilon/(x^2 + \epsilon)$

Table 4.1: Various kernels which can be used to produce kernel weighting functions. The light grey line is placed at $x = 1$.

4.3.1 Kernel Regression

Kernel regression (also known as kernel smoothing, weighted mean, or Nadaraya-Watson regression) is perhaps the simplest possible model for local regression. For each query point, the local model always predicts a constant value $b_{\boldsymbol{q}}$, regardless of the value of $\boldsymbol{x}$. That is, kernel regression uses the local model $\hat{y}_{\boldsymbol{q}}(\boldsymbol{x}) = b_{\boldsymbol{q}}$.

One nice property of this model is that when this local model is plugged into the WSE cost function, it is possible to derive a simple solution which minimizes the local error:

$$\begin{aligned}
\mathrm{WSE}_{\boldsymbol{q}} &= \sum_{n=0}^{P-1} w(\boldsymbol{x}_i, \boldsymbol{q})(y_i - \hat{y}_{\boldsymbol{q}}(\boldsymbol{x}_i))^2 \\
&= \sum_{n=0}^{P-1} w(\boldsymbol{x}_i, \boldsymbol{q})(y_i - b_{\boldsymbol{q}})^2 \\
\implies \frac{\partial}{\partial b_{\boldsymbol{q}}} \mathrm{WSE}_{\boldsymbol{q}} &= -2 \sum_{n=0}^{P-1} w(\boldsymbol{x}_i, \boldsymbol{q})(y_i - b_{\boldsymbol{q}}) \\
\implies 0 &= -2 \sum_{n=0}^{P-1} w(\boldsymbol{x}_i, \boldsymbol{q})(y_i - b_{\boldsymbol{q}}) \\
\implies 0 &= \sum_{n=0}^{P-1} w(\boldsymbol{x}_i, \boldsymbol{q})y_i - \sum_{n=0}^{P-1} w(\boldsymbol{x}_i, \boldsymbol{q})b_{\boldsymbol{q}} \\
\implies \sum_{n=0}^{P-1} w(\boldsymbol{x}_i, \boldsymbol{q})b_{\boldsymbol{q}} &= \sum_{n=0}^{P-1} w(\boldsymbol{x}_i, \boldsymbol{q})y_i \\
\implies b_{\boldsymbol{q}} \sum_{n=0}^{P-1} w(\boldsymbol{x}_i, \boldsymbol{q}) &= \sum_{n=0}^{P-1} w(\boldsymbol{x}_i, \boldsymbol{q})y_i \\
\implies b_{\boldsymbol{q}} &= \frac{\sum_{n=0}^{P-1} w(\boldsymbol{x}_i, \boldsymbol{q})y_i}{\sum_{n=0}^{P-1} w(\boldsymbol{x}_i, \boldsymbol{q})}.
\end{aligned}$$

Thus, for kernel regression the optimal model is $\hat{y}_{\boldsymbol{q}}(\boldsymbol{x}) = \frac{\sum w(\boldsymbol{x}_i, \boldsymbol{q})y_i}{\sum w(\boldsymbol{x}_i, \boldsymbol{q})}$.

If we have a constant weight function ($w(\boldsymbol{x}_i, \boldsymbol{q}) = 1$), then this estimate simply becomes the mean of the dataset:

$$\hat{y}_{\boldsymbol{q}}(\boldsymbol{x}) = \frac{\sum 1 \cdot y_i}{\sum 1} = \frac{\sum y_i}{N}. \tag{4.8}$$

On the opposite end of the spectrum, consider kernel regression with a kernel weighting function using a rectangular kernel and a nearest neighbor bandwidth. Aside from the case where there are ties on the kth nearest example, the prediction becomes:

$$\hat{y}_{\boldsymbol{q}}(\boldsymbol{x}) = \frac{\sum w(\boldsymbol{x}_i, \boldsymbol{q})y_i}{\sum w(\boldsymbol{x}_i, \boldsymbol{q})} = \frac{\sum\limits_{\boldsymbol{x}_i \in \mathrm{NBR}_k(\boldsymbol{q})} y_i}{\sum\limits_{\boldsymbol{x}_i \in \mathrm{NBR}_k(\boldsymbol{q})} 1} = \frac{1}{k} \sum_{\boldsymbol{x}_i \in \mathrm{NBR}_k(\boldsymbol{q})} y_i. \tag{4.9}$$

This is exactly the predictor for kNN regression (Equation 4.2).

One useful property of this model is that it is an $\mathcal{O}(1)$ operation to compute the leave-one-out prediction. That is, given the weighted sums (which are necessary for normal predictions), it takes only a constant number of additional operations to determine what the prediction would have been had that example not been in the dataset. The leave-one-out prediction is simply given by:

$$\widetilde{y}_{j,\boldsymbol{q}}(\boldsymbol{x}) = \frac{\sum w(\boldsymbol{x}_i, \boldsymbol{q})y_i - w(\boldsymbol{x}_j, \boldsymbol{q})y_j}{\sum w(\boldsymbol{x}_i, \boldsymbol{q}) - w(\boldsymbol{x}_j, \boldsymbol{q})}. \tag{4.10}$$

We make extensive use of this property in Chapter 5 to adapt kernel regression to online environments with drift.

One important consideration for kernel regression is that the fringes of the datasets become especially prone to bias. This can be seen in Figure 4.2, which uses kernel regression with various weight configurations. Many of the curves in Figure 4.2 have a premature “upward” bend at the extremities, even though all examples were constrained to the range $[-\pi, \pi]$. This is most evident for large bandwidths (see the last three rows of the last column), where the local contribution of the nearest point fades away in favor of the majority near the mean.

4.3.2 Locally Weighted Regression

In contrast to kernel regression which uses a constant local model, locally weighted regression (also known as locally linear regression or LOESS) uses a linear model for each query point. Unlike kernel regression, the locally weighted regression is affected by $\boldsymbol{x}$: $\hat{y}_{\boldsymbol{q}}(\boldsymbol{x}) = a_{\boldsymbol{q}} \cdot \boldsymbol{x} + b_{\boldsymbol{q}}$.

As is the case with ordinary linear regression, it is possible to use the same machinery to fit higher-order polynomials through the use of a *mapping function*:

$$\phi_d(\boldsymbol{x}_i) = (1, x, x^2, \cdots, x^d)^T. \tag{4.11}$$

Using this notation, the model for locally weighted regression thus becomes:

$$\hat{y}_{\boldsymbol{q}}(\boldsymbol{x}) = a_{\boldsymbol{q}} \cdot \phi_d(\boldsymbol{x}). \tag{4.12}$$

From this formulation, we can view kernel regression as a special case of locally weighted regression, with 0-degree polynomials.

Luckily, when this model is plugged into WSE, the minima can be computed algebrically by (Loader, 1999):

$$a_{\boldsymbol{q}} = (X^T W X)^{-1} X^T W \boldsymbol{y},$$

where W is a diagonal matrix with elements equal to $w(\boldsymbol{x}_i, \boldsymbol{q})$, and X is the design matrix with each example in one row.

4.4 Alternate Criterion

We briefly note that it is possible to extend the general idea of local weighting to alternate domains by changing the objective criterion. For example, the squared error in Equation 4.5 could be substituted for cross-entropy loss for classification tasks. Similarly, the regression can be made robust to outliers by similarly changing the objective criterion (Loader, 1999).

4.5 Computational Efficiency

Many popular kernels have a finite *support*, such that for all $x \geq 1$, $K(x) = 0$. When such kernels are used, all examples more than $h(\boldsymbol{q})$ away from $\boldsymbol{q}$ can be ignored without affecting the result. Other kernels, such as the Guassian kernel, have an infinite support, and thus technically require evaluating every example for each query. Thus, when using finite-support kernels, nearest neighbord data structures can be used to greatly reduce the computational expense of each query.

4.6 MFEC Pacman

We wanted to investigate how the choice of kernel and local model affects the performance of the MFEC agent.

Similar to our experiment in Section 3.5, we conducted multiple trials using the Ms. Pacman environment from the OpenAI Gym reinforcement learning library (Brockman et al., 2016). For each trial, we let an agent act in the environment for 16 hours or 100 million frames, whichever came first. At the end of each episode, we recorded statistics about that episode.

Our MFEC agent used experience buffers which held up to 1,000,000 experiences for each possible action. The agent used a transformed version of the screen pixels as input such that each observation was represented as a 64-dimensional vector, as explained in Section 3.5.

The KForests underlying the experience buffers used the settings which we previously found to have 76.82% accuracy on Ms. Pacman images (see Table 3.2). The agents used the $K = 11$ nearest neighbors for approximating the values of unseen frames, used an epsilon-greedy exploration strategy ($\epsilon = 0.005$), and repeated their actions for 4 frames between action decisions.

We note that it is possible that the choice of local model might have an interaction effect with the choice of type of kernel used. Thus, we tested all possible combinations of kernels (constant, inverse squared distance, triangle, and tricube) and local models (kernel and linear). For each setting, we performed three trials, for a total of $4 \times 2 \times 3$ trials in total.

4.6.1 MFEC Ms. Pacman Results

Figure 4.3 displays the results from this experiment. We note that, although the linear local model requires slightly more computational resources than kernel local models, this difference is negligible when compared to the costs of performing the kNN lookup and environment simulation. Thus, we only compare results based on frame counts, and not walltime. We show results for all frames (the 16 hour regime) as well as for just the first 10 million frames (the small-data regime).

As in Section 3.5.1, each trial is depicted as a partially-transparent line marking the smoothened return across episodes, as smoothed by the `gam` function from the R package `mgcv`. For each agent, we take the maximum value of this smoothed return, and then sort these maximum values by the agent type. The bold lines depict the trials which obtained the median of these maximum returns for each agent type.

In the 16 hour regime, the kernel local models using the inverse square distance and tricube kernels performed better than the constant or triangle kernels by the end of the trial. The kernel local model and linear local model generally performed similarly for the same kernel type, except for the notable exception of the inverse square distance kernel. For this kernel, the performance

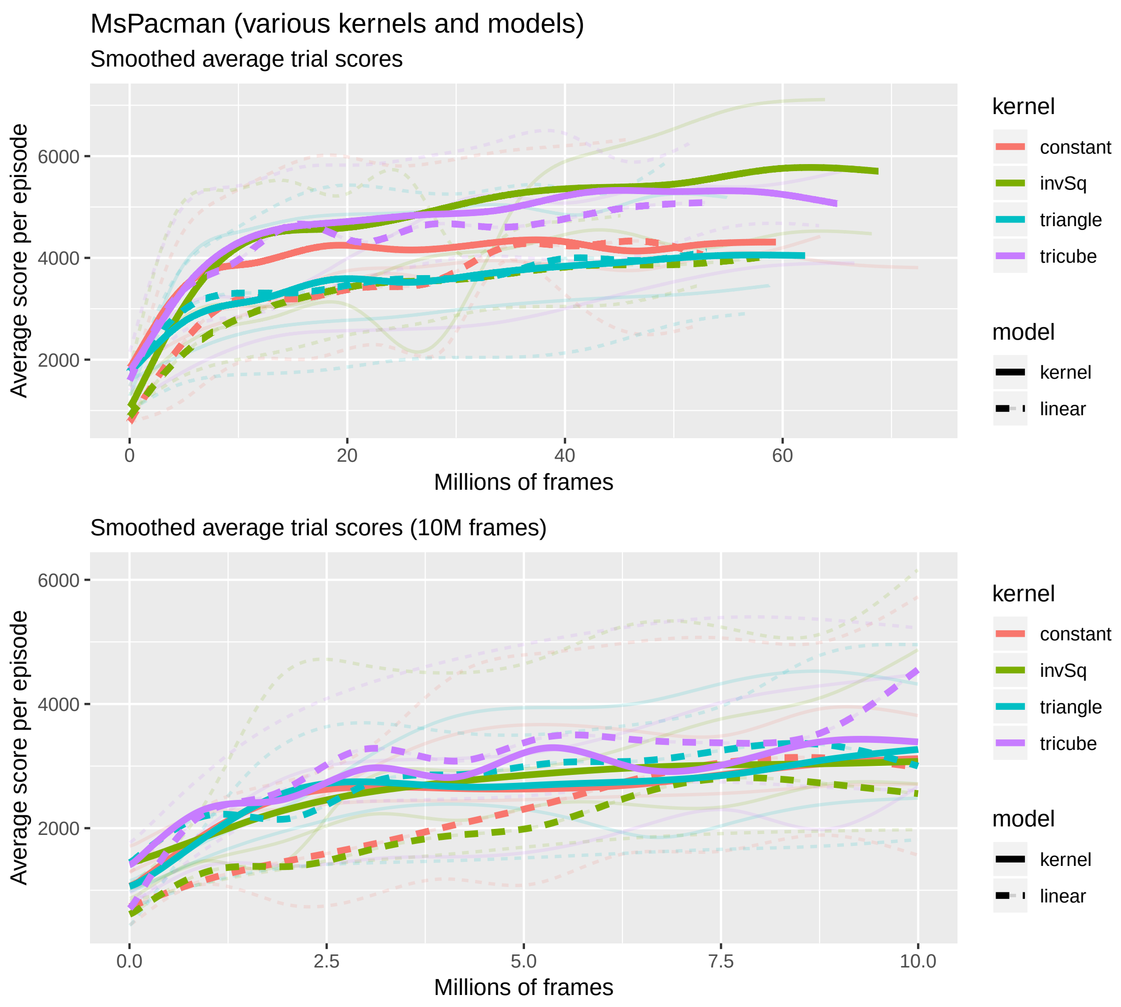


Figure 4.3: Ms. Pacman results for various kernel and local model settings.

of the kernel local model performed best overall, whereas the linear local model with the same kernel performed worst.

In the small-data regime, the constant and inverse square distance kernels with a linear local model produced the worst median scores for the first 5 million frames. However, there was a great deal of variance in this measure: the highest-scoring individual runs after 10 million frames were from the constant and inverse square distance kernels with linear local models. The median tricube agents performed best on the small-data regime, with the linear agent performing slightly better than the kernel agent at 10 million frames.

Overall, the performance of the median agents does give some indication that the tricube and inverse square distance kernels with the kernel local model are superior in the 16 hour regime. While the local model type does not seem to have a large impact in the 16 hour regime, for the small-data regime there seems to be a considerable dip in performance when using the linear model with the constant and inverse square distance kernels. However, the large amount of variance observed in the individual trials would seem to indicate that the choice of kernel and local model are not as important as other hyperparameter settings.

4.7 Summary

In this chapter, we described how simple k-nearest neighbors regression is just one of a family of regression methods known as local regression. Specifically, we described that these methods can be categorized based on two orthogonal criteria: the weighting function (kernel) and the local model. The kernel determines the relative weight of each neighbor based on its distance to the query point. The local model is the regressor which is ultimately fitted to the data according to the local weighted square error (Equation 4.5).

We then tested the performance of the MFEC agent on the Ms. Pacman environment across various kernel and local model settings. In this experiment, the usage of a linear local model did not improve performance above that of a constant local model (and actually hurt performance in one case). Additionally, although it did not make much of an impact in the 10-million frame regime, in the 16-hour regime the tricube and invSq kernels performed best on average.

Chapter 5

Drift Compensation

Many machine learning algorithms make the assumption that the target dataset is fixed and does not vary over time. These algorithms typically have a single training phase on a fixed dataset, after which the algorithm only gives static predictions. However, not all problems match this assumption of a fixed dataset. The field of *online learning* focuses on the more general problem of learning on streams of data: a dataset which can change during the course of learning. In order to accommodate to new data as it is gathered, online learning algorithms typically have a periodic or continuous training regimen. These algorithms must be able to produce predictions in an online fashion, interleaved with receiving new data. Indeed, human memory has been shown to accommodate for new information by actively removing old memories (Hardt et al., 2013).

Of particular importance to this thesis, we note that reinforcement learning is a form of online learning. If a fixed policy were given for some environment[1], then the problem of determining the expected return from following that policy from a given set of states (known as *policy evaluation*) is a static problem, since the actual policy value is fixed and thus serves as a fixed target. Thus, offline learning algorithms are sufficient for policy evaluation.

However, the full reinforcement learning problem differs from policy evaluation in two key ways. Firstly, reinforcement learning requires the agent to determine an expected return of all states the agent will encounter, which is potentially unknown a priori. Encountering a new state is akin to adding new elements to the target dataset. Secondly, and perhaps more importantly, if the agent's policy is allowed to change over time then the actual policy value will change as a result. Agents which use DQN or MFEC (or, indeed, any Q-learning algorithm) adjust their policies online, as a consequence of updating their beliefs about the consequences of their actions as they gain additional experience in their environments.

5.1 Drift

The online learning paradigm introduces a phenomenon not possible with fixed datasets: the dataset itself may change in some way over time. This phenomenon, known as *drift*, can occur in many different ways.

[1]We assume a static environment, whose transition probabilities are fixed.

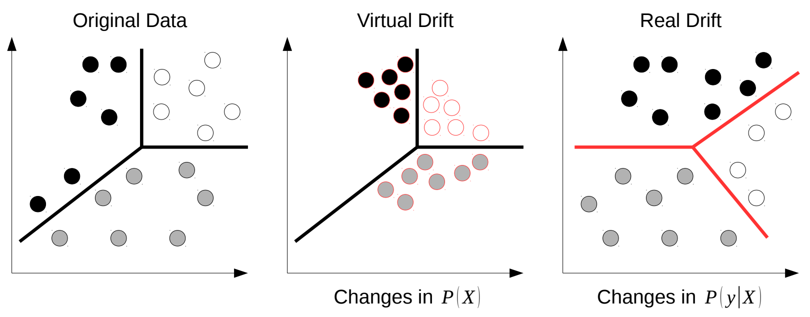


Figure 5.1: Illustration of virtual (middle) and real (right) drift types.

A dataset may change smoothly or abruptly. This change may occur only once, periodically, or continuously.

However, one of the most important distinctions of types of drift are known as *virtual drift* and *real drift*.[2] Virtual drift occurs when the distribution of the inputs change, but not the assigned labels. That is, the probability distribution $P(X)$ changes. Real drift occurs when the labels themselves change, but perhaps not the distribution of inputs. That is, the probability distribution $P(y_i \mid X)$ changes. Figure 5.1 gives a visual depiction of these two drift types.

It is important to note that in reinforcement learning both of these drift types commonly occur. Virtual drift corresponds to encountering a different distribution of observations as the agent's policy changes as a result of interaction with the environment. Virtual drift can also occur as a byproduct of exploration, which explores novel parts of the state space. Real drift occurs when the agent gains more knowledge about the environment and updates its Q-value representation as a result. This can occur either by learning a new strategy or through propagating temporal differences through the Q-function approximator. Note that, due to the immediate one-shot learning property of the MFEC's kNN regressor, this latter case is not as applicable (although the former definitely is).

Many different strategies for compensating for this drift have been proposed. There exist two general categories of algorithms for drift compensation: *active* algorithms which attempt to identify discrete time points where change occurs, and *passive* algorithms which attempt to adapt continuously, without explicit demarcations. The approach we propose is a passive algorithm, identifying only when our current model does not match our expectations.

We propose another extension to the locally weighted regression paradigm which accommodates for online learning with drift. It does this via immediate leave-one-out comparisons to determine the error a prediction would make had that item not been included. The general idea is that if the algorithm produces consistently better results with the item left out, it is beneficial to remove that item.

[2]The term real drift simply describes a specific kind of drift, and does not imply that other kinds of drift are not real or otherwise fake.

5.2 Leave-One-Out Drift Compensation

Many drift compensation schemes attempt to compensate for real drift by using a sliding window to determine a time threshold beyond which data is considered outdated (Losing et al., 2016). In most approaches using this sliding window technique, samples older than this threshold are simply forgotten and remain unused in training future models. This technique makes the implicit assumption that changes in the labels in a dataset occur uniformly across all inputs. However, this assumption does not necessarily hold true in the reinforcement learning domain. In particular, in reinforcement learning, it is sometimes beneficial to remove more recent memories while retaining older ones.

To illustrate this, imagine an agent attempting to learn to play Frostbite from the Atari Learning Environment (for details, see Appendix A.4). During an early episode, the agent immediately jumps downwards and lands in the water, resulting in a disadvantageous outcome at the beginning of the episode. This outcome is inevitable whenever the agent takes this action, and thus the agent should not forget it. Much later in training, the agent discovers that it is possible to enter the igloo after collecting 16 ice blocks, giving a large reward. This new information causes the agent's evaluation of some previously-encountered states to be outdated (namely, the value of states from which it is still possible to reach the igloo should be increased). However, the agent's knowledge about the consequences of jumping into the water is still relevant and should not be forgotten, despite those experiences being older.

We thus would like a drift compensation method which operates on a per-item basis, rather than a sliding window based on the age of the experience. We propose the use of a simple signal of drift: the leave-one-out prediction error. The leave-one-out prediction for a given item simply gives what the prediction would have been had that item never been added to the dataset, thus providing a measure of the contribution of an individual element to the overall prediction of an item, If real drift has occurred relative to a new example, we would expect that the contribution from outdated items would consistently cause a higher error, such that the error would be reduced if that element were removed.

The algorithm we propose only enforces consistency when adding new items, and does not require any explicit consistency-enforcement steps. Whenever a new example is added, we iterate through each of the previously-encountered items in the dataset and determine the leave-one-out prediction error for those items. To limit the computational effort required to compute these leave-one-out errors, we only consider the k items whose impact on the prediction at the insertion point is greatest. This also has the side effect of only focusing on the most relevant candidates for removal.

Instead of recording the direct leave-one-out prediction error, we actually sort the k items and record the resulting ranks. It should be reiterated that items which give a high leave-one-out prediction error should be retained, since those proved to be useful. We sort the leave-one-out prediction error in descending order, such that the first item is most beneficial and the last is most detrimental. Note that 1 is the best possible rank, and k is the worst. This rank-based approach has an important benefit: it does not require information about the range of possible errors possible. Thus, determining a threshold is more generalizable than, say, only using the raw leave-one-out error (which would require a threshold which depends on the domain of the regressor).

We keep a moving average of these ranks by using a fixed-size sliding window. This approach maintains a "first-in, first-out" queue of at most *maxHist* elements, such that when the *maxHist* + 1th item is added then the oldest item gets removed. Whenever a new item is added, the rankings of the leave-one-out prediction errors of the most impactful items in the dataset are appended to that item's sliding window queue. The estimated average for an item is then simply the average of the elements still within that item's queue. We note that this sliding window approach increases the memory requirements by a factor equal to the size of the sliding window, since at most *maxHist* of these elements must be stored. However, we found through informal testing that a sliding window approach performed better than other more memory-efficient moving average techniques such as an exponential moving average.

After adding the rankings to the k most-impactful items, we check to see if the new average rankings for these items exceeds that of the threshold, *driftThresh*. Those items with an average ranking higher than *driftThresh* are deleted from the stored dataset. Once this is completed, the new item is finally appended to the dataset.

Algorithm 3 describes our proposed method of drift compensation. Note that we use $\hat{y}(\boldsymbol{x}_i)$ to denote the current prediction for input $\boldsymbol{x}_i$, and $\widetilde{y}_j(\boldsymbol{x}_i)$ to denote element j's leave-one-out prediction for $\boldsymbol{x}_i$ (the value that would have been predicted for $\boldsymbol{x}_i$ if the jth element was never in the dataset).

Algorithm 3 Leave-one-out Drift Compensation

```
 1: function ADDWITHDRIFTCOMPENSATION($\boldsymbol{x}_i$, $y_i$)
 2:     mostImpactful ← min-heap of size $k$
 3:     for $\boldsymbol{x}_j \in data$ do
 4:         mostImpactful.add($|\widetilde{y}_j(\boldsymbol{x}_i) - \hat{y}(\boldsymbol{x}_i)|$, $\boldsymbol{x}_j$)
 5:     errors ← empty list
 6:     for $\boldsymbol{x}_j \in$ mostImpactful.values() do
 7:         errors.add($|\widetilde{y}_j(\boldsymbol{x}_i) - y_i|$, $\boldsymbol{x}_j$)
 8:     if ECDCa then
 9:         errors.sortAscending()
10:     else
11:         errors.sortDescending()
12:     for $j \leftarrow 1, 2, \ldots, minLeaves$ do
13:         error, $\boldsymbol{x}_j$ ← errors[$j$]
14:         histories[$\boldsymbol{x}_j$].add(error)
15:         if histories[$\boldsymbol{x}_j$].size() $\geq$ driftLen $\wedge$ histories[$\boldsymbol{x}_j$].average() $>$ driftThresh then
16:             histories.remove($\boldsymbol{x}_j$)
17:             data.remove($\boldsymbol{x}_j$)
18:     data.add($\boldsymbol{x}_i$, $y_i$)
19:     histories.add($\boldsymbol{x}_j$, new fixed-size queue of length driftLen)
```

We note that for some models, this approach is computationally intractable. In particular, if no information is known about the details of the regressor then a new model must be built for each item in the dataset upon each addition in order to determine the k most-impactful items. For models with computationally expensive training processes, such as deep neural networks, this would require days of computation per addition for even a moderately-sized dataset.

However, local regressors with finite-support kernels (those which have a 0-value for some items) have a desirable property: all items outside the kernel's support have no impact on the regressor's prediction. In particular, for a k-nearest neighbor regressor, all items besides those k which are nearest to the query point have no impact. Indeed, our choice of the variable k for both k-nearest neighbors and the size of the list of most-impactful items is deliberate. We can simply set mostImpactful to be the list of items returned from the k-nearest neighbors search, allowing us to skip directly to line 4 of Algorithm 3.

For kernel regression (local regression models with constant local models, see Section 4.3.1), it is possible to further increase the efficiency of computing the leave-one-out prediction. As described by Equation 4.10, it is possible to compute the leave-one-out prediction as a single $\mathcal{O}(1)$ operation. Thus, computing all of the k leave-one-out predictions requires only a single linear pass through the k items. Since this must already be done in order to make predictions, the overall runtime remains linear.

We note similarities between our proposed approach and that of Losing et al. (2016), which also proposed the use of drift compensation for a KNN model. However, we note important distinctions between this work and ours. First, their method is proposed specifically for usage in a classifier setting, as opposed to our proposed regression task. Secondly, their method uses a "two-tiered" system of memory, by which memories move from a “short-term” storage to a “long-term” one. Finally, their method relies on determining a fixed “window size”, essentially using a “leave-window-out” strategy as opposed to our “leave-one-out” strategy: an old but useful memory is evicted if it has enough outdated neighbors.

5.2.1 Ascending and Descending Error Ranking

We investigate versions of leave-one-out drift compensation which rank the leave-one-out errors in ascending as well as descending order. We use “ECDCa” to refer to agents which rank the errors in ascending order, and “ECDCd” to those which rank in descending order.

By ranking errors in ascending order, the ECDCa strategy effectively evicts memories with *high* leave-one-out errors. This should cause more productive elements to be deleted, while retaining those which were less so.

To demonstrate the effect of this ranking choice in a controlled environment, we tested both ECDCa and ECDCd on a synthetic drift problem. We note that for deterministic reinforcement learning environments and episodic agents, (1) the value function may have discontinuities, (2) updates to the model occur at the end of an episode, (3) changes in the value function typically do not affect all items, (4) changes in the value function only occur at the end of an episode, and (5) the value function may only monotonically increase. We thus devised a synthetic drift problem which includes all of these characteristics.

We setup the problem as a 64-dimensional regression with a non-stationary target function, corresponding to a 64-dimensional observation space for a reinforcement learning problem. The model was updated in batches, representing episodes from a reinforcement learning agent. The size of each episode was drawn from a uniform random distribution in the range $[200, 400]$. Each sample of the episode was a 64-dimensional vector with components drawn independently from $\mathcal{N}(0,1)$. After each episode we updated the model's drift compensation data structures (potentially evicting previously-inserted items) and then added the episode's samples to the model with the current target value.

The initial target function was set to to be a simple linear function:

$$f_0(\boldsymbol{x}) = 500(\boldsymbol{x} \cdot \boldsymbol{w}_0),$$

where $\boldsymbol{w}_0$ was a 64-dimensional vector with components drawn independently from $\mathcal{N}(0,1)$. After every 10 episodes we added an additional piece to the target function, corresponding to the agent discovering a new advantageous policy via exploration. On these epochs, we changed the cost function as follows:

$$f_t(\boldsymbol{x}) = \begin{cases} f_{t-1}(\boldsymbol{x}) + s & \text{if } \boldsymbol{x} \cdot \boldsymbol{w}_t > 0 \\ f_{t-1}(\boldsymbol{x}) & \text{otherwise} \end{cases},$$

where $\boldsymbol{w}_t$ was a 64-dimensional vector with components drawn independently from $\mathcal{N}(0,1)$ and s was drawn from a uniform random distribution in the range $[100, 300]$. We note that $\boldsymbol{w}_t$ is the normal vector of a random hyperplane passing through the origin, adding a fixed offset to points on one side of the hyperplane while not affecting those on the other side. Thus, each occurrence of drift affects exactly half of the potential observation space.

We simulated 1000 episodes (and thus 100 occurrences of drift) on a k-nearest neighbors model. The KForest for approximate nearest neighbors used the setting which we previously found to have 76.82% accuracy on Ms. Pacman images (see Table 3.2), and used the $K = 10$ nearest neighbors to make predictions. The models used a fixed kernel for weighting (simple k-nearest neighbors), and used kernel regression (constant local model).

We tested three types of drift compensation strategies: the ECDCa leave-one-out strategy, the ECDCd leave-one-out strategy, and no drift strategy. We noted the error for each sample and then averaged these errors across each epoch.

As can be seen in Figure 5.2, this ECDCa strategy produced the worst results on this synthetic environment, being outperformed even by the model with no drift strategy. Additionally, the average error was 2890.4 for the ECDCa drift strategy, 2599.7 for no drift strategy, and 1489.9 for the ECDCd drift strategy. This gives evidence that the ECDCa does indeed delete items which are useful.

However, we strangely found that the ECDCa agents produced markedly *better* results in the full reinforcement learning paradigm (as can be seen in the following sections). We thus present results for both ECDCa and ECDCd agents in the following section.

5.3 Episodic Control with Drift Compensation: Thresholds

From here on in this thesis, we refer to RL algorithms using this leave-one-out drift compensation scheme as Episodic Control with Drift Compensation (ECDC). All other aspects of the algorithm remain the same as MFEC, except that ECDC compensates for drift by using the algorithm described in Algorithm 3.

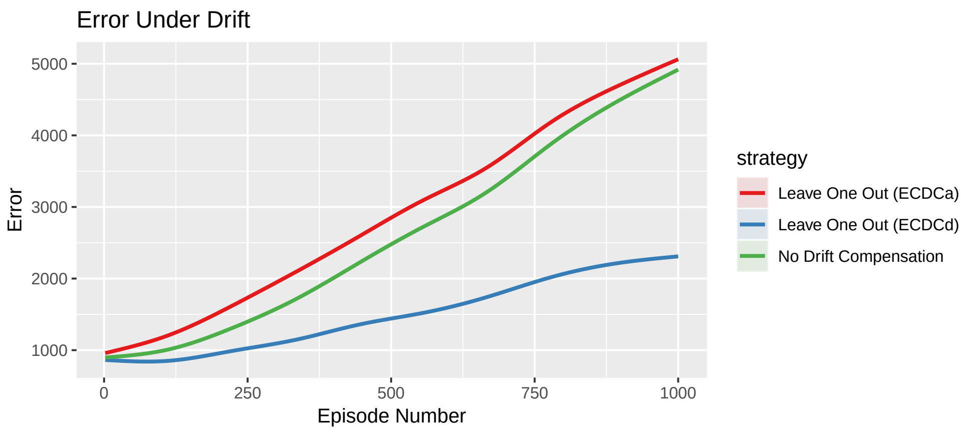


Figure 5.2: Error under drift for the synthetic drift problem described in Section 5.2.1 for different drift strategies.

We note one special case for drift compensation which occurs quite regularly in episodic control: it is undefined what should occur when adding an item which exactly matches one which is already in our dataset. The MFEC algorithm prescribes that when making a prediction in such a case, the highest return observed from that state should be returned, and thus no other items have any impact. In this view, adding an item with an exact match should not update any of the rank histories. We describe this type of drift as "NoExact".

However, an alternate view would be that what we really care about with drift regression is not what the regressor would predict, but rather about whether or not nearby data points have an outdated estimate. In this perspective, the k-nearest neighbors should still be investigated and updated for drift, regardless of whether or not the data point to be added is an exact match. We describe this type of drift as "Exact".

In addition, there is the important hyperparameter of *driftThresh* which we expect will have a large impact on the performance of the algorithm. Setting this too low will result in the algorithm being too eager to delete items, whereas setting it too high would cause it to never perform any actual drift compensation. Further, we expect this parameter has an interaction with the chosen drift type, since DriftExact will result in much more drift compensation steps than DriftNoExact.

Thus, we tested both of these parameters in tandem. Similar to our experiment in Section 3.5, we conducted multiple trials using the Ms. Pacman environment from the OpenAI Gym reinforment learning library (Brockman et al., 2016). For each trial, let an agent act in the environment for 16 hours or 100 million frames, whichever came first. At the end of each episode, we recorded statistics about that episode.

Our MFEC agent used experience buffers which held up to 1,000,000 experiences for each possible action. The agent used a transformed version of the screen pixels as input such that each observation was represented as a 64-dimensional vector, as explained in Section 3.5.

The KForests underlying the experience buffers used the setting which we

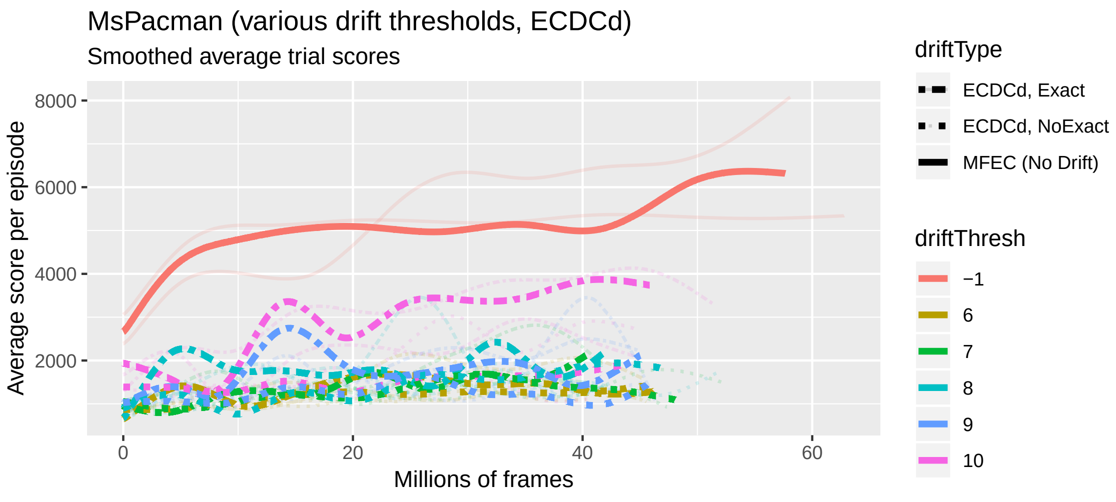


Figure 5.3: Ms. Pacman results for various *driftType* and *driftThresh* settings, with a descending drift strategy.

previously found to have 76.82% accuracy on Ms. Pacman images (see Table 3.2). The agents used the $K = 11$ nearest neighbors for approximating the values of unseen frames, used an epsilon-greedy exploration strategy ($\epsilon = 0.002$), and repeated their actions for 4 frames between action decisions. The agents used a fixed kernel for weighting (simple k-nearest neighbors), and used kernel regression (constant local model). In the *histories* list, the agents used *driftLen* $= 10$ for each item.

Since we expected there would be an interaction effect between the choice of drift type and *driftThresh* settings, we tested all possible combinations of both drift types and all tested *driftThresh* values.

As mentioned previously, we investigated two forms of leave-one-out drift compensation, distinguished by the ordering of leave-one-out error rankings. As noted above, we use ECDCa to denote ECDC agents which use leave-one-out ascending error ranking, and ECDCd to denote ECDC agents which use descending leave-one-out error ranking. We present results for the ECDCd agents in Section 5.3.1, and then present the results for the ECDCa in Section 5.3.2.

5.3.1 Results: ECDCd (Descending Drift)

For this experiment, we explored ECDCd agents with *driftThresh* settings ranging between 6 and 10, in increments of 1. Since we expected there to be an interaction effect between the choice of drift type and *driftThresh* settings, we tested all possible combinations of both drift types and all 5 *driftThresh* values. For reference, we also compare against the results of an agent with no drift compensation (and thus a *driftThresh* of -1). For each setting, we performed three trials, for a total of $3 \times (5 \times 2 + 1)$ trials total. Figure 5.3 displays the results from this experiment.

It is notable that all of the ECDCd agents performed worse than the MFEC baseline. Indeed, not even a single trial of an ECDCd agent outperformed the worst trial of the MFEC agent. In general, the Exact drift agents performed slightly better than their NoExact counterparts, but this effect was relatively small for all *driftThresh* settings other than 10.

The best-performing ECDCd agent was the one using Exact drift with a *driftThresh* of 10, the setting where the drift compensation has least effect. Under this setting, ECDCd only evicts items which receive the highest rank for *driftLen* episodes in a row, making this the most conservative possible drift compensation setting.

5.3.2 Results: ECDCa (Ascending Drift)

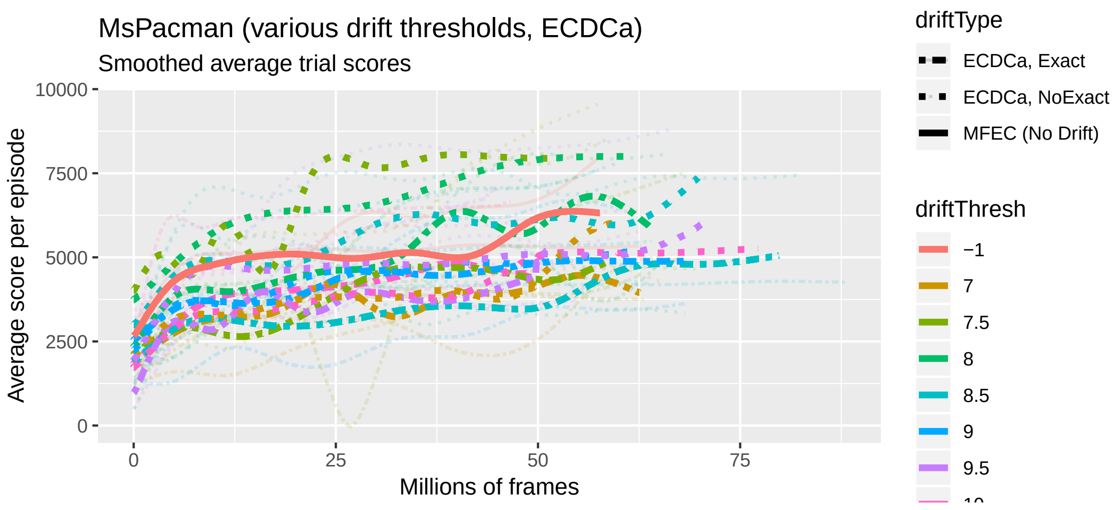
For this experiment, we explored ECDCa agents with *driftThresh* settings ranging between 7 and 10, in increments of 0.5. Since we expected there to be an interaction effect between the choice of drift type and *driftThresh* settings, we tested all possible combinations of both drift types and all 7 *driftThresh* values. For reference, we also compare against the results of an agent with no drift compensation (and thus a *driftThresh* of -1). For each setting, we performed three trials, for a total of $3 \times (7 \times 2 + 1)$ trials total. Figure 5.4 displays the results from this experiment.

We first note that the ECDCa agents as a whole performed much better than their ECDCd counterparts (compare to Figure 5.3). We consider this a surprising and unintuitive result. Somehow, it seems as though the fact that the drift compensation which removes *productive* memories actually improves the performance of the agent. Additional investigation is required to explain this phenomenon. However, given the marked difference in performance between ECDCd and ECDCa, we only evaluate the performance of ECDCa agents in future experiments in this thesis.

One prominent result is that for ECDCa agents, the NoExact drift type consistently produced better scores than the Exact type in almost every *driftThresh* level. It would appear that in this domain it is not useful to update the drift statistics for items nearby an item if that item is already in the dataset. We note that the MFEC algorithm already performs a simple method of drift compensation for exact matches by using the max operation, so this may be sufficient on its own to handle this case. Drift compensation only appears to be useful for novel additions, which do not fall under this domain.

It is also interesting to observe the importance of the exact threshold setting. For the NoExact drift type, settings of 7.5 and 8 produced significantly better results than the vanilla MFEC in the long run. We note that one trial of *driftThresh* $= 7$ produced the best results of any trial, although the median performance of this setting performed similarly to MFEC. Further, the Exact drift type with *driftThresh* $= 8$ produced very similar results as MFEC. However, in the small-data regime (10 million frames), only NoExact with *driftThresh* of 8 produced a slightly better median result than MFEC. Given that *driftThresh* $=$ 8 produced the best results with both NoExact and Exact drift types on both the small-data and the full 16-hour regimes, we use this setting of *drifthThresh* for the remainder of our experiments in this thesis.

Thus, we conclude that the primary benefit of drift compensation comes in the long run, when exploitation runs its course and MFEC has difficulties finding new advantageous paths. This could be an intuitive result if this weren't for ECDCa's poor results on the synthetic problem, as we could imagine that



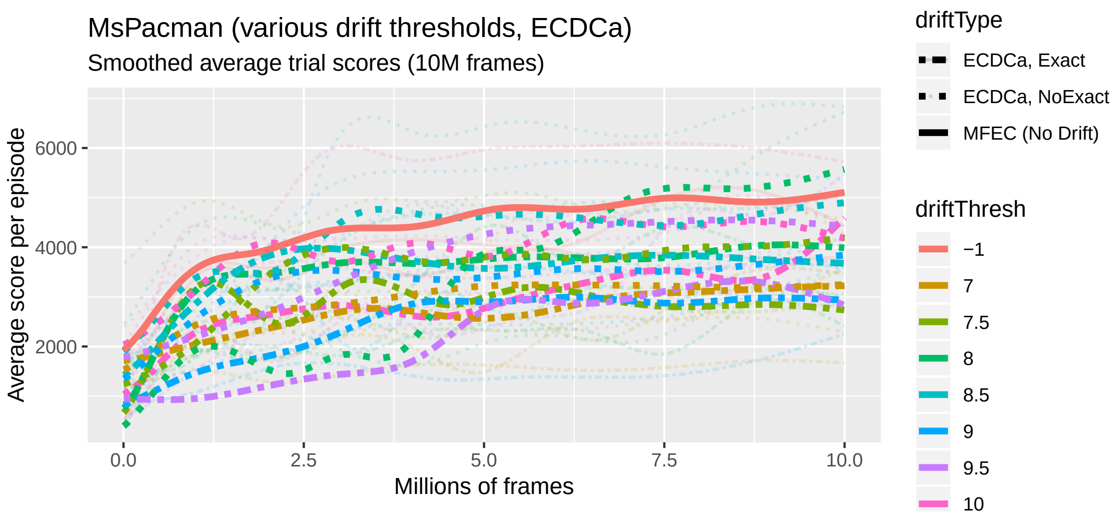


Figure 5.4: Ms. Pacman results for various *driftType* and *driftThresh* settings, with an ascending drift strategy.

drift compensation can help the agent determine when knowledge about some states is outdated and should be re-explored. However, the fact that ECDCa so significantly outperformed ECDCd raises doubt towards this interpretation. Regardless, this does raise an interesting question: does ascending drift compensation have varying impact at different levels of exploration?

5.4 Episodic Control with Drift Compensation: Exploration

The primary benefit of ECDCa's drift compensation seems to present itself in the long run, when exploitation runs its course and MFEC has difficulties finding new advantageous paths. Thus, we wish to investigate the impact drift compensation has on agents with various levels of exploration. To reiterate, ECDCa's exploration is handled by an epsilon-greedy policy, where a random action is taken with probability ϵ.

We explored ϵ at various orders of magnitude and investigated the relative performance between MFEC and ECDCa. We expect that the performance difference between ECDCa and MFEC will be the larger for smaller ϵ values, since in these domains drift plays an increasingly large role in the amount of novel states encountered. Since a lower exploration level means that states are more likely to be re-visited, we again explored the behaviors of both the Exact and NoExact drift types.

Given that it produced the best results in the previous section for both drift types, we used $\mathit{driftThresh} = 8$.

We explored values of ϵ between 5×10^{-2} and 5×10^{-5}, on a log scale in increments of negative powers of 10. Note that the previous experiment used $\epsilon = 2 \times 10^{-3}$, which is near the middle of this scale.

All other settings and parameters were identical to those described in the previous section.

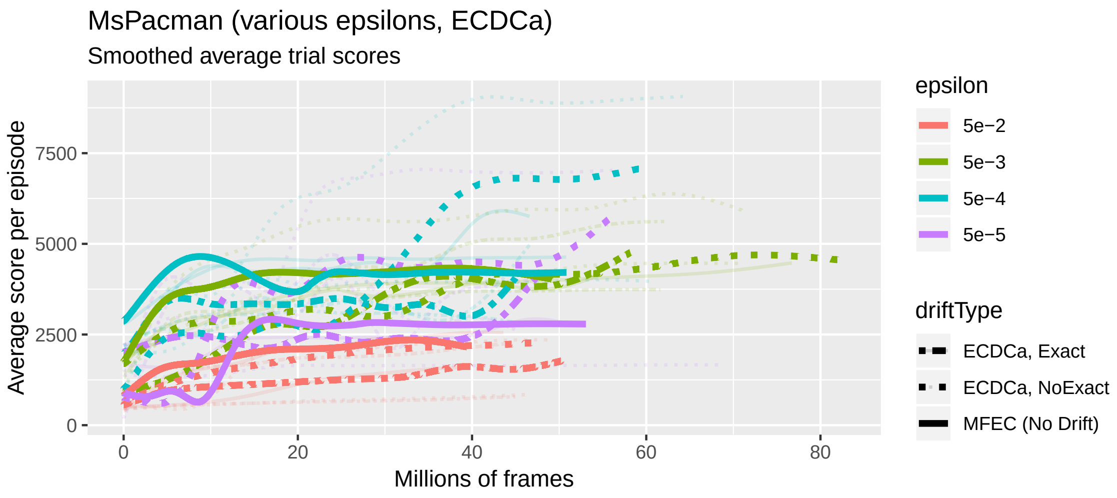
5.4.1 Results

Figure 5.5 displays the results from this experiment. Interestingly, in the low-data regime the Exact strategy slightly outperformed NoExact for $\epsilon = 5 \times 10^{-2}$ and $\epsilon = 5 \times 10^{-5}$. In the full 16-hour regime, however, NoExact again performed significantly better than Exact in all ϵ settings except $\epsilon = 5 \times 10^{-3}$, where the two drift settings performed about as well.

MFEC again performed better in the low-data regime than either NoExact or Exact in most cases. One notable exception is the case where $\epsilon = 5 \times 10^{-5}$, in which case the learning curve plateaued a little after 1 million frames. Presumably, this low of an exploration setting allowed the beneficial effects of drift compensation to present themselves earlier.

In the 16-hour regime, NoExact produced similar results to MFEC for the higher $\epsilon = 5 \times 10^{-2}$ and $\epsilon = 5 \times 10^{-3}$ settings. However, in the lower ϵ settings NoExact showed a considerable performance gain after the MFEC agents plateaued, presumably due to an over-propensity to exploit rather than explore.

Using $\epsilon = 5 \times 10^{-4}$ produced the best results overall, with MFEC producing the best results after 10 million frames and NoExact producing the best after 16 hours.



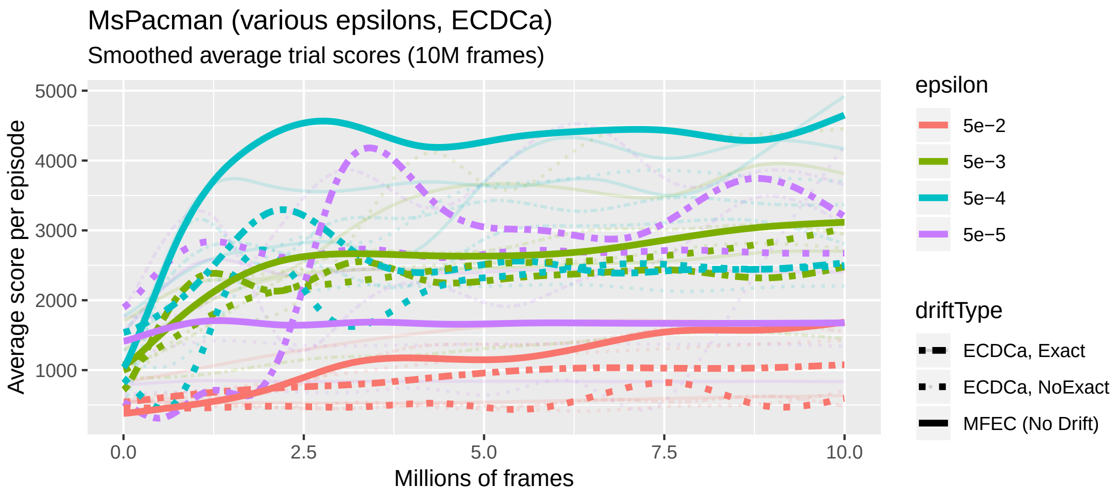


Figure 5.5: Ms. Pacman results for various epsilon exploration settings.

5.5 Summary

In this chapter, we introduced the topic of concept drift and explained its application to reinforcement learning. We then explained a method of drift compensation which uses the leave-one-out error as a signal for determining when drift occurs for individual elements.

We then implemented this method, producing an algorithm we refer to as Episodic Control with Drift Compensation (ECDC). However, we discovered a version of this algorithm, which produced worse-than-baseline results on a synthetic drift problem, proved to produce significantly *better* results on our Ms. Pacman experiments. The benefits of ECDCa primarily showed their effects in the long term, beyond the small-data regime of 10 million frames.

Given this, we then experimented to determine if the exploration rate affected the impact of the drift compensation. We found that ECDCa agents generally outperformed their respective MFEC agents in the long run, and that this discrepancy was larger at lower exploration rates.

Chapter 6

Full Atari Evaluation

Thus far in this thesis, all of our experiments have been conducted in the Ms. Pacman Atari environment. In this section, we explore in addition four other Atari environments, namely SpaceInvaders, Qbert, Frostbite, and River Raid. We give a description and a brief overview of the mechanics and reward properties of each of these games in Appendix A.

We further explore the individual contributions from Chapters 4 and 5. Namely, in Chapter 4 we found that MFEC with the inverse square distance kernel produced slightly better results than with the constant kernel. In Chapter 5, we found that a form of leave-one-out drift compensation (ECDCa) produced better results than vanilla MFEC. However, we only tested the effect of these two improvements in isolation from one another. In this chapter, we integrate these two suggestions and explore their relative impacts on learning performance across all five games.

6.1 Full Atari Experiments

Similar to our experiment in Section 3.5, we conducted multiple trials using the various environments from the OpenAI Gym reinforcement learning library (Brockman et al., 2016). For each trial, we let an agent act in the environment for 24 hours or 100 million frames, whichever came first. At the end of each episode, we recorded statistics about that episode.

The agents used experience buffers which held up to 1,000,000 experiences for each possible action. The agents used a transformed version of the screen pixels as input such that each observation was represented as a 64-dimensional vector, as explained in Section 3.5.

The KForests underlying the experience buffers used the setting which we previously found to have 76.82% accuracy on Ms. Pacman images (see Table 3.2). The agents used the $K = 11$ nearest neighbors for approximating the values of unseen frames, used an epsilon-greedy exploration strategy ($\epsilon = 0.0005$), and repeated their actions for 4 frames between action decisions. We note that this ϵ exploration value is an order of magnitude lower than in previous experiments, following the results of Section 5.4.

We explored the MFEC and ECDCa algorithms (with MFEC serving as a baseline), as well as the constant and inverse square distance kernels (with the constant kernel serving as a baseline). For the ECDCa agents, we chose to set $driftThresh = 8$, in accordance to the best-performing agent from Section 5.3.2. Thus, we tested all four possible combinations of kernels and algorithm types. For each condition, we performed three trials, for a total of $3 \times 2 \times 2$ trials per environment, for a total of 60 trials across all five environments.

Since the MFEC and ECDCa agents maintain a seperate buffer for each action, the time required for agents to choose an action at each step is proportional to the number of discrete actions. We note that Frostbite and River Raid have 18 potential actions, as opposed to Qbert's and Space Invader's 6 and Ms. Pacman's 9. Thus, the agents were able to obtain less experience in these games over the 24-hour period relative to the other games. This is reflected in the scale of the x axis in Figures 6.4 and 6.5.

6.1.1 Ms. Pacman

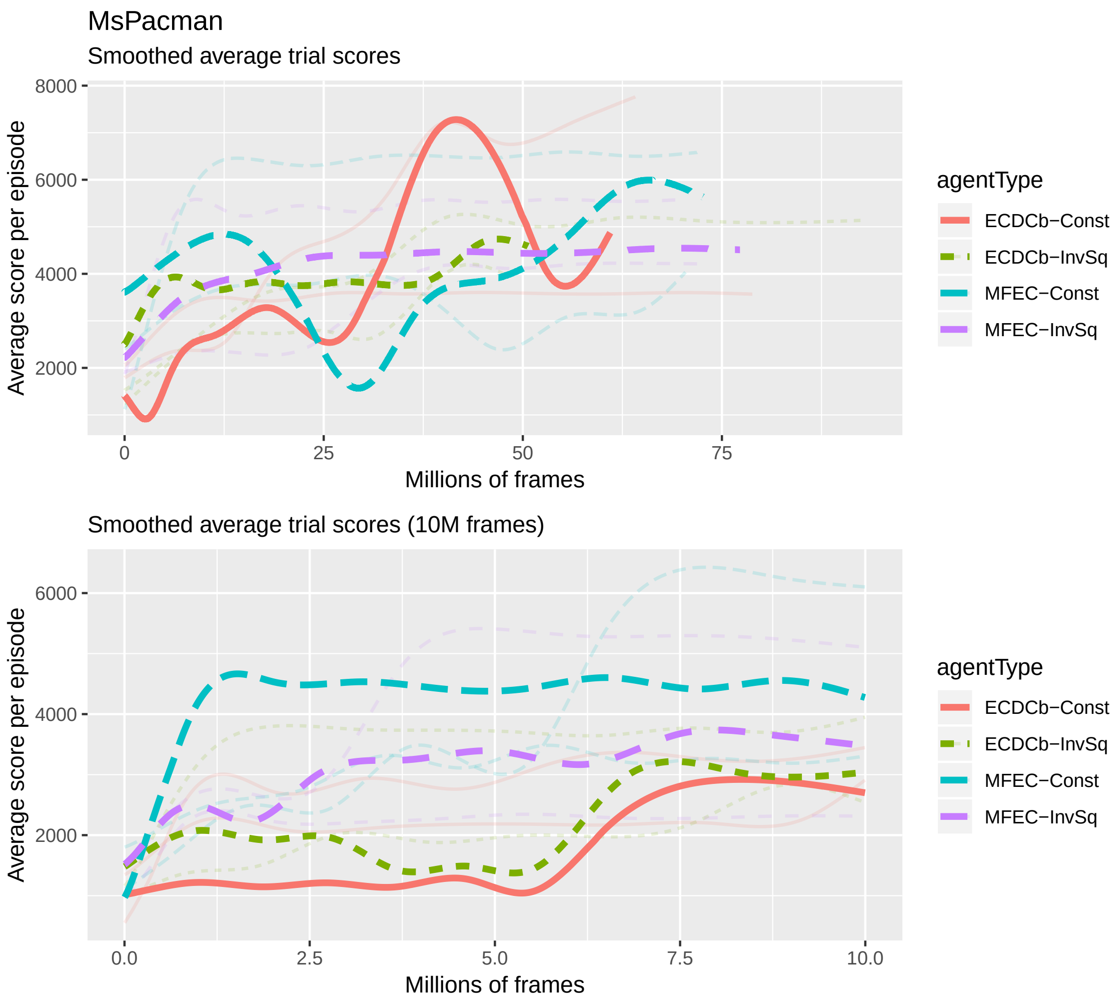


Figure 6.1: Combined results for Ms. Pacman.

Figure 6.1 displays the results on the Ms. Pacman environment. The best-performing individual agent in the full 24-hour regime was an ECDCa-Const agent, which achieved a score of nearly 8000. This is comparable to the results of Section 5.4. However, the median ECDCa-Const agent seemed to suffer a catastrophic loss at around 40 million frames, likely due to the deletion of a pivotal memory. This demonstrates the potential instability of the drift compensation.

It is interesting to note that the addition of the inverse square distance kernel to the ECDCa agent made it generally perform *worse* than the vanilla ECDCa-Const agent. The performance of the ECDCa-InvSq agent was similar to that of the MFEC-InvSq agent, despite the fact that the ECDCa-InvSq agents evicted a similar number of items as the ECDCa-Const agent.

It is also worth noting that the baseline MFEC-Const agents produced the best median scores on the small-data regime, and also outperformed both types of InvSq agents in the long run.

6.1.2 Space Invaders

Figure 6.2 displays the results on the Space Invaders environment. The best-performing individual agent in the full 24-hour regime was an ECDCa-Const agent. However, we again note an apparent case of catastrophic forgetting witnessed in one ECDCa-Const agent which went from scoring nearly 3000 to nearly 0 after around 50 million frames. This further illustrates the potential instability introduced by drift compensation.

We note that Space Invaders required the most frames of all five tested environments, averaging 8722 frames per episode across all agents. Given a fixed ϵ, we would expect to see larger variances in scores for environments with a larger number of frames per episode, since it is more likely that an early deviation from a known good path will have a larger impact down the line.

The ECDCa-InvSq agents produced the second-best median scores. Although these agents did obtain lower high scores than the ECDCa-Const agents, the ECDCa-InvSq agents seemed to display less catastrophic forgetting than their ECDCa-Const counterparts. Additionally, the median ECDCa-InvSq agent scored best for the majority of the low-data regime.

6.1.3 Qbert

Figure 6.3 displays the results on the Qbert environment. The best-performing individual agent in the full 24-hour regime was an ECDCa-InvSq agent. For instance, the highest-scoring ECDCa-InvSq agent achieved a score of nearly 10,000 points after only 10 million frames, a feat only matched after 60 million frames by the best ECDCa-Const and MFEC-InvSq agents. Indeed, ECDCa-InvSq was the only agent type that produced a median score that exceeded this threshold.

In comparison to previous environments, there were not many apparent instances of catastrophic forgetting.

The MFEC-InvSq agents in particular had highly variable results. The worst-performing MFEC-InvSq agent plateaued at around 2,000 points, tied for worst agent alongside MFEC-Const. Likewise, the median MFEC-InvSq agent performed worst of all median agents in the long term. However, the

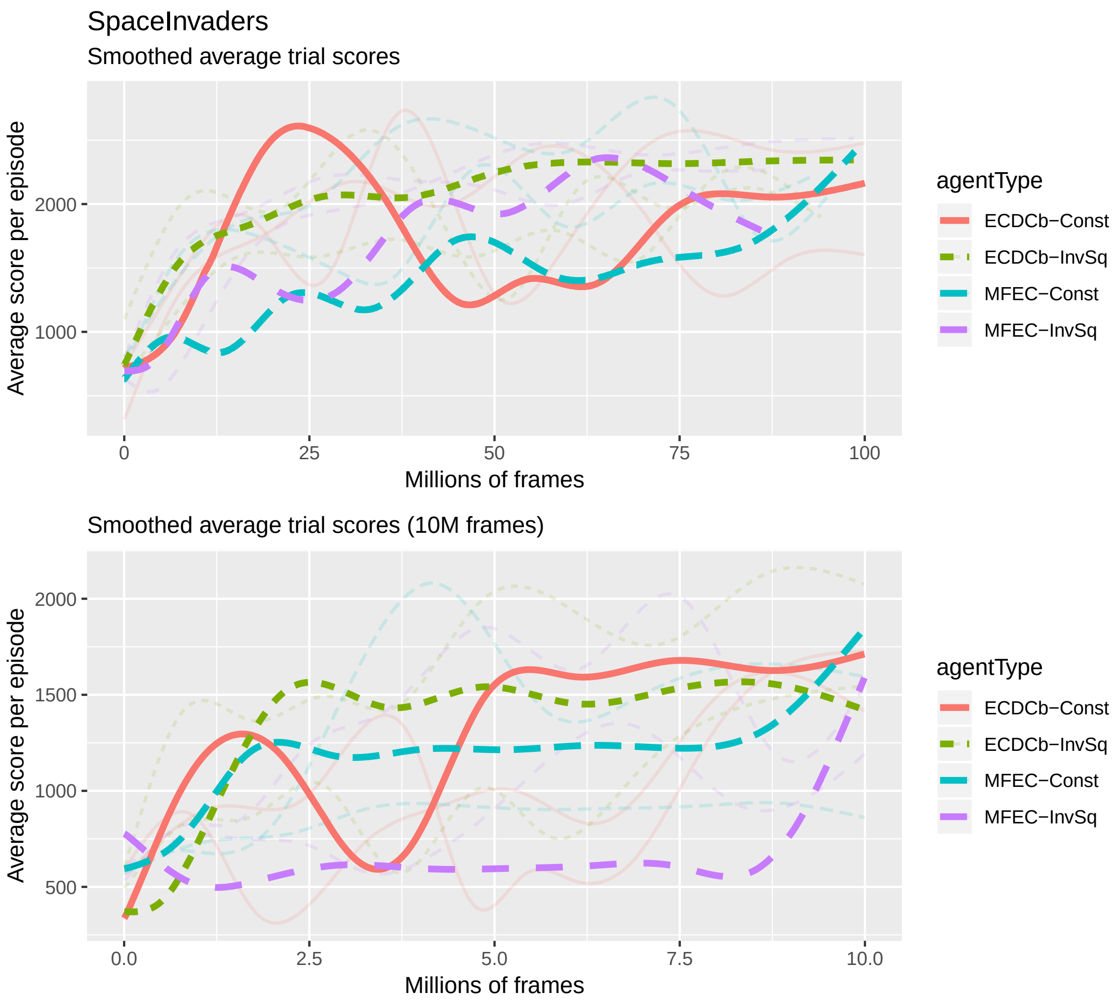


Figure 6.2: Combined results for SpaceInvaders.

best-performing MFEC-InvSq agent was one of only four agents to break the 10,000-point barrier, and in the small-data regime the median MFEC-InvSq agent performed best overall.

6.1.4 Frostbite

Figure 6.4 displays the results on the Frostbite environment.

As noted previously, the Frostbite environment contains 18 possible input combinations, and thus requires additional time to select an action compared to the other environments. Additionally, the Frostbite environment required more frames per episode, especially for successful agents: for trials where the agent scored above 1000, the average episode was nearly 19,000 frames long. Due to both of these facts, agents were able to experience fewer episodes in the Frostbite environment during the allotted 24 hours than in any other environment.

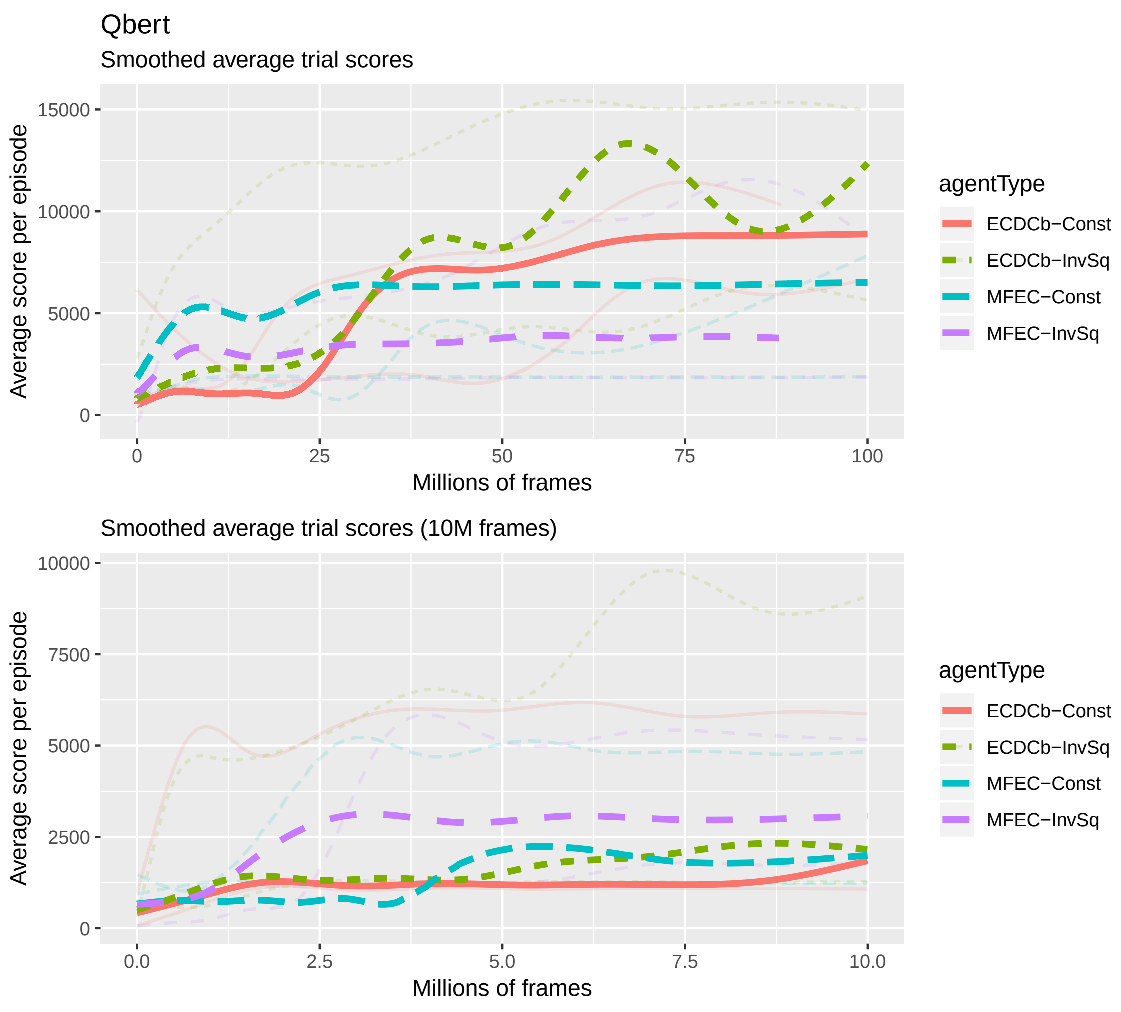


Figure 6.3: Combined results for Qbert.

We make note that most of the smoothed fits "curl" downwards in the far tail. This is unfortunately an artifact of the smoothing algorithm used to summarize the returns, which unfortunately places high emphasis on late episodes which achieved a low score. This tail behavior is not indicative of the capabilities of the agent, but unfortunately is on display here due to the relatively small number of episodes encountered by the agents.

These caveats aside, in the Frostbite environment the MFEC-InvSq agent produced the best median and maximum scores, for both the full 24 hours regime and the 10-million frame regime.

We do note, however, that every agent type had at least one agent which never made a score above 1000. ECDCa-InvSq produced two such agents, and every other agent type produced one. Each of these agents plateaued at somewhere between 200 and 290 points. Considering the homogeneity of these underperforming agents, combined with the erratic distribution across agent types,

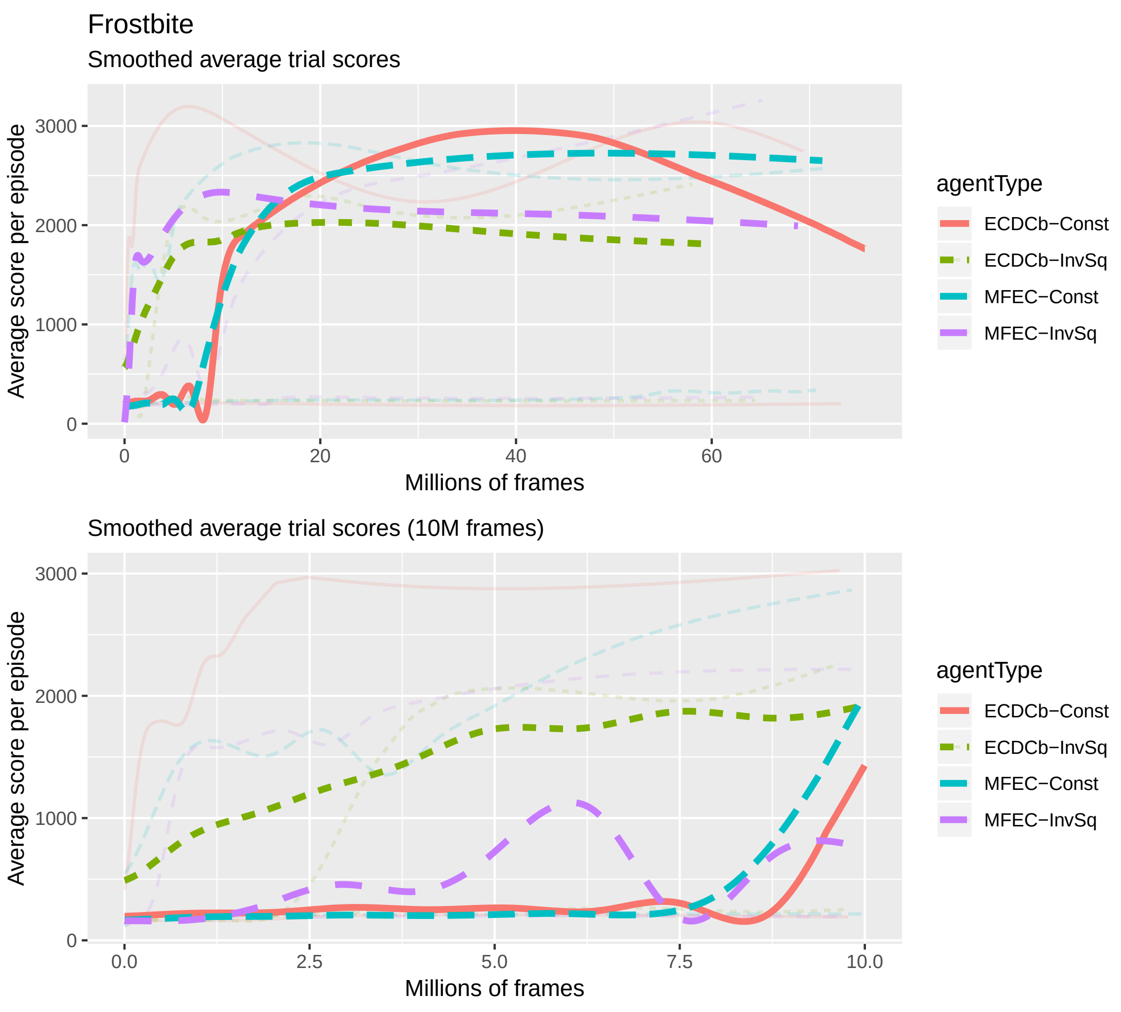


Figure 6.4: Combined results for Frostbite.

we believe this result is a consequence of exploration, rather than a meaningful difference between agent types.

6.1.5 River Raid

Figure 6.5 displays the results on the River Raid environment. In the River Raid environment, the best-performing median agent was MFEC-InvSq, which produced the highest median scores in both the full 24-hour and 10-million frame regimes.

Similarly to the Frostbite environment, the River Raid environment also has 18 possible input combinations. Because of this, the agent with the most experience in this environment still had less than 40 million frames of experience. Thus, even though the MFEC-InvSq agents produced the best median scores in both regimes, there is not as much discrepancy between the two regimes as in

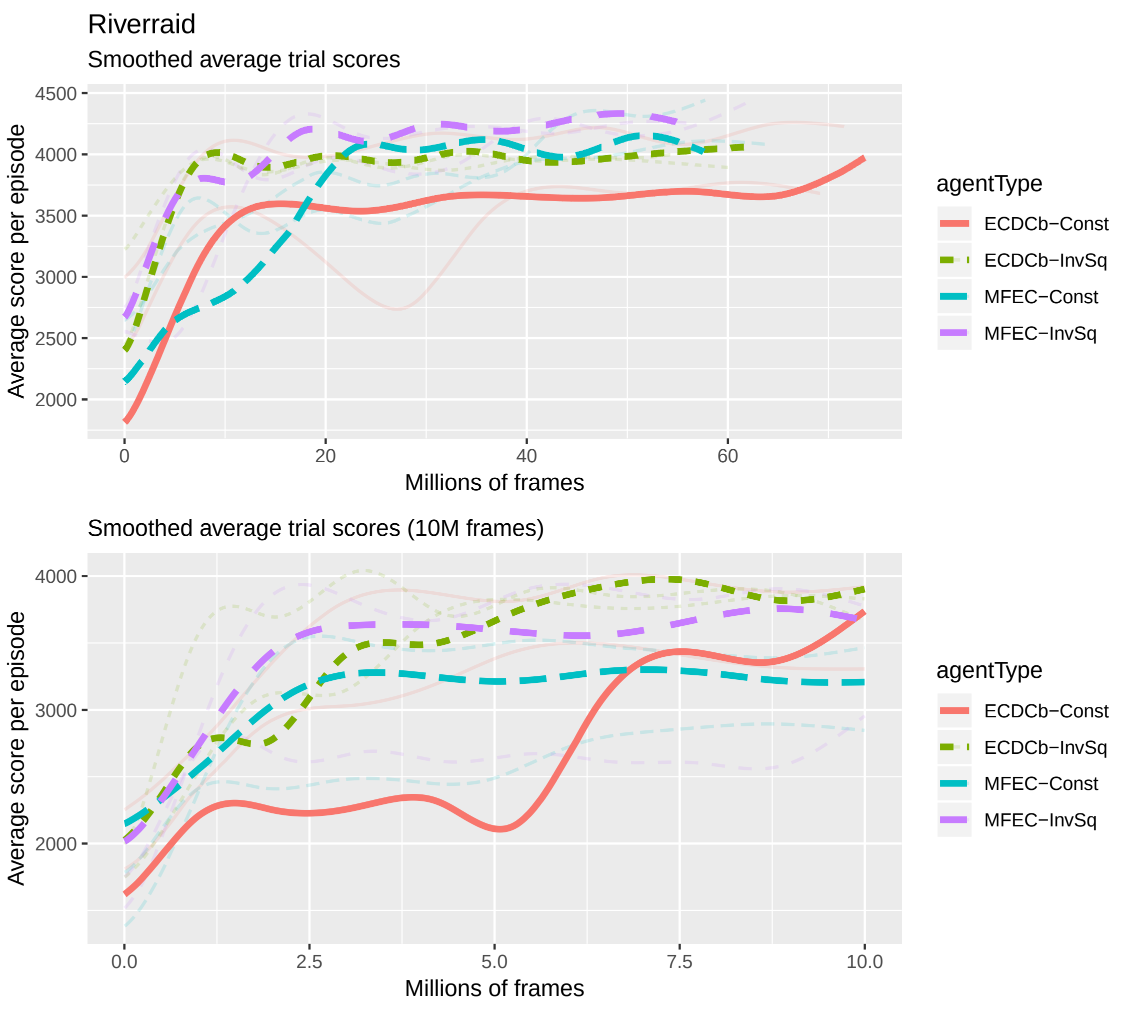


Figure 6.5: Combined results for River Raid.

previous environments. There was differentiation in the Frostbite environment despite this phenomenon, but that was mostly due to the difficulty of exploring to find a good initial strategy to latch onto. Additionally, the number of frames per episode was much smaller on average than those for Frostbite, so agents were able to experience more episodes while experiencing fewer frames.

There was not many apparent instances of catastrophic forgetting, although the relatively limited number of frames of experience may have limited the opportunity for such forgetting to take place.

6.2 Rank Analysis

In addition to the environment-by-environment results analysis, we also give a general overview of the performance across all agents by ranking the maximum

Table 6.1: Maximum score achieved by best, median, and worst agents across agent types and games (full 24-hour regime).

	MFEC-Const	MFEC-InvSq	ECDCa-Const	ECDCa-InvSq
Ms. Pacman Best (24 hours)	7090 (2)	5870 (3)	**9930 (1)**	5500 (4)
Ms. Pacman Median (24 hours)	6500 (2)	4980 (4)	**8800 (1)**	5100 (3)
Ms. Pacman Worst (24 hours)	**5090 (1)**	4760 (2)	3880 (4)	4650 (3)
Frostbite Best (24 hours)	4370 (2)	**5450 (1)**	3810 (3)	3700 (4)
Frostbite Median (24 hours)	3480 (3)	**3800 (1)**	3760 (2)	290 (4)
Frostbite Worst (24 hours)	280 (2)	200 (4)	240 (3)	**290 (1)**
Qbert Best (24 hours)	8000 (4)	14400 (2)	13325 (3)	**24175 (1)**
Qbert Median (24 hours)	6925 (3)	4150 (4)	10750 (2)	**19725 (1)**
Qbert Worst (24 hours)	3125 (3)	2400 (4)	6775 (2)	**7550 (1)**
River Raid Best (24 hours)	**5910 (1)**	5180 (2)	4940 (3)	4910 (4)
River Raid Median (24 hours)	4610 (2)	**4910 (1)**	4510 (3)	3740 (4)
River Raid Worst (24 hours)	4130 (2)	4090 (3)	**4270 (1)**	3320 (4)
Space Invaders Best (24 hours)	**4165 (1)**	4120 (2)	3965 (3)	3335 (4)
Space Invaders Median (24 hours)	3520 (2)	3380 (3)	**3930 (1)**	3270 (4)
Space Invaders Worst (24 hours)	3150 (4)	3300 (2)	**3670 (1)**	3155 (3)
Average Rank (24 hours)	2.27	2.53	2.20	3.00

score achieved by each agent across all games and all agent types. Table 6.1 displays these scores for the full 24-hour regime, and Table 6.2 displays those for the small-data 10-million frame regime. Note that these tables depict the actual highest score achieved by the agent, unlike the smoothed averages which were presented previously. These smoothed averages will by necessity always under-represent the maximum score achieved by the agent since they take into account all trials, including those with detrimental exploration.

On a whole, agents had similar relative performances across the 24-hour and 10-million frame regimes. ECDCa-InvSq produced the worst average ranking of all agent types, followed by MFEC-InvSq. MFEC-Const produced slightly better average rankings than ECDCa-Const in the 10-million regime, whereas ECDCa-Const performed better than MFEC-Const on the 24-hour regime.

The 24-hour regime produced more polarized results than the 10-million frame regime. ECDCa-InvSq, the worst-performing agent across both regimes, had an average ranking of 2.60 on the 10-million frame regime and 3.00 on the 24-hour regime. Meanwhile, the best-performing agent produced an average ranking of 2.37 on the 10-million frame regime and 2.20 on the 24-hour regime. This widening performance gap indicates that the manipulations have more prominent effects if the agents are allowed to have more experience, as is expected.

It is striking that agents which used the InvSq kernel always produced worse results than their Const kernel counterparts. That is, MFEC-Const outperformed MFEC-InvSq, and ECDCa-Const outperformed ECDCa-InvSq. This is even the case for the Ms. Pacman environment in the 24-hour regime, despite our results presented in Section 4.6.1 which seemed to show InvSq outperforming Const on median. The only difference between these two experiments was the reduction of ϵ by a factor of 10, as described in the introduction to this

Table 6.2: Maximum score achieved by best, median, and worst agents across agent types and games (small-data regime).

	MFEC-Const	MFEC-InvSq	ECDCa-Const	ECDCa-InvSq
Ms. Pacman Best (10M frames)	**7000 (1)**	5710 (2)	3640 (4)	4130 (3)
Ms. Pacman Median (10M frames)	**4920 (1)**	4120 (2)	3070 (4)	3230 (3)
Ms. Pacman Worst (10M frames)	**3820 (1)**	2580 (4)	3010 (3)	3020 (2)
Frostbite Best (10M frames)	3480 (3)	**4020 (1)**	3760 (2)	3280 (4)
Frostbite Median (10M frames)	2730 (2)	**3800 (1)**	1840 (3)	290 (4)
Frostbite Worst (10M frames)	230 (3)	200 (4)	240 (2)	**280 (1)**
Qbert Best (10M frames)	5200 (4)	5650 (3)	6700 (2)	**10350 (1)**
Qbert Median (10M frames)	2400 (2)	**3250 (1)**	1725 (4)	2400 (2)
Qbert Worst (10M frames)	1775 (2)	**1825 (1)**	1100 (4)	1300 (3)
River Raid Best (10M frames)	4020 (4)	4910 (2)	**4940 (1)**	4910 (2)
River Raid Median (10M frames)	3860 (3)	4230 (2)	**4270 (1)**	3380 (4)
River Raid Worst (10M frames)	3300 (2)	3180 (4)	**4250 (1)**	3210 (3)
Space Invaders Best (10M frames)	2680 (2)	2545 (4)	**3725 (1)**	2665 (3)
Space Invaders Median (10M frames)	**2500 (1)**	1910 (4)	2065 (3)	2175 (2)
Space Invaders Worst (10M frames)	1005 (4)	1795 (3)	2030 (2)	**2160 (1)**
Average Rank (10M frames)	2.37	2.57	2.47	2.60

chapter. This would seem to suggest an unforeseen interaction between the rate of exploration and the choice of kernel type. However, we do note the notable exception of Qbert with the 24-hour regime, which produced the best median agent by a substantial margin.

The environments on which ECDCa agents produced the worst average results on the 24-hour regime, Frostbite and River Raid, were those in which the agents had the least amount of experience in (as described previously). We are particularly interested in re-running these experiments with a larger time allocation so that a full 100 million frames could be completed.

6.3 Summary

In this chapter, we combined suggestions from previous chapters and performed experiments on five different Atari environments. There were some instances of ECDCa agents experiencing catastrophic forgetting, likely due to the deletion of a key memory. While there was a great deal of variability within each game, analyzing the performance of the median agents across environments portrayed a clearer picture.

In the small-data regime, the MFEC agents with a constant kernel performed best, with the MFEC agents with a inverse square distance kernel performing nearly as well. In the full 24-hour regime, the ECDCa agents with a constant kernel performed best, despite the fact that two of the environments were not able to be thoroughly explored due to computational restrictions. This gives us more evidence that the drift compensation algorithm is most useful for increasing the long-term capacity of episodic control agents.

Chapter 7

Conclusion

In this thesis, we investigated and proposed improvements to a state-of-the-art reinforcement learning algorithm which uses k-nearest neighbor regression.

In Chapter 3, we designed and implemented a tree-based data structure for online approximate nearest neighbor queries, and then showed that it was competitive relative to other offline state-of-the-art tree-based data structures. We then discovered that some inaccuracies in the nearest neighbor data structure actually improved agent performance (to a point), and that the brute-force nearest neighbor search (which had 100% nearest neighbor accuracy) actually produced the worst results for the reinforcement learning agent.

In Chapter 4, we explored additional local regression techniques by using various kernel weightings and local models. Although there was not a huge difference, we found that the invSq and tricube kernel weightings produced slightly better results. Additionally, we generally found the linear local models produced worse results than their constant (kernel) counterparts.

In Chapter 5, we developed a method of using the leave-one-out prediction error upon addition to determine which memories to evict based on consistency, rather than age. Incorporating this method into the MFEC agent produced better results. However, a version of drift compensation which evicts *useful* memories proved to be beneficial. We found that this drift compensation scheme especially improves agent performance in longer evaluation regimes. The beneficial effect of this drift compensation also became more pronounced as the agent's random exploration rate became lower. We also found that the positive effect of this drift compensation typically showed itself later in the trials.

Finally, in Chapter 6, we performed a full evaluation across 5 different Atari games, and integrated prior results to explore their interaction. By comparing results from a small-data and full 24-hour testing regime, we found additional evidence that the drift compensation was much more beneficial in the long-term than in the short-term. However, there were a concerning amount of instances of catastrophic forgetting witnessed in the agents which employed drift compensation. The agent which combined the two earlier recommendations (the use of a invSq kernel and the use of drift compensation) produced the worst results overall, when averaged across games. Overall, in the long-term evaluation, the agent with drift compensation produced the best median agents when averaged across all games, while the base MFEC model produced the best results in the small-data regime. Additionally, we found more evidence that would indicate that ECDCa has a more pronounced effect in the long run of the trials.

7.1 Future Perspectives

The explorations and findings in this thesis raises many questions and avenues for further research in many different directions.

7.1.1 KForest

One puzzling result we discovered is that the brute-force nearest neighbor search produced the *worst* agents, with a relatively inaccurate setting producing the best results of those tested. The explanation for this intriguing behavior remains an open question, and should be addressed in future research. In particular, in Section 3.5.1 we described a potential experimental setup to test the effects of artificial added noise on agent performance.

We found a significant performance gap between the graph-based and tree-based algorithms, with graph-based algorithms drastically out-performing tree-based ones. Thus, future research should focus on adapting existing graph-based approaches for an online environment.

Nevertheless, there are multiple areas upon which the KForest algorithm could potentially be improved. One such improvement could see the tree use lower-dimensional random projections, potentially using a smaller dimensionality for distance comparison of nodes close to the root and gradually increasing the dimensionality as the tree works its way to the leaves.

Another area to potentially improve the KForest algorithm for local regression would be to use the labels as a splitting criterion. In its current state, the KForest structure splits nodes based solely on its size. However, an alternate scheme would be to only split those nodes in which there is a large disagreement among the labels. Such a scheme could potentially be similar to that proposed by Mathy et al. (2015).

Finally, the speed of KForest modifications could be increased by parallelizing the addition and deletion of items. This would exploit the fact that MFEC and ECDCa add to the tree in relatively large batches. It is also possible to parallelize the query interface so that multiple queries can be processed at once, potentially allowing for multiple agents to operate using the same KForest backing.

Lastly, we note that when benchmarking approximate nearest neighbor algorithms in high-dimensional spaces, the majority of the computational time is spent computing distances. Thus, for increased accuracy, it may be beneficial for benchmarks to explicitly count the number of distance computations made rather than measuring the wall time. Additionally, these benchmarks need to test multiple different search settings, sometimes without changing the construction of the graph. Rather than performing multiple searches of various depths, it would likely be faster to simply perform a single wide search and note when the correct items are added to the result. While both of these suggestions require additional effort from each library, decreasing the time required to perform a benchmark would tighten the feedback loop for the development of future algorithms.

7.1.2 Local Regression

Our experiments only investigated linear and constant local models, but other higher-order (polynomial) models could also be investigated. Additional research could be conducted to determine the effect of higher-order local models on algorithm performance, specifically to see if the general trend of more complex local models being out-performed by simpler ones still holds true.

Additionally, as pointed out in Section 6.2, there does seem to be some sort of interaction between the choice of kernel and ϵ on agent performance. It is possible that the choice of local model has a similar dependency. Thus, future research should explicitly investigate this phenomenon.

7.1.3 Drift Compensation

One puzzling result was the effectiveness of the drift compensation method which used ascending error ranking (ECDCa), and the ineffectiveness of the method with descending rankings (ECDCd). Although ECDCd outperformed ECDCa on a synthetic drift problem, the opposite was the case when tested on the Atari environment. The explanation for this intriguing behavior remains an open question, and should be addressed in future research.

One highly promising direction for future research is to incorporate the drift compensation of ECDCa to the Neural Episodic Control algorithm (Pritzel et al., 2017). This paper expands upon MFEC by using a neural network to learn a state representation, rather than relying upon a fixed random projection. We believe that our method of drift compensation would be especially complimentary to this adaptive state representation, since this effectively introduces a new form of drift. By incorporating drift compensation, a Neural Episodic Control agent could detect and actively remove "old" or "stale" states whose representations have drifted.

7.1.4 Episodic Control

Unfortunately, due to lack of computational resources, we were only able to perform a full evaluation on 5 of the 54 environments used as a standard benchmark in the Arcade Learning Environment (Bellemare et al., 2013). And even of the 5 environments we did test, some environments required more time than others to evaluate (as explained in Chapter 6). Moreover, we performed all our hyperparameter searches on the Ms. Pacman environment, leading to potential overfitting issues. Given additional resources, we would like to perform more thorough testing and evaluation on the full suite of Atari games.

In our experiments, agents performed the entire episode, as defined by the default "end-of-episode" indicator provided by the OpenAI Gym (Brockman et al., 2016). However, it is possible to instead train agents until a single life is lost. Although this makes some environments more difficult (particularly environments like Montezuma's Revenge, where interactions resulting in death permanently affect the episode), terminating on life loss would be a useful setting to increase training speed. This would be in line with the training scheme described by Hessel et al. (2017).

It should be possible to reduce the computational burden for local regression agents to choose actions. Instead of having separate KForests for each possible action, one could simply allow multi-dimensional labels for each item. Thus, the KForest query (the computational bottleneck) would only need to be performed once, and the action values could be filtered appropriately after the query is complete. Doing this would avoid the additional computation time to query multiple different KForest buffers for environments with larger action spaces, such as Frostbite and River Raid. This would be similar to most deep reinforcement learning approaches, which use a single neural network with multiple outputs to simulate multiple Q-functions.

Finally, we believe that it is possible to use a similar local regression for continuous control problems. Such a scheme would involve storing action-return pairs as labels, instead of simply the return values. Then, one could find the nearest neighbors whose states are most similar to the query point, and then perform gradient ascent on the labels of those neighbors to find an action which maximizes the predicted return.

Bibliography

Abbeel, P., Coates, A., Quigley, M., and Ng, A. Y. (2007). An application of reinforcement learning to aerobatic helicopter flight. In *Advances in neural information processing systems*, pages 1–8.

Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., and Munos, R. (2016). Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 1471–1479.

Bellemare, M. G., Dabney, W., and Munos, R. (2017). A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, pages 449–458.

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

Bernhardsson, E. (2017a). ann-benchmarks. `https://github.com/erikbern/ann-benchmarks`. version:2655c44979c06b95d52051febb6f65e46662c60e.

Bernhardsson, E. (2017b). annoy. `https://github.com/spotify/annoy`. version:76718ef62477f593b47042c7e0372ea3d536d9d7.

Blundell, C., Uria, B., Pritzel, A., Li, Y., Ruderman, A., Leibo, J. Z., Rae, J., Wierstra, D., and Hassabis, D. (2016). Model-free episodic control. *arXiv preprint arXiv:1606.04460*.

Bom, L., Henken, R., and Wiering, M. (2013). Reinforcement learning to train Ms. Pac-Man using higher-order action-relative inputs. In *Proceedings of IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning*.

Boytsov, L. and Naidan, B. (2013). Engineering efficient and effective non-metric space library. In *Similarity Search and Applications - 6th International Conference, SISAP 2013, A Coruña, Spain, October 2-4, 2013, Proceedings*, pages 280–293.

Braylan, A., Hollenbeck, M., Meyerson, E., and Miikkulainen, R. (2015). Frame skip is a powerful parameter for learning to play atari. In *AAAI-15 Workshop on Learning for General Competency in Video Games*.

Breiman, L. (2001). Random forests. *Mach. Learn.*, 45(1):5–32.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

Campbell, M., Hoane Jr, A. J., and Hsu, F.-h. (2002). Deep blue. *Artificial intelligence*, 134(1-2):57–83.

Cartwright, S. (1983). *Frostbite Bailey's Arctic Architect's Handbook*. Activision, Inc.

Dasgupta, S. and Freund, Y. (2008). Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 537–546. ACM.

Dong, W. (2017). kgraph. https://github.com/aaalgo/kgraph. version:3a870c926434cdb8ff8689417f37e885c2689f38.

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., et al. (2017). Noisy networks for exploration. *arXiv preprint arXiv:1706.10295.*

Fu, C. and Cai, D. (2016). EFANNA: An extremely fast approximate nearest neighbor search algorithm based on kNN graph. *arXiv preprint arXiv:1609.07228.*

Garcia, V., Debreuve, E., and Barlaud, M. (2008). Fast k nearest neighbor search using GPU. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–6. IEEE.

Gersho, A. and Shoham, Y. (1984). Hierarchical vector quantization of speech with dynamic codebook allocation. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'84.*, volume 9, pages 416–419. IEEE.

Geva, S. (2000). K-tree: a height balanced tree structured vector quantizer. In *Neural Networks for Signal Processing X, 2000. Proceedings of the 2000 IEEE Signal Processing Society Workshop*, volume 1, pages 271–280.

Hardt, O., Nader, K., and Nadel, L. (2013). Decay happens: the role of active forgetting in memory. *Trends in cognitive sciences*, 17(3):111–120.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298.*

Jegou, H., Douze, M., and Schmid, C. (2011). Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128.

Johnson, J., Douze, M., and Jégou, H. (2017). Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734.*

Johnson, W. B. and Lindenstrauss, J. (1984). Extensions of Lipschitz mappings into a Hilbert space. *Contemporary mathematics*, 26(189-206):1.

Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

LeCun, Y., Bengio, Y., et al. (1995). Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995.

LeKander, M. (2017a). ECDC. https://github.com/MLLeKander/ECDC. version:591d62e1312c51e7226593fa2edcf904f681b47d.

LeKander, M. (2017b). VQTree. https://github.com/MLLeKander/VQTree. version:e1f57553a6d9bb132bdd2b9520d8f46d334c8b5d.

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

Liu, T., Moore, A. W., Yang, K., and Gray, A. G. (2005). An investigation of practical approximate nearest neighbor algorithms. In *Advances in neural information processing systems*, pages 825–832.

Loader, C. (1999). *Local Regression and Likelihood*. Springer Science & Business Media.

Losing, V., Hammer, B., and Wersing, H. (2016). KNN classifier with self adjusting memory for heterogeneous concept drift. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, pages 291–300.

Malkov, Y., Ponomarenko, A., Logvinov, A., and Krylov, V. (2014). Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68.

Malkov, Y. A. and Yashunin, D. (2016). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *arXiv preprint arXiv:1603.09320*.

Mathy, C., Derbinsky, N., Bento, J., Rosenthal, J., and Yedidia, J. S. (2015). The boundary forest algorithm for online supervised and unsupervised learning. In *AAAI*, pages 2864–2870.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529.

Muja, M. (2017). FLANN. https://github.com/mariusmuja/flann. version:06a49513138009d19a1f4e0ace67fbff13270c69.

Nister, D. and Stewenius, H. (2006). Scalable recognition with a vocabulary tree. In *Computer vision and pattern recognition, 2006 IEEE computer society conference on*, volume 2, pages 2161–2168. Ieee.

Pritzel, A., Uria, B., Srinivasan, S., Badia, A. P., Vinyals, O., Hassabis, D., Wierstra, D., and Blundell, C. (2017). Neural episodic control. In *International Conference on Machine Learning*, pages 2827–2836.

Ram, P. and Gray, A. (2013). Which space partitioning tree to use for search? In *Advances in Neural Information Processing Systems*, pages 656–664.

Sabatelli, M., Louppe, G., Geurts, P., and Wiering, M. A. (2018). Deep quality-value (DQV) learning. *arXiv preprint arXiv:1810.00368*.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

Shaw, C. (1982). *River Raid Plan of Operation*. Activision, Inc.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *Nature*, 550(7676):354.

Singh, S. P. and Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1):123–158.

Sutton, R. S. and Barto, A. G. (1999). Reinforcement learning. *Journal of Cognitive Neuroscience*, 11(1):126–134.

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68.

Tibshirani, R. (2014). Course notes for Advanced Methods for Data Analysis: Kernel regression. http://www.stat.cmu.edu/~ryantibs/advmethods/notes/kernel.pdf. Accessed 2017-07-25.

Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31.

Tord Romstad, Marco Costalba, J. K. (2018). Home - stockfish - open source chess engine. https://stockfishchess.org/.

van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double Q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 2094–2100. AAAI Press.

Verma, N., Kpotufe, S., and Dasgupta, S. (2009). Which spatial partition trees are adaptive to intrinsic dimension? In *Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence*, pages 565–574. AUAI Press.

Vishwanath, M. and Chou, P. (1997). Video image compression using weighted wavelet hierarchical vector quantization. US Patent 5,602,589.

Wang, J., Shen, H. T., Song, J., and Ji, J. (2014). Hashing for similarity search: A survey. *arXiv preprint arXiv:1408.2927*.

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and De Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning-Volume 48*, pages 1995–2003. JMLR. org.

Wei, L.-Y. and Levoy, M. (2000). Fast texture synthesis using tree-structured vector quantization. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 479–488. ACM Press/Addison-Wesley Publishing Co.

Wiering, M. and van Otterlo, M. (2012). *Reinforcement Learning: State of the Art*. Springer.

Appendix A

Arcade Learning Environment Game Descriptions

A.1 Ms. Pacman

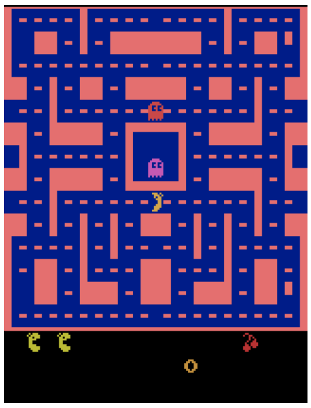


Figure A.1: In-game screenshot from the Atari 2600 version of Ms. Pacman.

In Ms. Pacman, the player controls the eponymous character, shown in yellow in the middle of the screen in Figure A.1. Directly above Ms. Pacman is the starting area for four “ghosts” which chase after Ms. Pacman as she moves through the maze. The player can control Ms. Pacman only by moving in one of the four cardinal directions. It is impossible for Ms. Pacman to stay still: if the player does not provide input, then Ms. Pacman continues in the most-recently given direction.

The player has 3 spare lives, and loses a life once a ghost catches Ms. Pacman. The game ends if the player loses all their lives.

Spread throughout the level are dots and pellets which must be collected before advancing to the next level. These items are collected by moving Ms. Pacman on top of the item. Collecting a dot gives an immediate reward of 10 points. Collecting a pellet gives an immediate reward of 50 points, and also causes the ghosts to become temporarily vulnerable. When Ms. Pacman collects the first, second, third, and fourth ghosts during this period, the player is given a reward of 200, 400, 800, and 1600 points, respectively. Finally, food items such as cherries, strawberries, and pretzels will intermittently appear in the maze for a limited amount of time. Collecting these items gives a substantial reward ranging between 100 and 5000 points, depending on the level.

No immediate reward is given for advancing levels, and no immediate punishment is given for losing a life.

A.2 Space Invaders

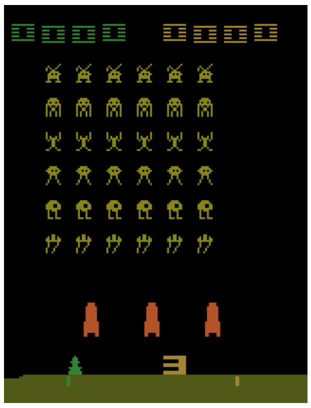


Figure A.2: In-game screenshot from the Atari 2600 version of Space Invaders.

In Space Invaders, the player controls a spaceship located at the bottom of the screen in Figure A.2. The top half of the screen contains "invaders" which slowly advance towards the bottom of the screen while firing shots towards the player. The player can move the ship left and right, but not vertically. The player can also fire a slowly-moving shot towards the invaders, but only one shot from the player may be on the screen at a time.

The player has 3 spare lives, and loses a life once hit by an invader's shot. The game ends if the player loses all their lives or if the invaders reach the bottom layer of the screen.

Each level begins with a set of barriers which take damage when hit by an invader's shot, providing temporary cover for the player. Once the invaders reach the same level as the barriers, the barriers disappear. Once all invaders are cleared, the player advances to the next level, which resets the barriers.

The player is given an immediate reward for each shot that contacts an invader. Invaders in the bottom row gives a reward of 5 points, and this reward increases by 5 for each layer. Thus, hitting an invader the top (sixth) row is worth 30 points. A flying saucer will intermittently fly across the very top of the screen. Hitting this saucer will give a large immediate reward of 200 points. No immediate reward is given for advancing levels, and no immediate punishment is given for losing a life.

A.3 Qbert

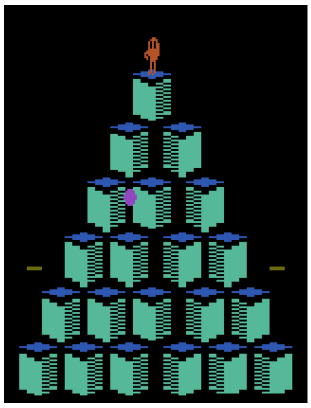


Figure A.3: In-game screenshot from the Atari 2600 version of Qbert.

In Qbert, the player controls the eponymous character, shown in red at the top of the screen in Figure A.3. Qbert is situated in a pyramid, and can move in one of the four diagonal directions. Each horizontal tile in the pyramid is a colored square which changes when Qbert jumps on it. The goal of the game is to turn each tile in the level to some goal color.

Enemies intermittently appear on the pyramid. Some enemies, such as the green ball and Sam, can be defeated by occupying the same square as them. However, other enemies, such as the red and purple balls, cannot be defeated in this manner. Indeed, coming into contact with these enemies damages the player. Thus, the agent should be able to distinguish between the differently-colored balls to maximize its points.

Once a purple ball reaches the bottom of the pyramid, it becomes a spring-like creature known as "Coily". This is the only enemy which can jump up from a lower level to a higher one. It is possible to perform a maneuver which causes Coily to jump off the pyramid, giving a large reward. This maneuver requires somewhat precise timing after a relatively long waiting period and requires an elevator platform (which disappears after being used), making it somewhat unlikely that an agent will stumble upon this strategy.

The player loses one life if the player character comes into contact with a damaging enemy or falls off of the pyramid. The player starts with 3 spare lives, and the game ends if the player loses all their lives.

The player is given the following rewards for performing each of the following actions:

- 25 points for changing a block to the goal color.
- 100 points for catching the green ball.
- 300 points for catching Sam.
- 500 points for luring Coily off the pyramid.
- 3100 points for completing the level (turning all blocks to the goal color).

The reward for completing the level is not immediate and instead is given in an animation during which the player's input has no impact. After completing the level, the player is given a reward of 100 every 5 frames for 31 iterations. No other rewards are given in this animated fashion and are given as a single lump sum.

No immediate punishment is given for losing a life.

A.4 Frostbite

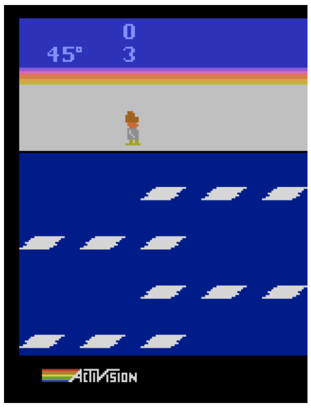


Figure A.4: In-game screenshot from the Atari 2600 version of Frostbite.

In Frostbite, the player controls a bear character (known as Frostbite Bailey) who is initially located at the top of the screen, as seen in Figure A.4. Below the starting location is a water area, across which sheets of ice drift horizontally in four rows. The player can jump on these rows of ice sheets to collect an ice block to construct an igloo. Once 16 ice blocks are collected, the igloo is fully constructed and the player can enter it to advance to the next level.

The player may move horizontally or jump between rows by moving vertically. It is also possible to switch the direction of the ice sheet the player character is currently standing on, at the cost of one previously-collected ice block. Since this sheet-switching action directly detracts from the ultimate goal of completing the level, the optimal strategy is to use this only very rarely. Indeed, the official Frostbite manual advises the player to "learn to use the [sheet-flipping action] sparingly" (Cartwright, 1983).

There are various obstacles (e.g. Polar Grizzlies, Killer Clams, and Snow Geese) which appear in some levels. However, Green Fish, which the player can collect for a small reward, sometimes appear in the same location as some of these obstacles. Thus, a high-scoring agent should learn to distinguish between obstacles and rewards.

The top of the screen contains a temperature monitor, which starts at 45 degrees and drops by one degree every second. The player loses one life if the player character comes into contact with an obstacle, falls into the water, or if the temperature reaches zero (i.e. 45 seconds elapse). The player starts with 3 spare lives, and the game ends if the player loses all their lives.

On the first level, an immediate reward of 10 points is given for collecting an ice block, and an immediate reward of 160 points is given for entering the igloo (and thus completing the level). These two reward sources are both affected by a level-based reward multiplier. This multiplier begins a 1, and each level increases the reward multiplier by 1, up to a maximum of 9. Thus, collecting an ice block gives a reward of 10 points on level 1, but 90 points on levels 9 and above.

The player is also given an immediate reward for each degree (second) remaining when the player enters the igloo. For each degree remaining, the player is given degree $\times$ level points. This level multiplier is unbounded, unlike the multiplier for collecting ice blocks or entering the igloo.

An immediate reward of 200 points is given for collecting a Green Fish. This reward is the same for all levels.

It is worth noting that the igloo, degree, and fish rewards are not actually given in a single lump sum. When these are awarded, an animation is played during which the player's inputs have no impact. When a fish is collected, the screen freezes and the player is given a reward of 10 points every 1.33 frames. When the igloo is entered, every 11 frames the player is given a reward of $10 \times \max(\text{level}, 9)$ points and one ice block is removed from the igloo. Once this animation is completed, the degree animation begins, giving a reward of $10 \times$ level points every 3 frames until the temperature counter reaches 0.

No immediate punishment is given for losing a life.

A.5 River Raid

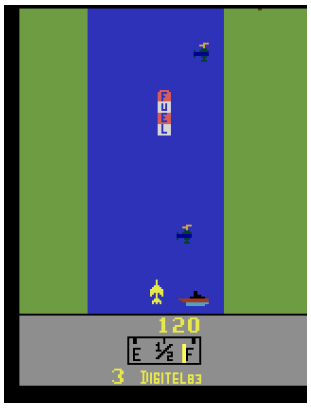


Figure A.5: In-game screenshot from the Atari 2600 version of River Raid.

In River Raid, the player controls a fighter jet, shown in yellow at the bottom of Figure A.5. The player must navigate their jet across a river canyon, destroying or otherwise avoiding obstacles long the way. The player can control their jet by moving left or right, by increasing or decreasing the jet's speed, or by firing missiles.

The river canyon contains a variety of objects, most of which are enemies and damage the player jet upon collision. The enemy objects include Tankers, Helicopters, Jets, and Bridges. The only destructible object which does not damage the jet are Fuel Depots, which refuel the player's jet on contact.

It is important to note that although it is necessary to refuel to progress far into the game, no immediate reward is given for refueling. In fact, a small reward is actually given for destroying Fuel Depots. Thus, an agent must learn to sometimes not target these objects in spite of them offering a short-term reward. Indeed, the official River Raid manual advises the player to "concentrate on flying to the next fuel depot, and don't try to destroy every object" (Shaw, 1982).

The player loses one life if the player character comes into contact with an enemy object, crashes into the surrounding walls, or runs out of fuel. The player starts with 3 spare lives, and the game ends if the player loses all their lives.

Points are only gained by destroying objects in the river. Destroying each type of object gives the following rewards:

- Tanker for 30 points,
- Helicopter for 60 points,
- Fuel Depot for 80 points,
- Jet for 100 points,
- and Bridge for 500 points.

When the player runs into an enemy object, both the player's plane as well as the enemy object are destroyed. Thus, it is sometimes possible to gain points while losing a life.

Appendix B

Approximate k-Nearest Neighbors Algorithms

Algorithm 4 Brute Force Search

```
1: function bruteForceSearch($D$, $\boldsymbol{q}$, $k$)
2:     NBR $\leftarrow$ max-heap of size $k$
3:     for $i \leftarrow 0, 1, 2, \ldots, N-1$ do
4:         NBR.add($d(\boldsymbol{q}, \boldsymbol{D_i})$, $\boldsymbol{D_i}$)
5:     return NBR.values()
```

Algorithm 5 LSH Construction

```
1: function lshBuild($D$, hash)
2:     table $\leftarrow$ hash-table mapping hashes to sets of points
3:     for $i \leftarrow 0, 1, 2, \ldots, N-1$ do
4:         table.insert(hash($\boldsymbol{D_i}$), $\boldsymbol{D_i}$)
5:     return table
```

Algorithm 6 LSH Search

```
1: function lshSearch(table, $\boldsymbol{q}$, hash)
2:     NBR $\leftarrow$ max-heap of size $k$
3:     for $\boldsymbol{D_i} \leftarrow$ table.get(hash($\boldsymbol{q}$)) do
4:         NBR.add($d(\boldsymbol{q}, \boldsymbol{D_i})$, $\boldsymbol{D_i}$)
5:     for nearbyHash $\leftarrow$ nearby(hash($\boldsymbol{q}$)) do        $\triangleright$ e.g. random mutation
6:         for $\boldsymbol{D_i} \leftarrow$ table.get(nearbyHash) do
7:             NBR.add($d(\boldsymbol{q}, \boldsymbol{D_i})$, $\boldsymbol{D_i}$)
8:     return NBR.values()
```

Algorithm 7 Neighbor Graph Search

```
function NeighborGraphSearch(graph, $\boldsymbol{q}$, initial$\boldsymbol{D}$s)
    NBR ← max-heap of size $k$
    visited ← set of datapoints
    frontier ← min-heap
    for $\boldsymbol{D_i}$ ← initial$\boldsymbol{D}$s do
        frontier.append(d($\boldsymbol{q}$, $\boldsymbol{D_i}$), $\boldsymbol{D_i}$)
        visited.add($\boldsymbol{D_i}$)
    while NBR.size() $<$ $k$ or NBR.maxKey() $>$ frontier.minKey() do
        $\boldsymbol{c}$ ← frontier.removeMin()
        NBR.add(d($\boldsymbol{q}$, $\boldsymbol{c}$), $\boldsymbol{c}$)
        for $\boldsymbol{D_i}$ ← graph.neighborsOf($\boldsymbol{c}$) do
            if not visited.contains($\boldsymbol{D_i}$) then
                frontier.append(d($\boldsymbol{q}$, $\boldsymbol{D_i}$), $\boldsymbol{D_i}$)
                visited.add($\boldsymbol{D_i}$)
    return NBR.values()
```

Algorithm 8 k-d Tree Construction

```
function kdTreeBuild($D$, $maxLeafSize$)
    function kdTreeNode($D$, axis)
        if $D$.size() $\leq$ $maxLeafSize$ then
            node.isLeaf ← True
            node.data ← $D$
        else
            node.isLeaf ← False
            node.split ← median of $\{\boldsymbol{D_i}[\text{axis}] \mid \boldsymbol{D_i} \in D\}$
            left$D$ ← $\{\boldsymbol{D_i} \mid \boldsymbol{D_i} \in D \wedge \boldsymbol{D_i}[\text{axis}] < \text{node.split}\}$
            right$D$ ← $D$ − left$D$
            node.lChild ← kdTreeNode(left$D$, axis + 1 mod $d$)
            node.rChild ← kdTreeNode(right$D$, axis + 1 mod $d$)
        return node
    return kdTreeNode($D$, 0)
```

Algorithm 9 k-d Tree Search

```
function kdTreeSearch(rootNode, $\boldsymbol{q}$, $\varepsilon$)
    NBR ← max-heap of size $k$
    function kdTreeSearchNode(node, axis)
        if node.isLeaf() then
            for $\boldsymbol{D_i}$ ← node.data do
                NBR.add($d(\boldsymbol{q}, \boldsymbol{D_i})$, $\boldsymbol{D_i}$)
        else
            aNode ← ($\boldsymbol{q}$[axis] < node.split) ? node.lChild : node.rChild
            bNode ← ($\boldsymbol{q}$[axis] < node.split) ? node.rChild : node.lChild
            kdTreeSearchNode(aNode, axis + 1 mod $d$)
            splitDist ← $\big|\boldsymbol{q}$[axis] − node.split$\big|$
            maxNBRDist ← NBR.maxKey() $* (1 + \varepsilon)$
            if NBR.size() $< k \lor$ maxNBRDist > splitDist then
                kdTreeSearchNode(bNode, axis + 1 mod $d$)
    kdTreeSearchNode(rootNode, 0)
    return NBR.values()
```

Algorithm 10 RP Tree Construction

```
function rpTreeBuild($D$, $\mathit{maxLeafSize}$)
    function rpTreeNode($D$)
        if $D$.size() $\leq$ $\mathit{maxLeafSize}$ then
            node.isLeaf ← True
            node.data ← $D$
        else
            node.isLeaf ← False
            node.direction ← random unit direction $\in \mathbb{R}^d$
            node.bias ← median of $\{\boldsymbol{D_i} \cdot$ node.direction $\mid \boldsymbol{D_i} \in D\}$
            left$D$ ← $\{\boldsymbol{D_i} \mid \boldsymbol{D_i} \in D \land \boldsymbol{D_i} \cdot$ node.direction > node.bias$\}$
            right$D$ ← $D$ − left$D$
            node.lChild ← rpTreeNode(left$D$)
            node.rChild ← rpTreeNode(right$D$)
        return node
    return rpTreeNode($D$)
```

Algorithm 11 Mean Tree Construction

```
function meanTreeBuild($D$, $maxLeafSize$, $p$)
    function meanTreeNode($D$)
        if $D$.size() $\leq$ $maxLeafSize$ then
            node.isLeaf $\leftarrow$ True
            node.data $\leftarrow$ $D$
        else
            node.isLeaf $\leftarrow$ False
            node.centers $\leftarrow$ list of $p$ cluster centers, as computed by k-means
            for $j \leftarrow 0, 1, 2, \ldots, p-1$ do
                sub$D$ $\leftarrow$ $\{\boldsymbol{D_i} \mid j == \text{argmin}_k\, d(\text{node.centers}[k], \boldsymbol{D_i})\}$
                node.children[$j$] $\leftarrow$ meanTreeNode(sub$D$)
        return node
    return meanTreeNode($D$)
```

Algorithm 12 Mean Tree Search

```
function meanTreeSearch(rootNode, $\boldsymbol{q}$, $\varepsilon$)
    NBR $\leftarrow$ max-heap of size $k$
    function meanTreeSearchNode(node)
        if node.isLeaf() then
            for $\boldsymbol{D_i}$ $\leftarrow$ node.data do
                NBR.add($d(\boldsymbol{q}, \boldsymbol{D_i})$, $\boldsymbol{D_i}$)
        else
            sort node.centers and node.children by distance from $\boldsymbol{q}$
            meanTreeSearchNode(node.children[0])
            for $j \leftarrow 1, 2, \ldots, N-1$ do
                split $\leftarrow$ planeBetween(node.centers[0], node.centers[j])
                maxNBRDist $\leftarrow$ NBR.maxKey() $*$ $(1+\varepsilon)$
                if NBR.size() $== k \wedge$ maxNBRDist $< d(\boldsymbol{q}, \text{split})$ then
                    break
                meanTreeSearchNode(node.children[i])
    meanTreeSearchNode(rootNode, 0)
    return NBR.values()
```

Algorithm 13 Online Mean Tree Add

```
function onlineMeanTreeAdd(node, $\boldsymbol{D_i}$)
    if node.isLeaf then
        node.data.append($\boldsymbol{D_i}$)
        if node.data.size() > $\mathit{maxLeafSize}$ then
            node.isLeaf ← False
            node.centers ← list of $p$ cluster centers, as computed by k-means
            for $j \leftarrow 0, 1, 2, \ldots, p-1$ do
                child ← new node
                child.isLeaf ← True
                child.data ← $\{\boldsymbol{D_i} \mid j == \text{argmin}_k\, d(\text{node.centers}[k], \boldsymbol{D_i})\}$
                node.children[$j$] ← child
    else
        $k \leftarrow \text{argmin}_k\, d(\text{node.centers}[k], \boldsymbol{D_i})$
        node.centers[$k$].append($\boldsymbol{D_i}$)
        onlineMeanTreeAdd(node.children[$k$], $\boldsymbol{D_i}$)
    return node
```

Algorithm 14 K-tree Add

```
function ktreeAdd(root, $\boldsymbol{D_i}$)                     ▷ returns resulting root node
    node ← root
    while !node.isLeaf() do                           ▷ find leaf closest to $\boldsymbol{D_i}$
        node.center.add($\boldsymbol{D_i}$)
        node ← $\text{argmin}_{\text{child}\in\text{node.children}}\, d(\text{child.center}, \boldsymbol{D_i})$
    node.center.add($\boldsymbol{D_i}$)
    node.data.append($\boldsymbol{D_i}$)
    if node.data.size() > $\mathit{maxLeafSize}$ then
        splitA, splitB ← 2Means(node.data)              ▷ k-means with k=2
        ktreeRemoveChild(node.parent, node)
        ktreeAddChild(node.parent, splitA, root)
        return ktreeAddChild(node.parent, splitB, root)
    return root

function ktreeAddChild(node, newChild, root)
    node.children.append(newChild)
    if node.children.size() > $\mathit{branchFactor}$ then
        splitA, splitB ← 2Means(node.children)
        if node != root then
            ktreeRemoveChild(node.parent, node)
            ktreeAddChild(node.parent, splitA, root)
            return ktreeAddChild(node.parent, splitB, root)
        else
            newRoot ← new node
            newRoot.children.append(splitA)
            newRoot.children.append(splitB)
            return newRoot
    return root
```

Algorithm 15 2Means Splitting

```
function 2MEANS($d$)
    centerA ← data.center
    centerB ← data.center
    labels ← list of length $d$.size, filled with -1
    data.shuffle()
    flag ← True
    while flag do
        flag ← False
        for $i \leftarrow 0, 1, 2, \ldots, d.\text{size}() - 1$ do
            newLabel ← $d(\text{centerA}, \boldsymbol{D_i}) < d(\text{centerB}, \boldsymbol{D_i})$ ? 0 : 1
            if labels[$i$] != newLabel then
                if labels[$i$] == 0 then
                    centerB.remove($\boldsymbol{D_i}$)
                else if labels[$i$] == 1 then
                    centerA.remove($\boldsymbol{D_i}$)
                (newLabel == 0 ? centerA : centerB).add($\boldsymbol{D_i}$)
                flag ← True
            labels[$i$] ← newLabel
    splitA ← $\{\boldsymbol{D_i} \mid \text{labels}[i] == 0\}$
    splitB ← $\{\boldsymbol{D_i} \mid \text{labels}[i] == 1\}$
    return splitA, splitB
```

Algorithm 16 Forest Add and Forest Search

```
function FORESTADD(forest, $\boldsymbol{D_i}$)
    for tree ← forest do
        tree ← TREEADD(tree, $\boldsymbol{D_i}$)

function FORESTSEARCH(forest, $\boldsymbol{D_i}$)
    NBR ← min-heap of size $k$
    for tree ← forest do TREESEARCH(tree, $\boldsymbol{D_i}$, NBR)
```

Algorithm 17 Greedy Search (for Mean Trees)

```
function GREEDYSEARCH(rootNode, $\boldsymbol{q}$)
    NBR ← max-heap of size $k$
    while !node.isLeaf() do
        node ← $\text{argmin}_{\text{child} \in \text{node.children}}\, d(\text{child.center}, \boldsymbol{D_i})$
    for $\boldsymbol{D_i}$ ← node.data do
        NBR.add($d(\boldsymbol{q}, \boldsymbol{D_i})$, $\boldsymbol{D_i}$)
    return NBR.values()
```

Algorithm 18 Prototype Distance Search

```
1: function PROTOTYPEDISTSEARCH(rootNode, $\boldsymbol{q}$, $minLeaves$)
2:     NBR $\leftarrow$ max-heap of size $k$
3:     frontier $\leftarrow$ max-heap of size $minLeaves$
4:     frontier.add(0, rootNode)
5:     for $j \leftarrow 1, 2, \ldots, minLeaves$ do
6:         node $\leftarrow$ frontier.removeMin()
7:         while !node.isLeaf() do
8:             closest $\leftarrow$ $\text{argmin}_{\text{child}\in\text{node.children}}\, d(\boldsymbol{q}, \text{child.center})$
9:             for child $\leftarrow$ node.children $\setminus$ {closest} do
10:                frontier.add($d(\boldsymbol{q}, \text{child.center})$, child)
11:            node $\leftarrow$ closest
12:        for $\boldsymbol{D_i}$ $\leftarrow$ node.data do
13:            NBR.add($d(\boldsymbol{q}, \boldsymbol{D_i})$, $\boldsymbol{D_i}$)
14:    return NBR.values()
```

Algorithm 19 Plane Distance Search

```
1: function PLANEDISTSEARCH(rootNode, $\boldsymbol{q}$, $minLeaves$)
2:     NBR $\leftarrow$ max-heap of size $k$
3:     frontier $\leftarrow$ max-heap of size $minLeaves$
4:     frontier.add(0, rootNode)
5:     for $j \leftarrow 1, 2, \ldots, minLeaves$ do
6:         node $\leftarrow$ frontier.removeMin()
7:         while !node.isLeaf() do
8:             closest $\leftarrow$ $\text{argmin}_{\text{child}\in\text{node.children}}\, d(\boldsymbol{q}, \text{child.center})$
9:             for child $\leftarrow$ node.children $\setminus$ {closest} do
10:                plane $\leftarrow$ planeBetween(closest, child)
11:                frontier.add($d(\boldsymbol{q}, \text{plane})$, child)
12:            node $\leftarrow$ closest
13:        for $\boldsymbol{D_i}$ $\leftarrow$ node.data do
14:            NBR.add($d(\boldsymbol{q}, \boldsymbol{D_i})$, $\boldsymbol{D_i}$)
15:    return NBR.values()
```

Algorithm 20 Leaf Graph Search

```
function LEAFGRAPHSEARCH(rootNode, leafLookup, $\boldsymbol{q}$, $minLeaves$)
    NBR ← max-heap of size $k$
    leafNode ← rootNode
    while !leafNode.isLeaf() do
        leafNode ← $\text{argmin}_{\text{child}\in\text{leafNode.children}}\, d(\boldsymbol{q}, \text{child.center})$
    frontier ← max-heap of size $minLeaves$
    visited ← empty set
    frontier.add(0, node)
    for $j \leftarrow 1, 2, \ldots, minLeaves$ do
        node ← frontier.removeMin()
        for $\boldsymbol{D_i}$ ← node.data do
            NBR.add($d(\boldsymbol{q}, \boldsymbol{D_i})$, $\boldsymbol{D_i}$)
            for neighbor ← leafLookup[$\boldsymbol{D_i}$] do
                if neighbor $\notin$ visited then
                    visited.add(neighbor)
                    frontier.add($d(\boldsymbol{q}$, neighbor), neighbor)
    return NBR.values()
```