# IMPROVING CACHE REPLACEMENT POLICY USING DEEP REINFORCEMENT LEARNING

BY

CHRISTOPHER BENSON

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Assistant Professor Jian Peng

## ABSTRACT

This thesis explores the use of reinforcement learning approaches to improve replacement policies of caches. In today's internet, caches play a vital role in improving performance of data transfers and load speeds. From video streaming to information retrieval from databases, caches allow applications to function more quickly and efficiently. A cache's replacement policy plays a major role in determining the cache's effectiveness and performance. The replacement policy is an algorithm that chooses which piece of data in the cache should be evicted when the cache becomes full and new elements are requested. In computer systems today, most caches use simple heuristic-based policies. Currently used policies are effective but are still far from optimal. Using more optimal cache replacement policies could dramatically improve internet performance and reduce database costs for many industry-based companies.

This research examines learning more optimal replacement policies using reinforcement learning. In reinforcement learning, an agent learns to take optimal actions given information about an environment and a reward signal. In this work, deep reinforcement learning algorithms are trained to learn optimal cache replacement policies using a simulated cache environment and database access traces. This research presents the idea of using index-based cache access histories as input data for the reinforcement learning algorithms instead of content-based input. Several approaches are explored including value-based algorithms and policy gradient algorithms. The work presented here also explores the idea of using imitation learning algorithms to mimic optimal cache replacement policies. The algorithms are tested on several different cache sizes and data access patterns to show that these learned policies can outperform currently used replacement policies in a variety of settings.

## ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Jian Peng, for helping me throughout my graduate studies and research. During my time working for Professor Peng, he always provided invaluable research ideas, useful guidance for areas to explore, and encouragement for my work. I am tremendously thankful for all the patience and time he put into helping me throughout the research process and the completion of this thesis.

## TABLE OF CONTENTS

# CHAPTER 1: INTRODUCTION

Caches are a critical part of many aspects of today's computer systems. Caches allow faster access to data and can generally improve performance of computer systems. A wide variety of applications that are critical to people all around the world use caching to improve their functionality. Caches are used to store saved versions of web pages which allows for faster access and less bandwidth usage. Video services, such as YouTube or Netflix, improve video streaming quality by caching videos in distributed databases. Caches are vital to the functionality of the internet and improving database access speeds. Better cache performance would dramatically improve many applications and have significant impact throughout the world.

More specifically, a cache is an intermediate store of data between an application using the data and the original source of the data. A cache typically allows faster retrieval of data but is also typically smaller than the original store of data. If an application needs a piece of data that is already stored in a cache, the application can simply retrieve the data from the cache and never has to interact with the original data source. However, if the piece of needed data is not in the cache, a cache miss occurs, and the application must retrieve the data from the original data source. A diagram depicting a standard cache setup is shown in Figure 3.1. Due to the limited size of caches, it is critical for a cache to hold relevant pieces of data that will be needed in the future. When a cache becomes full and wants to store a new piece of data, it must decide which pieces of data in the cache to evict. The cache decides which items to evict based on the cache's replacement policy. The replacement policy is an algorithm that decides which elements to remove based on the cache's state and past data accesses. Typically, the replacement policy is based off simple heuristic functions such as eliminating the least recently used piece of data in the cache. These heuristics are effective but are not the optimal replacement policy. Recent results in machine learning, suggest that these replacement policies could be improved by using algorithms to learn more optimal policies.

Machine learning has recently had several break throughs in the areas of reinforcement learning and deep learning. Reinforcement learning is a subsection of machine learning where

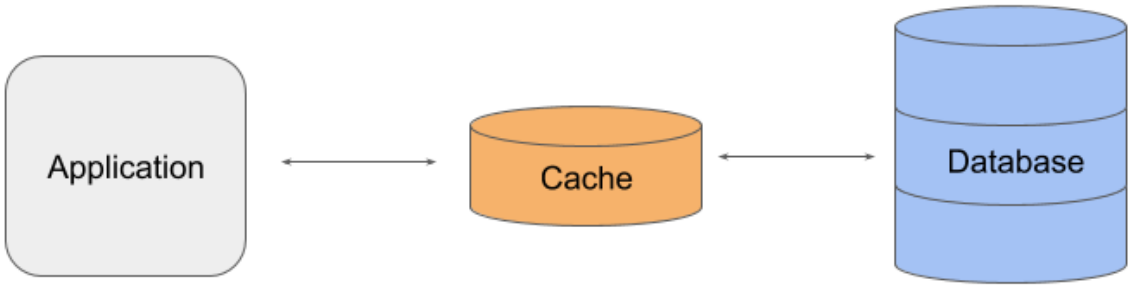



Figure 1.1: Example cache setting. Arrows represent data requests and transfers.

agents learn to behave optimally in environments in order to maximize rewards [1]. Deep learning uses artificial neural networks with many layers to learn to approximate complex functions. Deep learning has succeeded in impressive tasks such as classifying images [2]. In the last few years, reinforcement learning and deep learning have been combined into deep reinforcement learning. Deep reinforcement learn has enabled computers to be able to learn complex tasks such as playing Atari video games [3]. Recent work of deep reinforcement learning has investigated its application to learning replacement policies for caches.

The work presented in this thesis explores a new approach to applying deep reinforcement learning to cache replacement policies. The problem of cache replacement policy is modeled as a Markov decision process where the learning agent must decide which pieces of data should be evicted from the cache. A novel part of the research presented here is the use of index-based information for cache state representation instead of content-based information. Previous approaches using deep reinforcement learning for cache replacement policies, looked at the content of the items in the cache to determine which elements to remove. The approach presented here uses a new cache state representation that is purely based on cache index access histories.

Using index-based cache states, agents are trained to learn a replacement policy to minimize the number of cache misses that occur over a set of data accesses. Recent state-of-the-art algorithms are used to solve this problem. Reinforcement learning methods such as the value-based Double Deep Q-learning algorithm and the policy-based Advantage Actor-Critic algorithm are explored. This thesis also presents new work on using imitation learning to train cache replacement policies to mimic an optimal replacement algorithm. The optimal Bélády's replacement algorithm is used as an expert agent that always achieves the fewest possible cache misses over a set of data accesses. Using supervised learning and Generative Adversarial Imitation Learning, agents are trained to attempt to mimic the actions taken by this optimal algorithm. The agents trained using imitation learning are also used to initialize the policies of reinforcement learning algorithms.

The approaches in this work are evaluated using a simulated cache and database access patterns. For experimentation, a simplified cache model is simulated on a series of data reads. The data access patterns are generated following a Zipf distribution to attempt to approximate real data access patterns. The algorithms are tested on several different cache sizes and data distributions. The performances of these algorithms are evaluated based on cache misses and compared to standard baseline algorithms.

The results of this thesis show that reinforcement learning algorithms are able to learn better cache replacement policies than the standard replacement methods in a variety of cache settings. Additionally, the results show that the algorithms tend to perform better in more complex cache settings which suggests that the approaches described here could be successfully extended to more complicated real-world caching systems.

# CHAPTER 2: BACKGROUND

The proceeding chapter will provide background information on caching and reinforcement learning. This should help to clarify concepts and topics that will be explored in more detail during later chapters of the thesis presented here.

## 2.1 CACHING

As described in the introduction, caches act as intermediate stores of data that can improve performance and data transfer speeds. Caches improve performance by storing data that is used multiple times so that applications can access this reused data faster than if it were retrieved from the original data source. Caches can provide faster access times because they are typically not as large as original data sources and only store a subset of all data available. Caches are effective when they can keep relevant data in the smaller data store for future work. The cache replacement policy decides which pieces of data are kept in the cache. This work focuses on evaluating and learning better cache replacement policies to allow more relevant data to be present in the cache. Caches are quite complicated and can follow several different designs, but this work will focus on simple caches that are effective for evaluating cache replacement policies.

### 2.1.1 Evaluation Metrics

A cache improves performance when data requested by the application is already stored in the cache. When a cache becomes full and a new piece of data is requested, certain elements of the cache are evicted by the replacement policy, and the new data is put in the cache. Following a good replacement policy, caches are able to keep useful pieces of data in the cache and evict other irrelevant pieces of data. A cache hit occurs when a newly requested piece of data is currently stored in the cache. A cache miss occurs when a newly requested piece of data is not present in the cache. Ideally, all requested data will always be in the cache, but this is not possible in practice. Therefore, to measure a cache replacement policy's performance, cache hit ratio is used. A cache's hit ratio is the number of cache hits divided

by the total number of accesses [4, 5, 6].

$$\text{Hit Ratio} = \frac{\text{Cache Hits}}{\text{Data Accesses}} = \frac{\text{Cache Hits}}{\text{Cache Hits} + \text{Cache Misses}} \tag{2.1}$$

This is the main metric that will be used to compare replacement policies. With a higher hit ratio, more cache hits are occurring leading to faster data access and better application performance. Hit ratio is typically examined with respect to a set of data access. A set of data access refers to the order in which several pieces of data were requested from a database or cache. In this work, a set of data accesses will also be referred to as an access trace or data access pattern. There are many other methods to evaluate cache performance, however hit ratio is the most direct metric to evaluate performance of the cache replacement policy.

### 2.1.2 Baseline Replacement Policies

Currently, caches use a number of heuristic based replacement policies. There are a wide range of replacement policies used, but two of the most common are the least recently used policy and least frequently used policy.

Least Recently Used (LRU) policy removes the item in the cache that was least recently accessed in the cache [7]. The cache keeps track of the order that the items in the cache were last accessed. When it needs to purge an element from the cache, the policy removes the oldest item in the cache according to access history.

Least Frequently Used (LFU) policy removes the item that has been accessed least frequently while in the cache [7]. The cache keeps track of the number of times each piece of data has been accessed and will purge the item that had been accessed the fewest number of times. If an item is removed from the cache and then placed back into the cache, its access count is reset.

Another baseline replacement policy to consider is a truly random replacement policy. When an item from the cache must be evicted, a random policy will choose an element to evict at random. This baseline will provide a lower bound for how the cache should perform with no knowledge or information.

### 2.1.3 Bélády's Algorithm

In addition to considering baseline algorithms, this work also looks at optimal cache replacement policies such as Bélády's algorithm or the clairvoyant algorithm [8]. Bélády's algorithm guarantees optimal cache hit ratio for a given trace of data accesses. However, it is not feasible to implement in the practice because it requires knowledge of future data accesses. Because it requires future knowledge of data accesses, it is sometimes referred to as the clairvoyant algorithm. The algorithm works by removing the element in the cache that will be accessed furthest ahead in time based on future data accesses. This has been proven to be optimal but is not possible to use in real caching systems [8]. However, in simulated data traces, it provides a useful upper bound to consider how well a replacement policy can behave.

## 2.2 REINFORCEMENT LEARNING

Reinforcement learning is an area of machine learning that focuses on training agents to act in unknown environments based on given reward signals. An agent must learn to maximize cumulative rewards from the environment by repeatedly choosing an action to take. The problem becomes an repeated decision making challenge. The agent observes the current state of the environment and then takes an action. After taking the action, the agent receives a reward and a new state from the environment. This process repeats as shown in Figure 2.2 until some terminating event. The agent must learn to take the best actions such as to maximize the reward received from the environment at each step.

To formalize the environment for reinforcement learning, Markov Decision Processes (MDPs) are commonly applied to provide a mathematical framework to describe the situation[1]. MDPs are a method to formalize decision making processes where an agent repeatedly interacts with an environment and learns to behave such as to maximize reward. In an MDP, the agent must repeatedly decide on actions to take. An MDP is defined by the set of states the environment can be in $S$, the set of actions the agent can take $A$, the transition function $P(s'|s,a) : S \times S \times A \to \mathbb{R}$ where $s, s' \in S$ and $a \in A$, the reward function $R(s,a) : S \times A \to \mathbb{R}$ where $s \in S$ and $a \in A$, and a discount factor $\gamma$ where $0 < \gamma \leq 1$.

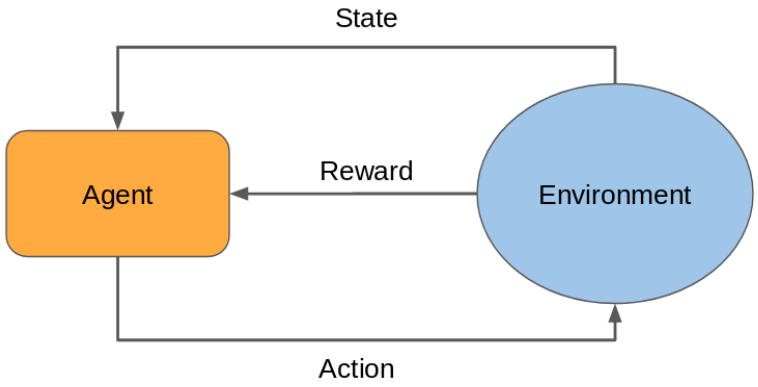



Figure 2.1: Example reinforcement learning environment

The transition function models the probability of transitioning from one state to another state given that the agent took some action. The reward function returns a scalar reward value for taking an action in the state of the environment. The discount factor $\gamma$ is used to scale summations of rewards so that rewards do not go to infinity over long time periods. An important property of MDPs are that they follow the Markov Property. The Markov Property implies that the environment's future states purely depend on the current state. States prior to the current state have no impact upon future states or rewards if the current state is known. This assumption is typically true, but sometimes the assumption is violated.

To clarify MDPs further consider the situation described below. For timesteps, t = 0,1,2,..., the agent observes a state $s_t \in S$ from the environment, then takes an action $a_t \in A$, receives a reward for taking this action $r_t \in R$, and then ends up in the next state $s_{t+1}$ based on the transition probability. This can be written as series of events:

$$s_0, a_0, r_0, s_1, a_1, r_1, ... \tag{2.2}$$

The goal of the agent is to maximize cumulative reward of this series of actions or $\sum_t r_t$. To do this, the agent must learn a policy function $\pi(s) : S \rightarrow A$ which tells the agent an action to take given a state of the environment. This function is also sometimes written as $\pi(a|s) : S \times A \rightarrow \mathbb{R}$ which defines a function that tells the agent the probability it should

take action $a$ given it is in state $s$.

$$\pi(a|s) = Pr(action = a|state = s) \tag{2.3}$$

However, solving for this policy function be challenging for a few reasons. Typically, some parts of the MDP are not fully known. The transition function $P(s'|s,a)$ is typically stochastic and not fully defined. After taking an action $a$ in state $s$, it is not always known which state $s'$ the environment will end up in. This will typically follow a random process where the probabilities are not known. Additionally, the reward function $R(s,a)$ can be a partially known or a stochastic function. The agent does not know what reward it will get from the environment when it takes an action. These unknown parts of the MDP makes solving for the optimal policy challenging. It is impossible to directly derive the optimal policy since these parts of the MDP are not known. Instead methods must be used to attempt to learn the optimal policy by interacting with the environment.

Solving a stochastic MDP can be broken into two different categories. The first category is model-based approaches. These approaches attempt to directly learn the transition function $P(s'|s,a)$ and reward function $R(s,a)$ to solve the MDP. The other category is model-free approaches which do not attempt to learn a model of the environment and try to directly find a policy to follow in the unknown environment. In this work, model-free algorithms will be explored because they are more directly applicable to the problem being solved.

In model-free based approaches, a policy is learned to achieve as much cumulative reward as possible. The value function of a policy determines how much reward will be achieved following that policy for all future timesteps. This can be formally written as $V^\pi$:

$$V^\pi(s_t) = E_\pi \left[ \sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s \right] \tag{2.4}$$

$V^\pi(s)$ represents the expected reward from following policy $\pi$ starting in state $s$. The goal of reinforcement learning is to find a policy that achieves the highest possible value function for all states.

To learn this optimal policy that maximizes the cumulative reward, there are two com-

monly used approaches. Policy based approaches directly optimize the policy function. Value based approaches learn a form of the value function and base the policy function of the value estimates. Additionally, there is a third method that uses a combination of policy and value functions to behave optimally. These algorithms are called actor-critic methods.

A popular set of policy-based approaches are policy gradient methods. These methods attempt to create a parametric approximation of the policy function $\pi(a|s)$. This function approximation can be written as $\pi(a|s;\theta)$ where $\theta$ is the parameters of the function approximator. Given a policy function $\pi(a|s;\theta)$, policy gradient methods attempt to optimize this policy by performing gradient ascent updates to the functions parameters $\theta$ so as to maximize the expected reward [9]. The basic update rule for these algorithms is shown in Equation 2.5 where $J$ is the estimated expected value following the policy approximated with $\theta$. The policy function $\pi(a|s;\theta)$ is commonly approximated using a deep neural network who's parameter weights are $\theta$.

$$\theta \leftarrow \theta + \nabla_\theta J(\theta) \tag{2.5}$$

A popular set of value-based algorithms is the family of Q-learning algorithms and extensions. These algorithms attempt to approximate the function $Q^\pi$ which can be defined as:

$$Q^\pi(s,a) = E_\pi \left[ \sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s, a_t = a \right] \tag{2.6}$$

$Q^\pi(s,a)$ represents the expected reward of taking action $a$ in state $s$ and then following policy $\pi$ until the end of the episode. Deep Q-learning algorithms attempt to do this by using a neural network as a function approximator for the Q function. The algorithms attempt to approximate $Q^\pi(s,a)$ as $Q^\pi(s,a;\theta)$ where the function is approximated using a deep neural network with parameters $\theta$. If $Q^\pi$ is known, it is easy to formulate a policy based on this:

$$\pi(s) = argmax_{a'} Q^\pi(s,a') \tag{2.7}$$

A third category of algorithms are actor-critic algorithms. These algorithms approximate both the $Q(s,a)$ function and the policy function $\pi(a|s)$. This provides the benefit of directly learning the policy while also using the estimated value of each action to determine how to

update the policy [10]. This improves on the learning approach of policy gradient by reducing variance in the updates to the parameters.

Another topic related to reinforcement learning is imitation learning. Imitation learning is where an agent attempts to learn to mimic the actions of an expert agent. This is different from reinforcement learning in that the agent does not receive any reward from the environment and instead simply tries to clone the behavior of an expert. Initially learning from an expert agent can provide a good method for initializing the parameters of an agent using reinforcement learning. It is typically easier for an agent to initially learn to mimic an expert than to learn its own optimal policy directly.

This section has given some background into the ideas of reinforcement learning. The specific algorithms used in this work will be expanded in greater detail in later sections.

# CHAPTER 3: PROBLEM FORMULATION FOR CACHE REPLACEMENT

The chapter presented below will detail the approach used in thesis to model cache replacement policy as a reinforcement learning problem. The chapter will also describe the MDP formalizing the problem.

## 3.1 CACHE REPLACEMENT POLICY AS A MARKOV DECISION PROCESS

The problem of cache replacement policy can be modeled as a Markov Decision Process (MDP). In this research, the agent of the MDP is modeled as the algorithm that decides which element of the cache should be evicted when the cache becomes full.

The problem of cache replacement is well suited for being treated as a Markov Decision Process. In any cache setting, a series of data items are requested from the original data source. As these pieces of data are requested, some of the items are stored in the cache for future accesses. However, once the cache becomes full, some items must be evicted. The cache repeatedly must choose which item to remove from the cache when it is full and a new request for data occurs. This type of repeated decision process is exactly what MDPs are designed to mathematically model. The agent in this Markov decision process can be thought of as an internal agent of the cache that chooses which elements to keep and which elements to evict. This setting is depicted in Figure 3.1.

Another import aspect for a problem to be well suited for modeling as a Markov Decision Process is that the Markov Assumption is true. The current state of the cache is based only off the elements in the cache. The previous items in the cache do not impact the state of the cache. Additionally, the past data pieces that were evicted by the cache play no role in current state of the cache. Therefore, for cache replacement policy, the Markov property holds. This indicates again that this problem is clearly well made for being an MDP.

The state space $S$ of the MDP is the possible set of the states the cache can be in. In this work the cache's state is recorded as a history of cache index accesses. This will be explained in more detail in the next section.

The action space $A$ is the set of indices in the cache that can be evicted. For example, if

the cache was of size 3, the action space would be size 3 because each of the elements in the cache could be evicted. The action space is equal to the size of the cache. Each index of the cache can be evicted.

The reward function $R(s, a)$ is defined to discourage cache misses. The agent takes an action whenever the cache is full, and a cache miss has occurred. It only chooses an action when an element needs to be evicted from the cache. This implies that a cache miss has occurred. Therefore, the reward when a cache miss occurs is always -1. If the cache reaches the end of the data accesses and there is no miss, the reward is 0. The reward function is formally written in Equation 3.1.

$$R(s_t, a_t) = \begin{cases} -1 & \text{if cache miss occurs at } s_{t+1} \\ 0 & \text{if } s_t \text{ is terminal state} \end{cases} \tag{3.1}$$

This reward function makes the agent attempt to reduce the number of cache misses. By attempting to reduce the number of cache misses, the agent is attempting to increase the number of cache hits and improve the cache hit ratio.

The transition function $P(s'|s, a)$ for this Markov decision process is stochastic and unknown. The transition from one cache state to the next depends on the next pieces of data that are requested in the data access pattern. When a piece of data is evicted from the cache, the next piece of data takes the spot in the cache and then a series of new data requests occurs. The state depends on the next data requested and used. This cannot be known ahead of time and therefore means that this MDP is not fully known.

The discount factor $\gamma$ in this MDP is chosen to be close to 1. The series of data accesses tends to be long and each reward is not overly dependent on each individual action taken. Choosing $\gamma$ to have a higher value makes the model consider long-term cumulative rewards more important when compared to short-term rewards. In this problem, maximizing long term hit ratio is most important and therefore choosing a high $\gamma$ is best.

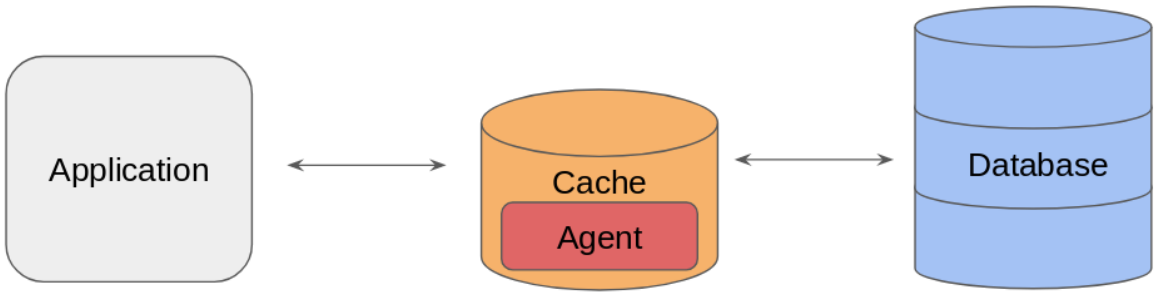



Figure 3.1: Cache setup for modeling cache replacement policy as a Markov Decision Process and using an agent to solve the problem.

## 3.2 INDEX BASED STATE REPRESENTATION

Related work that models cache replacement as a reinforcement learning problem, use a content-based state representation. In contrast, this work uses an index-based state representation of the cache. An index-based cache state representation means that the state of the cache is represented as the history of cache access for each index in the cache. The state representation does not care about what content is in the cache. The only thing considered for creating the cache state is which index of the cache is accessed at each timestep.

The cache state is represented by a two-dimensional matrix. The matrix is of shape cache size by history length. Cache size is the number of elements that fit inside of the cache. History length is the number of timesteps in the past that the access pattern of the cache is stored. The history length can be adjusted and changed. Each row in the matrix represents the access history for the element at that cache index. Each column represents which index of the cache was accessed at that timestep and if the element at that index was in the cache at that timestep.

More formally, the state space $S$ is in the set of matrices with shape cache size by history length. If $C$ is defined as the number of elements that can fit in the cache, and $H$ is defined

as the history length, the state space can be written as:

$$S \subset \mathbb{R}^{C \times H} \tag{3.2}$$

For a specific state $s_t \in S$ where $0 \leq i < C$ and $0 \leq h < H$, each location of the matrix is defined as described in Equation 3.3.

$$s_{t_{i,h}} = \begin{cases} 1 & \text{if element i accessed at timestep (t-h)} \\ x & \text{if element i not in cache at timestep (t-h)} \\ 0 & \text{Otherwise} \end{cases} \tag{3.3}$$

In Equation 3.3, x is a flag value to differentiate an element being in the cache but not being accessed from an element not being in the cache at this time.

For a concrete example of this, consider a basic cache of size 3 using a history length of 4. The cache will begin empty and will have the access pattern from left to right off

$$d_m, d_y, d_y, d_z, d_m... \tag{3.4}$$

Each $d$ represents a piece of data being accessed. Data accesses with the same subscript imply that the same piece of data is being accessed again. The state at $s_4$, or after the second access of $d_m$, would be

$$s_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & x & x \end{bmatrix} \tag{3.5}$$

The first row corresponds to the accesses of element $d_m$ and has a 1 in the first column because $d_m$ is accessed at timestep 4. The second row corresponds to the accesses of element $d_y$. The third row corresponds to accesses of element $d_z$ and has an $x$ in the final two columns because element $d_z$ was not in the cache at those times.

When an element in the cache is removed, the row that corresponded to that element of that cache is replaced with the default value for the entire row. This row is then updated to show that the most recent access is of the new element.

For example, consider a continuation of the previous example where the next access in the trace in Equation 3.4 is a new piece of data $d_n$. This new data element is not in the cache and one of the elements must be evicted. If $d_m$ at index 0 is evicted and $d_n$ is placed into that index, the new state $s_5$ becomes the value shown in Equation 3.6. The second and third rows are shifted to the right by one from $s_4$ based on the passing of one timestep.

$$s_5 = \begin{bmatrix} 1 & x & x & x \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & x \end{bmatrix} \tag{3.6}$$

This state representation is an expressive representation of state that gives the reinforcement learning algorithms plenty of information to use for deciding which action to take. Using this state representation, it is simple to approximate either the LRU or LFU directly. To follow an LFU based policy, simply remove the index in the cache whose row has minimum sum. This will be the item with the least accesses in the cache history. To follow an LRU based policy, remove the row whose index of the first occurrence of a 1 is furthest to the right. Both functions can be easily learned and approximated using deep neural networks. This implies that reinforcement learning should be able to at least learn policies comparable to these baselines and the set of algorithms that are combinations of these policies [11]. However, the neural networks will also be able to learn some sort of combination of these approaches that could be more effective than LRU or LFU. Access patterns and correlations between indices can be used more effectively than with other state representations. Overall, the state provides plenty of information for the algorithms to learn good policies.

Now that the state definition is clarified, the MDP is fully-defined. Next, this work considers the algorithms for solving this problem and finding an optimal policy.

# CHAPTER 4: REINFORCEMENT LEARNING ALGORITHMS

In the previous chapter, the problem of cache replacement policy was defined as a Markov Decision Process. In this chapter, the process for solving this MDP will be examined in detail. In this research, several different reinforcement learning algorithms were applied to solving the problem of cache replacement policy. The reasons for using reinforcement learning and the details of the algorithms will be presented below.

## 4.1 APPLICABILITY OF REINFORCEMENT LEARNING

The problem of cache replacement policy is an almost perfect problem to be solve with Reinforcement learning. First, the problem can be modeled as an MDP which is described in the previous chapter. The MDP for cache replacement policy is defined to be partially known. This means that most of the MDP is known, but the transition function $P(s'|s, a)$ is stochastic and not fully known. Because this is unknown, direct optimization techniques cannot be applied. Instead a policy must be learned by repeatedly interacting with the environment. This is the exact type of problem reinforcement learning was designed to solve.

Secondly, cache replacement policy can be simulated and run many times. Current state-of-the-art reinforcement learning algorithms are still quite sample inefficient. This means that they require seeing interacting with an environment a large number of times before they are able to achieve good policies [1]. Current state-of-the-art algorithms are only able to perform well after millions of games [12, 13]. For the cache problem, data is accessed through caches or databases millions of times per day throughout the internet. There are many data access traces with thousands of data requests that exist in the world that can be used to simulate a cache problem. It is even possible to create purely synthetic data that mimics the access patterns of real data. All of this allows reinforcement learning algorithms to interact with the cache environment enough to be effective at learning policies. This ability to run many training steps allows state-of-the-art reinforcement learning approaches to work in this setting.

The following sections describe the reinforcement learning algorithms used in this thesis. The work examines value-based methods such as Double Deep Q-learning. It examines versions of policy gradient algorithms such as Advantage Actor-Critic. The work also explores initializing these reinforcement learning approaches by using imitation learning such as Generative Adversarial Imitation Learning.

## 4.2 DOUBLE DEEP Q-LEARNING

Double Deep Q-Learning (DDQN) is a value-based reinforcement learning algorithm that uses the Q function as defined in Equation 2.6. The policy of the algorithm is based off of the values returned by the Q function such that $\pi(s) = \text{argmax}_a Q(s, a)$. The algorithm uses a deep neural network to approximate the Q function. This is defined as $Q(s, a; \theta)$ where the weights of the neural network are represented by the parameters $\theta$. If the Q function approximation provides accurate value predictions for all state-action pairs, then the algorithm will behave optimally. For Q-learning based algorithm to behave optimally, the Bellman equation described in Equation 4.1 should hold true for sequences following $s, a, r, s'$.

$$Q(s, a) = E\left[r + \gamma \text{max}_{a'} Q(s', a') | s, a\right] \tag{4.1}$$

Because of this property of optimal Q functions, the neural network is trained to make Equation 4.1 as close to true as possible. To do this, the neural network is updated using stochastic gradient descent on batches of sequences $(s, a, r, s')$ to minimize the loss value in Equation 4.2. The sequences are sampled from a set of stored past experiences gathered form interacting with the environment.

$$L(\theta) = E\left[((r + \gamma \hat{Q}(s', \text{max}_{a'} Q(s', a'; \theta); \theta^-)) - Q(s, a; \theta))^2\right] \tag{4.2}$$

The function $\hat{Q}(s, a; \theta^-)$ is called the target network for the algorithm and is used for stabilizing training. This network is initialized to have the same parameters as the main Q function. The main Q network is updated each step using gradient descent and then after a constant number of step the target network is set to be equal to the main network again or

$\theta^- = \theta$

In this work, the Double Deep Q-Learning algorithm is used to predict the values of removing each item in the current cache state. The input to the neural network is the state of the cache based on Equations 3.2 and 3.3. The output of the neural network is a vector of shape $\mathbb{R}^C$ where C is the size of the cache. Each index of the output vector corresponds to the value of removing that index from the cache given its current state. The full pseudo-code for this equation can be seen in Algorithm 4.1.

This algorithm could be effective for solving the cache replacement policy because it has been proven to effectively work with large state spaces [14]. Previous versions of Q-learning struggled with large state spaces because it was hard to get an effective method to approximate the Q function. However, deep neural networks have proven to be effective at doing this. In this work, the state space is a reasonably large two-dimensional matrix. This is comparable to some of the problems where deep Q-learning has proved to be effective. Deep Q-learning has been effective taking images in directly as states. An image is similar to the cache state history as it is again simply a large 2D matrix. Using the 2D matrix state representation as input, the Deep Q-learning should be able to effectively find a good policy.

**Algorithm 4.1** Double Deep Q-Learning with Experience Replay

Initialize experience replay $D$ with capacity $N$
Initialize action-value function $Q$ with random parameters $\theta$
Initialize target action-value function $\hat{Q}$ with random parameters $\theta^- = \theta$
**for** episode = 1,M **do**
  Initialize $s_0$ to start state of cache
  **for** t = 1,T **do**
    With probability $\epsilon$ select a random action $a_t$
    Otherwise select $a_t = \text{argmax}_a Q(s_t, a; \theta)$
    Execute action $a_t$ and observe reward $r_t$ and $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $D$
    Sample random minibatch of transitions $(s_j, a_j, r_j, s_{j+1})$ in $D$
    $y_j = \begin{cases} r_j & \text{if } s_{j+1} \text{ is terminal} \\ r_j + \gamma \hat{Q}(s_{j+1}, \max_{a'} Q(s_{t+1}, a'; \theta); \theta^-) & \text{if } s_{j+1} \text{ is non-terminal} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_j, a_j, ; \theta))^2$ with respect to parameters $\theta$
    Every C steps set $\hat{Q} = Q$
  **end for**
**end for**

## 4.3 ADVANTAGE ACTOR-CRITIC

Advantage Actor-Critic (A2C) is a policy gradient based reinforcement learning algorithm [15]. This policy gradient algorithm is modeled as an actor-critic method meaning that it directly optimizes the policy function, but it also approximates the value function. In this work, a deep neural network is used to approximate the policy function defined in Equation 2.3. This neural network is defined as $\pi(a|s;\theta_\pi)$ where $\theta_\pi$ is the weights of the neural network. The algorithm also approximates the value function as defined in Equation 2.4 using a deep neural network defined as $V(s;\theta_v)$.

To train the policy function, the parameters $\theta_\pi$ are updated using gradient descent to directly maximize the expected reward $E[R]$ from using those parameters. $E[R_t]$ is defined as the expected reward after timestep t as $R_t = \sum_{i=t}^{T} r_i$. Standard policy gradient algorithms update the parameters $\theta_\pi$ based on the value $\nabla_{\theta_\pi} \log(\pi(a_t|s_t;\theta_\pi)R_t$ which acts as an estimate for $\nabla_{\theta_\pi} E[R_t]$. This estimate is unbiased but tends to have high variance. To reduce variance, the approximated value function $V(s;\theta_v)$ is used as a baseline. The value $(R_i - V(s_i;\theta_v))$ is the estimated advantage of taking action $a_i$ in terms of received reward compared predicted value. to The final gradient update used is shown in Equation 4.3.

$$\theta_\pi \leftarrow \theta_\pi + \alpha \nabla_{\theta_\pi} \log \pi(a_t|s_t;\theta_\pi)(R_t - V(s_t;\theta_v)) \tag{4.3}$$

Similar to Double Deep Q-learning, the value network is trained to minimize the mean squared loss described in Equation 4.4.

$$L(\theta_v) = (R_t - V(s_t;\theta_v))^2 \tag{4.4}$$

In this work, the policy network predicts the probability of which index of the cache should be evicted given the current state of the cache. The pseudo-code for this A2C is formally written in Algorithm 4.2.

The A2C algorithm should be able to excel in the MDP defined in this work. Policy gradients tend to be able to perform quite well in areas involving large action spaces [16]. The algorithm is directly learning to approximate the $\pi(a|s;\theta_\pi)$ function. This is different from

the previously seen Q-learning algorithm. By directly learning this function, the algorithm should be able to perform better in larger action spaces. This can be quite useful here as the action space for this problem is the size of the cache. The size of the cache can get quite large and therefore it would be beneficial to use an algorithm that can handle such a large action size. Similar to Deep Q-learning, A2C has proved to be able to handle large 2D input matrices as state inputs [15]. While there are some more sample efficient algorithms, A2C provides close to state-of-the-art performance for policy gradient algorithms.

**Algorithm 4.2** Advantage Actor-Critic (A2C)

Initialize policy function $\pi(a|s;\theta_\pi)$ with random parameters $\theta_\pi$
Initialize value function $V(s;\theta_v)$ with random parameters $\theta_v$
Initialize T = 0
Initialize t = 0
**repeat**
  Reset gradient: $d\theta_\pi \leftarrow 0$
  Reset gradient: $d\theta_v \leftarrow 0$
  **for** episode = 1,K **do**
    $t_{start} = t$
    **repeat**
      Perform action $a_t$ according to policy $\pi(a|s;\theta_\pi)$
      Receive reward $r_t$ and new state $s_{t+1}$
      $t \leftarrow t + 1$
      $T \leftarrow T + 1$
    **until** $s_t$ is terminal or $t - t_{start} == t_{max}$
    $R = \begin{cases} 0 & \text{if } s_t \text{ is terminal} \\ V(s_t, \theta_v) & \text{if } s_t \text{ is non-terminal} \end{cases}$
    **for** $i \in \{t-1, ..., t_{start}\}$ **do**
      $R \leftarrow r_i + \gamma R$
      Accumulate gradients wrt $\theta_\pi$: $d\theta_\pi \leftarrow d\theta_\pi + \nabla_{\theta_\pi} \log \pi(a_i|s_i;\theta_\pi)(R - V(s_i;\theta_v))$
      Accumulate gradients wrt $\theta_v$: $d\theta_v \leftarrow d\theta_v + \frac{\partial (R - V(s_i;\theta_v))^2}{\partial \theta_v}$
    **end for**
  **end for**
  Perform update of $\theta_\pi$ using $d\theta_\pi$
  Perform update of $\theta_v$ using $d\theta_v$
**until** T > $T_{max}$

## 4.4 IMITATION LEARNING FOR INITIALIZATION

In addition to simply using reinforcement learning, imitation learning was explored as a way to initialize the neural networks prior to starting to train using reinforcement learning. Imitation learning allows agents to learn to mimic an expert agent. By mimicking the expert agent, the learning agent can achieve good performance that ideally is comparable to the expert. Imitation learning can be applied to the cache replacement problem because the perfect expert action can always be known if the entire trace of future data accesses is known. Future data accesses are known in simulated environments when training the algorithms. In the problem of cache replacement policy, an agent following Bélády's algorithm, as defined in Section 2.1.3, will always perform optimally. The issue is that Bélády's algorithm is not possible in practice because Bélády's algorithm requires knowledge of future cache accesses. However, imitation learning can be used to train an agent to mimic the actions of Bélády's algorithm on a training set of data accesses with the hope that it can perform similar to it on a new set of data accesses.

A simple form of imitation learning is supervised learning using expert state-action pairs. The policy network $\pi(a|s;\theta_\pi)$ of A2C can be initialized by training it to correctly predict the action the expert would take. The expert algorithm is run for a set of training examples. The state-action pairs produced by this are saved. The policy network is then trained using supervised learning and stochastic gradient descent to make the network predict the action that the expert would take given the state. This is done by training the network to minimize the log cross entropy loss described in Equation 4.5 [17]. In this equation, $y_i$ is the true label that the expert action is action i and $p_i$ is the predicted probability that the expert action is action i.

$$L(\theta_\pi) = -\sum y_i \log(p_i) \tag{4.5}$$

After training using this supervised learning approach, the weights learned in this method can be transfered to the policy network of the A2C algorithm. The normal A2C algorithm can then be run to additionally train the agent.

Generative Adversarial Imitation Learning (GAIL) is a state-of-the-art imitation learning algorithm [18]. GAIL trains an agent to mimic expert actions using the ideas from Generative

Adversarial Networks (GAN) [19]. A discriminator function $D(s, a)$ predicts if an action is an expert action or the learning agent's action. The discriminator is approximated with $D(s, a; \theta_D)$. The goal is to have the discriminator be unable to distinguish expert actions from agent actions. This discriminator is used to provide a reward to update the policy function. If the discriminator thinks the actions are expert actions, the policy is updated to take those actions more frequently. The discriminator is updated using binary cross entropy loss and supervised learning of batches of state-action pairs. The original GAIL paper updates the policy using a Trust Region Policy Optimization (TRPO) step [18, 15]. However, more recent work has shown that using Proximal Policy Optimization (PPO) is more effective and simpler to implement [20]. In this work, the GAIL algorithm is run using PPO to update the agent's policy and is run using an actor-critic approach. PPO is similar to A2C in that it is a policy gradient algorithm, but it instead does a proximal policy gradient update. The update uses the ratio function defined in Equation 4.6.

$$r_t(\theta_\pi) = \frac{\pi(a_t|s_t; \theta_\pi)}{\pi(a_t|s_t; \theta_{\pi_{old}})} \tag{4.6}$$

This ratio is then used to update the policy using the clipped policy gradient estimate defined by Equation 4.7.

$$\min(r_t(\theta_\pi)A_t, clip(r_t(\theta_\pi), 1-\epsilon, 1+\epsilon)A_t) \tag{4.7}$$

In this equation, $A_t = \log(D(s_t, a_t; \theta_D)) - V(s_t; \theta_v^-)$ which estimates the advantage for taking action $a_t$. The equation here does not use any reward but is simply based off the prediction of discriminator. The clip function keeps the ratio value between the other two constant values for stability. The value function $V(s; \theta_v)$ is updated using the mean squared error as described for the A2C algorithm. The entire pseudo-code for the algorithm is written in Algorithm 4.3.

In this thesis, Bélády's algorithm is run on a train set of data. The algorithm stores the states and optimal actions taken. GAIL is run to train the agent to mimic the behavior of the optimal algorithm on new data access traces. The learned policy and value networks can then be used as an agent to behave in the cache environment. By itself, this could be used as

**Algorithm 4.3** Generative Adversarial Imitation Learning (GAIL)

**Input:** Stored trajectories from expert policy: $\tau_E \sim \pi_E$
Initialize discriminator function $D(s, a; \theta_D)$ with random parameters $\theta_D$
Initialize policy function $\pi(a|s; \theta_\pi)$ with random parameters $\theta_\pi$
Initialize target policy function $\pi(a|s; \theta_\pi^-)$ with parameters $\theta_\pi^- = \theta_\pi$
Initialize value function $V(s; \theta_v)$ with random parameters $\theta_v$
Initialize target value function $V(s; \theta_v^-)$ with parameters $\theta_v^- = \theta_v$
**for** m = 1,Episodes **do**
  Sample trajectories from current policy $\tau_\pi \sim \pi(a|s; \theta_\pi)$
  Sample trajectories from expert policy $\tau_E \sim \pi_E$
  Update the discriminator $D(s, a; \theta_D)$ parameters $\theta_D$ following the gradient:

$$E_{\tau_\pi} \left[\nabla_{\theta_D} \log(D(s, a; \theta_D))\right] + E_{\tau_E} \left[\nabla_{\theta_D} \log(1 - D(s, a; \theta_D))\right]$$

  **for** n = 1, EpisodesPPO **do**
    $t_{start} = t$
    **repeat**
      Perform action $a_t$ according to policy $\pi(a|s; \theta_\pi)$
      Receive new state $s_{t+1}$
      $t \leftarrow t + 1$
    **until** $s_t$ is terminal or $t - t_{start} == t_{max}$
    **for** $j = t_{start}, t$ **do**
      Compute $r_j(\theta_\pi) = \frac{\pi(a_j|s_j;\theta_\pi)}{\pi(a_j|s_j;\theta_\pi^-)}$
      Accumulate gradient for j with respect to $\theta_\pi$ into $d\theta_\pi$ using the gradient step:

$$\min(r_j(\theta_\pi)A_j, clip(r_j(\theta_\pi), 1 - \epsilon, 1 + \epsilon)A_j)$$

      Where $A_j = \log(D(s_j, a_j; \theta_D)) - V(s_j; \theta_v^-)$
      Accumulate gradient for j with respect to $\theta_v$ into $d\theta_v$ using the loss:

$$(\log(D(s_j, a_j; \theta_D) - V(s_j; \theta_v))^2$$

    **end for**
    Perform update of $\theta_\pi$ using $d\theta_\pi$
    Perform update of $\theta_v$ using $d\theta_v$
  **end for**
  $\theta_\pi^- \leftarrow \theta_\pi$
  $\theta_v^- \leftarrow \theta_v$
**end for**

an agent. An additional step taken in this work is to have this trained agent be used as the starting point for the A2C algorithm. This is done by taking the neural networks trained in the GAIL setting and initializing the weights of the A2C networks to be these values. This is different because the A2C networks are typically randomly initialized as described in Algorithm 4.2. This provides a good initialization to be improved upon by A2C and can be thought of as helping the algorithm find a good initial policy.

The GAIL algorithm should be effective here because it has been used successfully in problems with large action spaces and complex input states. Many of the initial use cases of the GAIL algorithm was working in continuous action spaces. Since it can work in continuous action spaces, GAIL should be effective working in large discrete action spaces. With abundant expert trajectories and large action space, the cache replacement policy problem seems like an ideal problem to initially attempt to solve by mimicking Bélády's algorithm.

# CHAPTER 5: EXPERIMENTAL SETTINGS

The following chapter will detail the experiments run to evaluate the methods described in the previous chapter. The details of the cache model, the training process, and the algorithm parameters will be explained.

## 5.1 CACHE MODEL

Cache environments can be quite complex. In real-world settings, caches must handle reading data and writing pieces of data from many different processes. Each piece of data tends to have different sizes and data elements are commonly broken up into smaller pieces. All of these factors can complicate cache performance. In the experiments of this thesis, a simplified cache model is used to make evaluating the performance of the cache replacement policy clearer.

The agents are run in a simplified simulated cache model. The cache model is a fixed size cache that only deals with reading data and does not deal with writing data. Each piece of data is assumed to have the same size. The cache size is therefore equal to the number of elements that can fit in the cache. The simulated cache is implemented in Python and uses a NumPy array for storage of data and history of information [21].

The cache state described in Equations 3.3 and Equation 3.2 is stored as a two-dimensional NumPy array that is updated every data access. For the $x$ placeholder values in the cache history, a small positive value was used. This would give a slight positive signal similar to when an item was accessed in the cache but would be far less important than an actual index access. For all experiments, the value $1/H$ where $H$ is the length of the cache history was used for $x$. It was also explored using negative values such as -1, but this led to worse performance.

In this simulation, each piece of data has the same size and takes up one index of the cache. For example, a cache of size 10 will be able to hold exactly 10 items in the cache. In this work, relatively small cache sizes are used for evaluation. The cache sizes evaluated are 25, 50, and 100. These cache sizes are small for real-world situations but allow for smaller

scale evaluation and faster experimental run times. For the cache history length, a variety of number were explored. History length of around 50 provided a good tradeoff between enough information and reasonable run time so that the state did not become too large. This is the number that was explored in experiments.

## 5.2 SIMULATED DATA

The cache is evaluated on simulated data access patterns. Each trace is an ordered list of data accesses. Each access represents the next piece of data requested from the database. Each piece of data is simply an integer number. The data is generated following a Zipf distribution. The Zipf distribution is defined following the probability density function shown in Equation 5.1 [22].

$$p(x) = \frac{x^{-\alpha}}{\zeta(\alpha)} \text{ where } \zeta(y) = \sum_{n=0}^{\infty} \frac{1}{n^y} \tag{5.1}$$

In the above equation, $\alpha$ is a parameter and $\zeta$ is the Riemann zeta function. This distribution results in the frequency of a data item being inversely-proportional to the rank of the item where ranking is based off how frequently it occurs. For example, the nth most common piece of data occurs with frequency proportional to $\frac{1}{n}$.

This is chosen because database access patterns tend to follow a Zipf-like distribution [23, 22]. While it is not the same as the Zipf Law, accesses do generally tend to follow a pattern similar to the Zipf distribution. In this work, the $\alpha$ parameters for the Zipf distribution is chosen to be 1.15 and 1.3. These two $\alpha$ values are chosen because they provide two different data distributions that are reasonably challenging for the cache model described.

## 5.3 TRAINING AND EVALUATION

The algorithms were trained on a set of 2000 cache traces each having 1000 data accesses in them. The algorithms learned on these 2000 training traces while being evaluated on 100 separate evaluation traces. Training continued until performance on the evaluation set

stopped improving. Finally, the performance was tested on a set of 1000 distinct testing traces. The algorithm's performance was then compared to baseline implementations of LFU, LRU, and random replacement policies using the same cache state information. For imitation learning algorithms, another set of 500 training traces was used to generate expert actions based on the optimal algorithm. This set of optimal state-action pairs were then used in the supervised learning or GAIL algorithms.

To run the training and testing process, this work made use of the Illinois Campus Cluster, a computing resource that is operated by the Illinois Campus Cluster Program (ICCP) in conjunction with the National Center for Supercomputing Applications (NCSA) and which is supported by funds from the University of Illinois at Urbana-Champaign.

## 5.4 NEURAL NETWORK ARCHITECTURES AND PARAMETERS

All the algorithms described in the chapter on algorithms use neural networks as function approximators. These algorithms are implemented using the python deep neural network library called Pytorch [24]. This library provides an easy way to implement neural networks for testing and provides methods for automatic backpropagation.

Most neural networks take the cache state matrix as input and output a value for each possible action. Given the input state matrix, a value is output for each index of the cache that indicates if that element of the cache should be removed. Initially, each row of the cache history was only used in predicting the value of removing the item at that row in the cache history. For example, the access history of the element at the first index is only used to predict if the element at the first index should be removed. This makes logical sense because the value of a row is only dependent on the access history of that row. The other rows of the access histories should not impact the value of that index. Therefore, networks using structures similar to the one shown in Figure 5.1 were initially used.

However, neural networks that used all the data in the cache state to predict all the values in the action space were also explored. While this might not be as intuitive, this provides the neural network with additional information and possible input combinations to perform better. This ended up leading to better performance and therefore architectures similar to

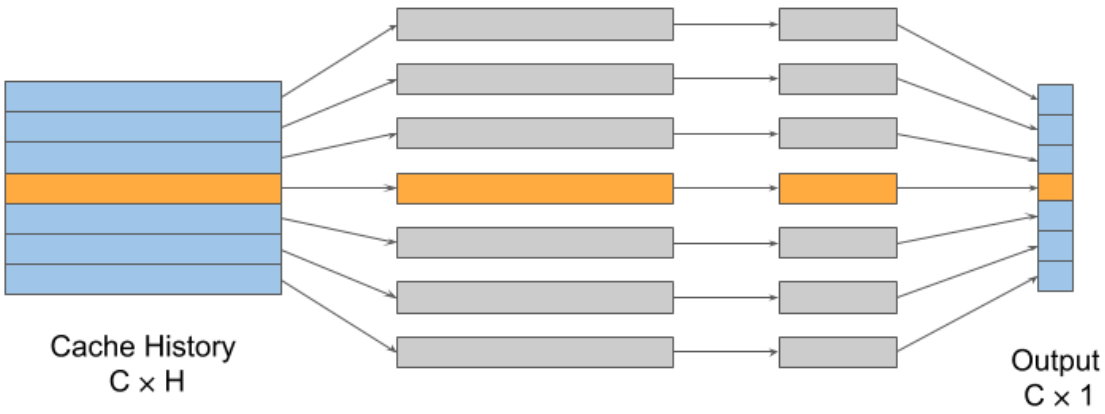



Figure 5.1: Neural network architecture that keeps each cache history separate. The cache history in the orange row is only used to predict the value from the action at that index in orange.

the one shown in Figure 5.2 were used for achieving the best performance.

Different neural network architectures were explored for all algorithms. The most successful networks tended to be simple feed forward networks. Convolutional networks were explored, but they did not perform as well.

The below subsections will detail the best network architectures used and the parameters used for training them for each algorithm.

### 5.4.1 Double Deep Q-Learning

The Double Deep Q-learning algorithm uses two neural networks that each estimate the value function $Q(s, a)$. The two networks are the current network and the target network. Both have the same architecture. The architecture used in this work is a simple feed forward network. The input to the network is the two dimension matrix of the cache state history as described in Equation 3.3. This input state is flattened to a vector of length $(CH)$ where $C$ is cache size and $H$ is history length. The network has 2 hidden fully connected layers each respectively with size $2CH$ and $CH$. All layers except for the output layer use the

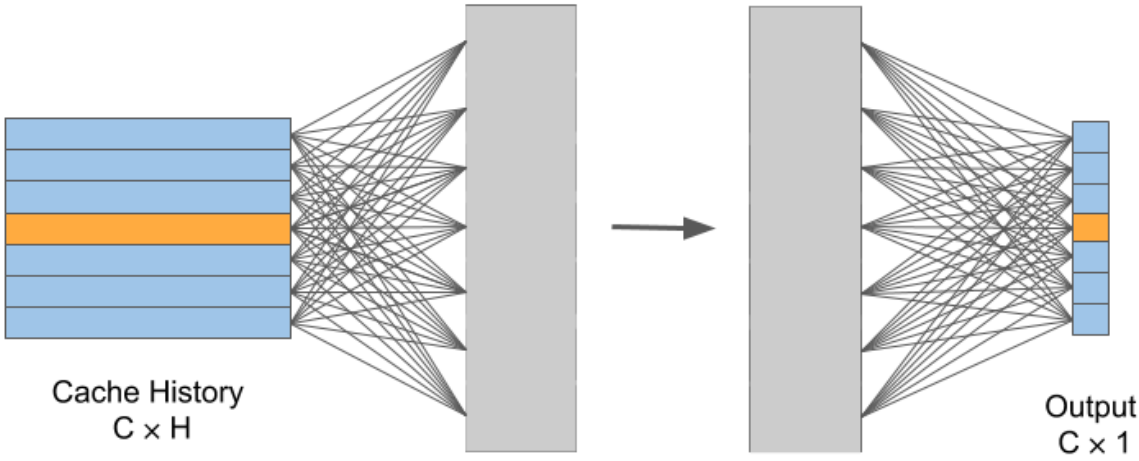



Figure 5.2: Neural network architecture that combines all cache histories. The value for the orange row is predicted using all the rows in the cache history including the blue rows.

ReLU activation function. The final output size is $C$ with one value for each possible action. The output value is the predicted value for $Q(s, a)$ for the given state and the corresponding index.

The replay memory for holding experience tuples is initialized to have size 10000. For the training algorithm, the neural network is updated using the Adam optimizer. The batch size for each update is 128. The learning rate starts at 0.001 and decreases in half every 200 episodes. The algorithm follows an $\epsilon$-greedy exploration policy with $\epsilon$ set to 0.05 to start and decreases in half every 200 episodes. The target network is updated every 10000 updates. $\gamma$ value of 0.99 was used.

### 5.4.2 A2C

A2C has two neural networks. One network estimates the policy function $\pi(a|s;\theta_\pi)$ and the other estimates the value function $V(s;\theta_v)$. The policy function takes a state value and outputs the probabilities of taking each action. The input is the flattened state of size $CH$. The network is a fully connected network with 3 hidden layers. The hidden layers have size $CH/4$, $CH/8$, and $CH/16$. The output layer has size $C$. The activation function of

all layers except for the output layer is ReLU. The final layer activation is a SoftMax layer to convert the logits into probabilities. The neural network for the value function takes a state as inputs and gives the value of being in that state or one scalar output. The neural network follows the same structure as that described in the policy function except the final output layer size is 1. It also does not use a SoftMax layer at the end because it should output a scalar value and not a probability. In some works, the value and policy networks are combined into one network with two separate output layers. In this work, two separate networks seemed to perform better and therefore separate networks were used.

The algorithm is run for until each data access trace is completed. After 4 access traces or episodes, the cumulative rewards are computed, and the value and policy networks are updated based on the algorithm rules. The Adam optimizer is used with learning rate starting at 0.05 and decreasing by half every 200 episodes. $\gamma$ is set to 0.99.

### 5.4.3 GAIL

GAIL uses 5 neural networks in the implementation in this work. The three functions to be approximated are the policy function $\pi(a|s;\theta_\pi)$, the value function $V(s;\theta_v)$, and the discriminator function $D(s,a;\theta_D)$. Both the policy and value functions also have target functions so there are 5 total networks. The policy and value functions follow the same format as the A2C algorithm.

The discriminator is different. The discriminator takes the state of the cache and an action as input. The state is flattened similar to the previous algorithms. The action is converted to a one-hot vector. The state and the one-hot vector are concatenated together to form the input to the neural network. The network has 3 hidden layers each using a ReLU activation function. The input size is $CH+C$ for the combination of state and one-hot action representation. The hidden layers have size $CH/4$, $CH/8$, and $CH/16$. The output layer is a single value and the activation function is the sigmoid function to make the value between 0 and 1. The output of this function is the probability that the action taken in the given state is the expert action.

The Adam optimizer is used to update the policy function, the value function, and the

discriminator. The policy, value, and discriminator functions are updated every rollout. The discriminator is updated using the state-action pairs from the rollout and an equivalent number of state-action pairs sampled from expert behavior. The expert actions are gathered from running Bélády's algorithm on a set of 500 data access traces from the training set. The agent is then trained on the 2000. After 5 updates, the target policy function and the target value function are updated to the current policy and value functions.

### 5.4.4 Imitation Learning for Initialization

In addition to directly using imitation learning algorithms, the algorithms were used to intialize reinforcement learning algorithms. Two imitation learning techniques were used to train neural networks whose weights would then be used to initialize the neural networks of the A2C algorithms.

First, supervised learning was used to initialize the policy network for A2C. The policy network was trained to predict the expert action from a set of state-action pairs from Bélády's algorithm. This dataset was generated from saving state-action pairs from 500 cache traces and resulted in about 250 thousand expert state-action pairs. This was split into a training and validation set. Using Adam optimizer and cross entropy loss, the network was trained for 25 epochs with a learning rate of 0.001. This trained network was then used as initialization for A2C policy network. After using this initialization, the standard training procedure was run.

Second, GAIL was used as initialization for the policy and value networks for the A2C algorithm. The training procedure for GAIL was used as described above. After GAIL completed training, the weights of the neural networks for the policy and value networks were saved and then used to initialize the equivalent networks in the A2C algorithm.

# CHAPTER 6: RESULTS

The following section will detail the results of the methods and experiments described in the previous chapters. In the experiments, reinforcement learning algorithms and imitation learning algorithms were tested on a variety of different cache and data settings. The algorithms were examined on different cache sizes including 25, 50, and 100. They were also examined on two different data access patterns including Zipf with $\alpha = 1.3$ and $\alpha = 1.15$. In almost all of the settings, algorithms were able to achieve cache hit ratios that were higher than the baselines.

## 6.1 PERFORMANCE OF ALGORITHMS

The below section describes the performance of the different algorithms and shows how their training process progressed. For each algorithm, there is a training curve graph showing information about the performance of the algorithm such as Figures 6.1, 6.2, and 6.3. For these figures, the unit of the x-axis is an episode where each episode is one data trace from the training set. The y-axis shows the hit ratio. The blue line is the cumulative running average hit ratio on training set up until that episode. The orange line is the hit ratio of the algorithm on the evaluation set. The horizontal lines show that performance on the final testing set. The red horizontal line is the performance of the algorithm on the test set. The green line shows the optimal cache hit ratio possible obtained using Bélády's algorithm on the test set. The other lines show the LFU, LRU, and random policy performance on the test set.

Double Deep Q-learning resulted in the worst performance of the reinforcement learning algorithms explored. The algorithm was able to achieve performance better than random and improved over time but did not achieve results better than baseline algorithms in most settings. A variety of different network architectures and learning parameters were explored, but performance would not converge to a hit ratio higher than the baselines. An example training curve for the algorithm is shown in Figure 6.1. The algorithm's training performance begins right around the expected performance of a random cache policy. This is to be

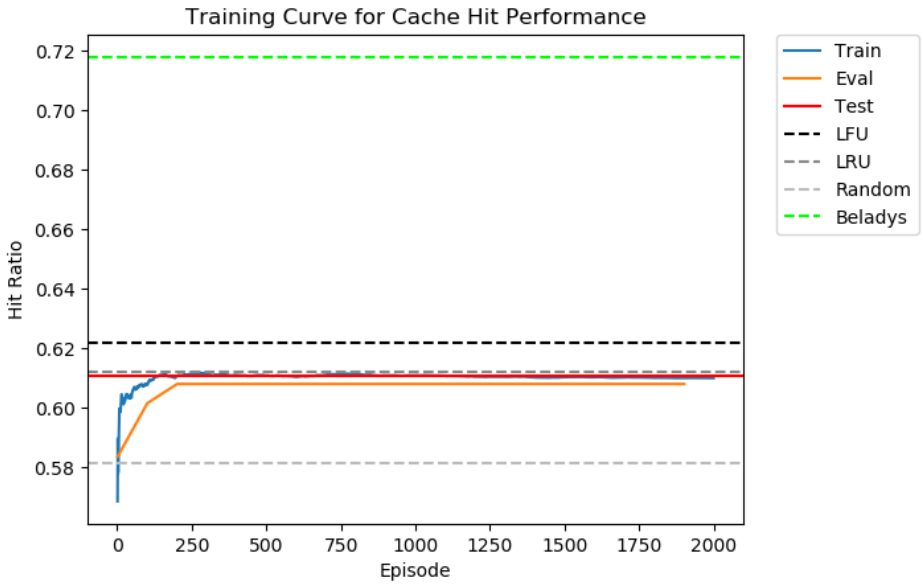



Figure 6.1: Training curve for Double Deep Q-learning using cache size 50 and Zipf distribution with $\alpha = 1.3$.

expected as the neural networks are randomly initialized. After a few training episodes, the algorithm's performance begins to improve. It quickly moves above the random performance and is clearly learning a better policy that improves cache hit ratio. The performance of the algorithm begins to converge after around 250 episodes where each episode is one data access trace from the training set. The algorithm converges to about the performance level of LRU in this setting and is unable to improve past this hit ratio.

Advantage Actor-Critic is able to learn a good policy that is able to outperform the results achieved by the previous value-based approach. The algorithm is even able learn policies that achieve higher hit ratios than those achieved by the LRU and LFU baselines. A training curve for the performance of the algorithm is shown in Figure 6.2. This curve looks rather similar to the training curve of DDQN but has some important differences. The algorithm's training performance begins around random performance and even slightly below. After a few episodes, the training hit ratio increases quickly. After around 300 episodes, the training and validation cache hit ratios are about as good as the best baseline algorithm. This is different than DDQN because it was never able to achieve hit ratio this high. Over the next 1000 episodes, the algorithm gradually increases training hit ratio, and validation hit ratio

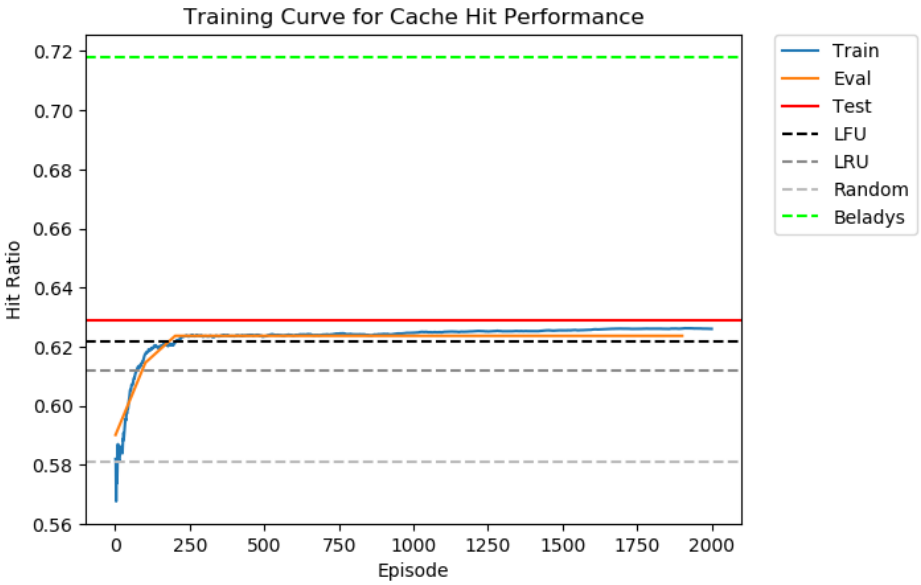



Figure 6.2: Training curve for A2C algorithm using cache size 50 and Zipf distribution with $\alpha = 1.3$. See Figure 6.1 for explanation of all lines.

improves slightly. The final testing hit ratio of the A2C algorithm is above both the LFU and LRU algorithms. However, there is still a significant gap between the achieved hit ratio and the optimal possible hit ratio.

The third main algorithm explored was the GAIL algorithm which trained the agent to mimic the behavior of the optimal actions derived from Bélády's algorithm. The algorithm achieved good results and produced performance better than LFU and LRU. However, it was not able to perform better than A2C. An example training curve and performance are shown in Figure 6.3. This training curve shows slightly different trends than the other previous training curves. The training performance again begins around random, but this time the algorithm improves performance more slowly. The training hit ratio gradually increases to around the baselines. The validation hit ratio takes around 500 episodes for it to converge to a level above the LFU and LRU baselines. This difference in training curve is most likely due to the algorithm doing imitation learning instead of reinforcement learning. The algorithm is learning to mimic an expert instead of learning form rewards. Additionally, this algorithm uses five neural networks compared to the two used by the other algorithms, so it makes sense for it to take more time to converge.

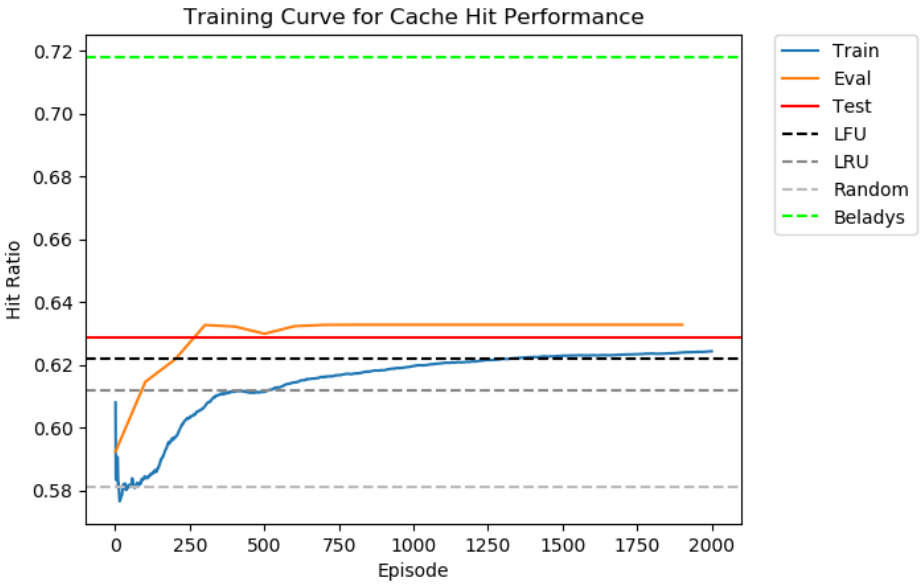
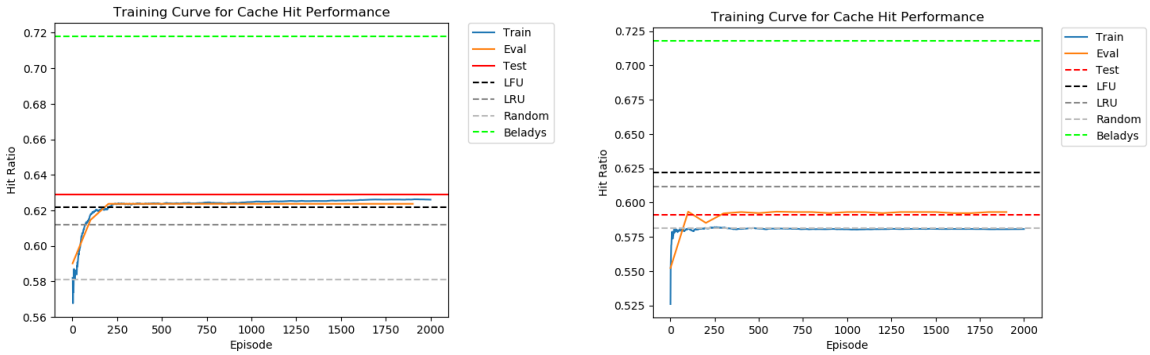



Figure 6.3: Training curve for GAIL algorithm using cache size 50 and Zipf distribution with $\alpha = 1.3$.

## 6.2 IMPACT OF INITIALIZATION

Over the course of trying these experiments, it became clear that initialization of the neural networks had significant impact on the performance of the algorithms. If the neural network's random initialization was poor, the end performance of the algorithm would be much lower than the performance of the same algorithm with a good initialization. An example of this problem is shown in Figure 6.4. Therefore, to provide good initialization, both supervised learning and GAIL algorithms were used to initialize networks for reinforcement learning.

Supervised learning was not able to provide great initialization for the neural networks. After training on a set of supervised data, this algorithm was able to predict what action the expert agent would take most of the time, but it was not accurate enough to have good performance. For example, Figure 6.5 shows the training curve for using for a cache of size 50 with Zipf distribution with $\alpha = 1.3$. This training curve shows the neural network's accuracy for predicting the expert action given the input state. This curved was obtained using a training set of around 250 thousand state action pairs retrieved from expert actions taken by Bélády's algorithm. A validation set of around 25 thousand examples were kept separate to evaluate performance. The supervised learning approach was only able to achieve around 78




(a) Good Initialization

(b) Bad Initialization

Figure 6.4: Comparison of performance of two different random initializations for neural networks. Bad initialization can lead to dramatically lower final performance.

percent accuracy on the validation set. Different neural network architectures and methods were unable to achieve higher validation accuracies than this. It was possible to achieve higher training accuracies, but this would lead to overfitting and the validation accuracy would decrease. This accuracy was far better than randomly guessing which action to take. However, it was unable achieve good performance when running the learned policy as the caches replacement policy. When using this neural network as initialization, the agent's testing hit ratio tended to be lower than if the agent had a random initialization.

GAIL provided a better method for learning to mimic the expert actions and provided a good initialization for the A2C algorithm. Initially, the agent was trained to mimic the behavior of the expert agent following Bélády's algorithm. This resulted in good performance that was a bit higher than the baseline methods even before additional training as shown in 6.3. The weights learned in GAIL were then transferred to the be the weights of the A2C policy and value networks. The A2C algorithm was then run and trained. This resulted in performance that was consistently around the peak hit ratio and in some situations performed better than standard A2C. An example training curve is shown in Figure 6.6. The figure shows that the algorithm clearly starts with higher performance than random initialization and continues to improve slightly from there. The GAIL algorithm is most likely able to provide better initialization to the A2C algorithm because it is better able to mimic the distribution of action taken by the expert. Instead of simply trying to predict the correct action based on supervised examples, GAIL attempts to take actions that follow

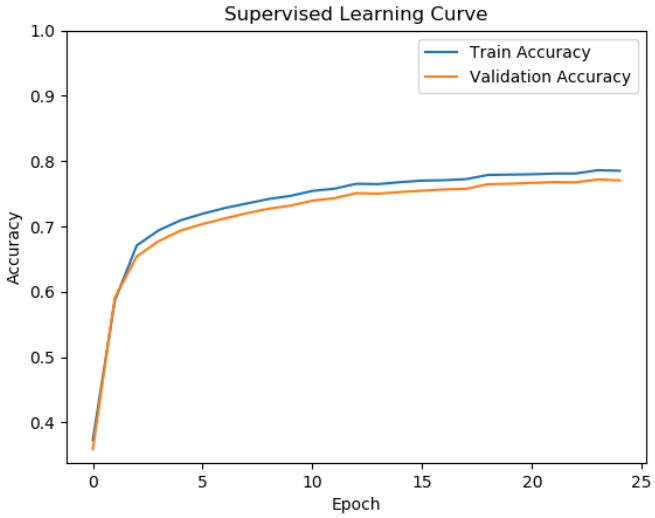



Figure 6.5: Training curve for supervised learning with cache size 50 and Zipf distribution with $\alpha = 1.3$.

a distribution that cannot be distinguished from the actions of the expert. This proved to be able to generalize better than supervised learning and provides a consistently good initialization for A2C.

## 6.3 DISCUSSION OF NUMERICAL RESULTS

This section contains a more detailed numerical set of results for each algorithm on a number of different cache sizes and access pattern distributions. A table of all final testing performance for the algorithms is shown in Table 6.1. These results are discussed, and intuition is given as to why these results occurred.

Looking at the baselines and Bélády's algorithm, there are some clear trends in performance based on cache size and Zipf distribution. Obviously, random cache policy has the lowest hit ratios in all settings and Bélády's algorithm has the highest. As cache size gets larger, cache hit ratio gets higher. This makes logical sense because as the cache gets larger more items can be in the cache making the likelihood of have an element in the cache larger. Similarly, the larger the $\alpha$ in the Zipf distribution, the larger the cache hit ratio. If the $\alpha$ value is larger, the frequency of each data item increases. This also makes the cache hit ratio higher because if elements are accessed frequently, they are more likely to already be in the

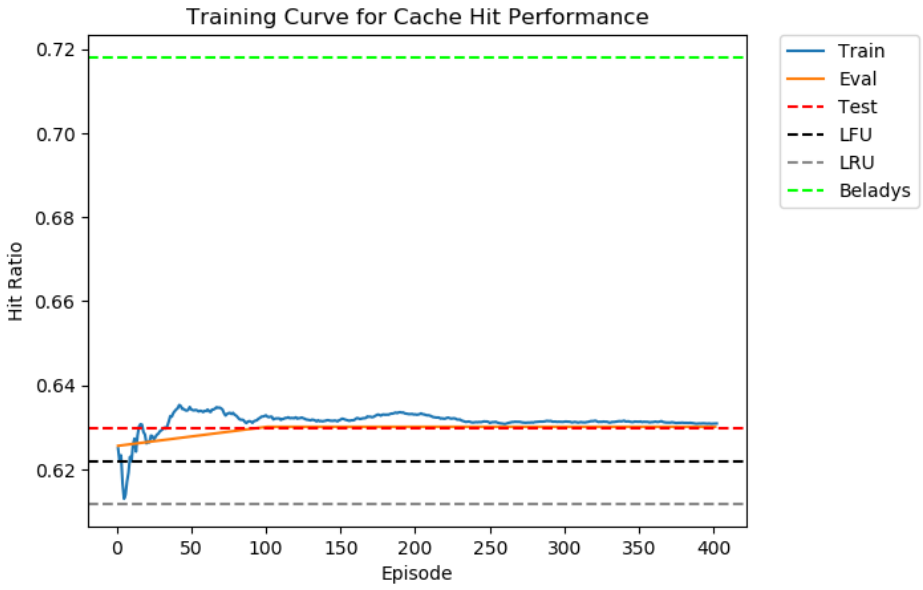



Figure 6.6: Training curve for A2C algorithm after being initialized using the GAIL algorithm. Trained with cache size 50 and Zipf distribution with $\alpha = 1.3$.

cache from the last access.

Another trend that the data shows is that as the cache size gets larger, the performance of the random policy gets closer to the performance of baseline algorithms and the optimal policy. For example, the random cache policy is much closer to LRU, LFU, or optimal policy when cache size is 100 compared to when the cache size is 25. This is caused because as the cache gets larger, it is easier to have more relevant data items in the cache purely because it is larger.

The data also shows that Bélády's algorithm does not increase performance significantly when larger cache sizes are used. For example, from cache size 50 to cache size 100 with $\alpha = 1.3$, the hit ratio only goes up 0.002. Some cache misses simply cannot be avoided. For example, if a data item is only accessed once, it will always result in a cache miss. Additionally, if an item is only accessed a few times and there are many accesses in between repetitions, it will be quite challenging to avoid a cache miss on later accesses. This is clearly evident with the optimal algorithm because even though the cache doubles in size it is barely able to improve performance.

Double Deep Q-learning did not perform great and resulted in mediocre performance in

| $\alpha$ | 1.30 | | | 1.15 | | |
|---|---|---|---|---|---|---|
| Cache Size | 25 | 50 | 100 | 25 | 50 | 100 |
| Random | 0.49320 | 0.58118 | 0.64287 | 0.23858 | 0.29520 | 0.36159 |
| LRU | 0.54253 | 0.61180 | 0.66307 | 0.28263 | 0.33612 | 0.38473 |
| LFU | 0.55687 | 0.62183 | 0.67152 | **0.29010** | 0.34721 | 0.39424 |
| DDQN | 0.54524 | 0.61057 | 0.67259 | 0.26520 | 0.33259 | 0.39516 |
| A2C | **0.56143** | 0.62912 | **0.68547** | 0.28594 | 0.34782 | **0.40533** |
| A2C (GAIL Init.) | 0.56029 | **0.63201** | 0.68124 | 0.28417 | **0.35103** | 0.39826 |
| A2C (Supervised Init.) | 0.54732 | 0.60913 | 0.65391 | 0.26934 | 0.33734 | 0.39151 |
| GAIL | 0.5411 | 0.62867 | 0.67718 | 0.28260 | 0.34163 | 0.40349 |
| Supervised Learning | 0.50231 | 0.57293 | 0.63343 | 0.25632 | 0.28494 | 0.36267 |
| Bélády's algorithm | 0.68215 | 0.71879 | 0.72075 | 0.44219 | 0.47475 | 0.47794 |

Table 6.1: Table of results showing a comparison of different algorithm's performance on different cache sizes and data distributions. The numbers are the hit ratios on the test set and the bold number shows the best performing algorithm in terms of hit ratio.

most settings. It typically was not able to outperform the baseline algorithms and resulted in lower cache hit ratios than LRU and LFU. It was clearly able to outperform the random cache replacement which shows that it was able to learn a a method better than simply random. In most settings, it performed similarly to baselines but typically a bit lower. This could be due to the somewhat large action space. Deep Q-learning methods have excelled in large state spaces, but relatively small actions spaces. For example, many Atari games have only couple possible actions. In this case, there are between 25 and 100 actions. Since Deep Q-learning attempts to model the $Q(s, a)$ function, this larger action space could make it challenging for the algorithm to achieve high performance.

For imitation learning, the numbers generally show that GAIL provided good performance while supervised learning imitation was ineffective. For supervised learning, the performance of the algorithm was typically close to random and rarely close to performing as well as the baselines. For $\alpha = 1.3$ and cache size 50 or 100, the algorithm performs worse than random. Clearly, attempting to directly copy actions does not generalize well and was not an effective way to learn a replacement policy. Without being able to directly copy the actions perfectly, the supervised learning approach cannot perform well and leads to it taking worse actions than random.

On the other hand, GAIL performs quite well. GAIL always achieves higher hit ratios

than random policies and commonly outperforms LRU and LFU. For $\alpha = 1.3$ with cache 50 and 100, GAIL performs better than any baselines. Similarly, for $\alpha = 1.15$ and cache size 100, GAIL outperforms the baselines by a significant amount. This shows that learning directly from the actions of an expert without receiving any reward function can be used to improve cache performance. GAIL is able to perform better with larger cache sizes which indicates that with more complexity it is able to perform better relative to baselines. This could indicate that with larger action sizes it is able to get closer to mimicking the expert or close enough that the performance is good. GAIL has been shown to be effective to perform in large or continuous action spaces and this environment would be another situation where it can perform well with more action choices.

Across most environments, the A2C algorithm was the best performing algorithm and was able to beat the performance of the baseline algorithms. With $\alpha = 1.3$ and some sort of initialization, A2C was as able to outperform the baseline algorithms when cache size was 25, 50, and 100. With $\alpha = 1.15$ and some sort of initialization, A2C was able to outperform the baseline algorithms when cache size was 50 and 100. The trend in the performance of A2C shows that generally as the cache size gets larger, A2C performs better than the baseline methods by greater margins. When the cache size is 25, A2C only outperforms LFU slightly when $\alpha = 1.3$ and performs worse than LRU when $\alpha = 1.15$. However, cache size is 100, A2C is able to outperform baselines by over 0.01 when $\alpha = 1.3$ and when $\alpha = 1.15$. This seems to indicate that as the cache size gets larger, the algorithm is able to improve more significantly compared to baselines. A2C is a policy gradient algorithm. This means it directly learns the policy function $\pi(a|s)$ instead of learning the $Q(s, a)$ function. Removing this intermediate learning allows the algorithm to more directly optimize the policy and do better with larger action spaces. The larger cache size gives the agent access to more possible actions and choices when following a cache policy. This increase allows the agent to learn more effective policies. When the cache size is smaller, A2C has a harder time learning optimal behavior because there are fewer actions and there is less room for small mistakes. However, A2C improves more when cache size increases from 25 to 50 than when cache size increases from 50 to 100. For $\alpha = 1.3$, A2C performs around 0.07 better when cache size is 50 compared to 25. A2C only performs around 0.05 better for cache size 100 compared to 50. This can

be attributed to the unavoidable cache misses that occur even with larger cache sizes.

For initialization, A2C can be slightly improved with GAIL initialization. For both $\alpha$ values, using GAIL initialization improved performance when cache size was equal to 50. This seems to indicate that training, based off an expert, leads to a policy that is effective in these settings. After this, the policy is able to be marginally improve by continuing to train using the rewards from the environment. In other settings, the GAIL initialization lead to comparable or slightly lower performance of A2C. This could indicate that the policy learned by GAIL was different than the optimal policy learned by A2C. Therefore, when A2C started to train it could have trouble as GAIL had already initialized the network in a way that was not helpful for A2C or lead to a local minimum. Supervised learning was ineffective for initialization. Supervised learning caused A2C to perform worse in all settings. Since supervised learning performs worse than random on its own, it makes sense for A2C to be negatively impacted by supervised initialization.

Overall, algorithms presented here are able to outperform the baselines policies. In a range of settings, the A2C method can outperform LFU and LRU. However, the hit ratios are not dramatically higher than the baseline performances. The performance of the algorithms is still a good bit lower than the performance of the optimal policy. However, this is somewhat to be expected as the optimal algorithm can only achieve these hit ratios by seeing future access patterns. It is infeasible to expect cache replacement policies to be able to achieve results overly close to the optimal. Another important aspect to consider here is training time. These results were achieved after a significant number of training exercises. This indicates that it could be difficult to use these algorithms in an online setting. In an online setting, the algorithms would have to learn the optimal cache policy while running. This could lead to a significant amount of time where the algorithms are not performing well. However, even though this online method might not be effective, these algorithms can be used in other situations well. Caching systems can store past data access patterns. Then, in an offline setting, these algorithms can be trained to perform well on these traces. After training on these traces, the algorithms should be able to outperform the baselines in new unseen data access patterns if the underlying pattern of the accesses is similar. Having similar access patterns is quite common in databases or applications that run similar or

repetitive jobs. Using these algorithms trained on these specific data patterns, could lead to significant performance increase in terms of cache hit rate. Increasing the cache hit ratio could greatly improve overall cache performance and improve run time of applications.

# CHAPTER 7: RELATED WORK

Improving cache replacement policy has been explored in several different works and has been approached in a number of different ways. This area of research has been popular because cache performance has a significant impact on many areas of computing. Due to the recent improvements of machine learning, many approaches have explored using learning-based approaches to improve cache replacement policies. In this section, related approaches will be explored and compared to the work presented here.

## 7.1 MACHINE LEARNING APPROACHES

For several years, learning-based approaches have been used to determine cache policy. One category of these works has used adaptive cache replacement policies. These are approaches that mostly use standard heuristic-based replacement policies but have some sort of mechanism to change the replacement policy in different situations. Adaptive Replacement Cache (ARC) uses an adaptive approach to balance the importance of both LRU and LFU replacement policies [25]. If LRU is performing better than LFU, then an LRU approach is used more frequently than the LRU approach. If the LFU starts performing better, it will be used more. This builds on other works describing the spectrum of policies made up of combinations of LRU and LFU [11]. Another approach called Adaptive Caching Using Multiple Expert (ACME) uses machine learning to decide how to weight a number of different expert static replacement policies [26]. This algorithm has a pool of cache policies that it runs simultaneously. Each policy is given a weight that is updated based on if following that policy would have resulted in more cache hits or misses. The action chosen is then based off the combination of weights of these static methods.

Supervised learning approaches have also been used to approximate cache replacement policies. Neural networks have been trained on real-world data to predict if an piece of data will be requested again [6, 27]. The real-world memory access traces are labeled to indicate whether each piece of data will be accessed in the future. The neural network takes in input about the cached item such as frequency and recency of use and predicts if the item will be

accessed again. Using the neural network, objects are removed if they are not predicted to be reused. Similarly, other non-linear and linear classifiers have been used in similar methods as described above [5].

## 7.2 REINFORCEMENT LEARNING APPROACHES

Reinforcement learning approaches have also been explored relating to cache policy. Going back a few years, reinforcement learning has been applied to memory management for improving performance [28]. The work uses an older tabular version of Q-learning to improve memory scheduling and processing speed. Another work models the problem of cache policy as a multiarmed bandit problem [29]. A multiarmed bandit problem is a common version of a reinforcement learning problem where an agent must choose which level to pull at each timestep to maximize reward. The work explores a few older reinforcement learning approaches to solve this problem. Another work used reinforcement learning to learn the popularity distribution of content requests [30].

Recent work has even examined using deep reinforcement learning for content caching. One such work used a version of actor-critic methods to perform content based cache replacement policy [31]. This work uses a similar deep reinforcement learning approach as to the work in this thesis but uses a different state representation of the cache. Each piece of data accessed is assigned a content ID and this is used to derive the state representation. The work uses a deep neural network to approximate the policy function.

## 7.3 COMPARISON

The work presented in this thesis is different from the previous work described and shows more promise than other techniques. First, when compared to static methods such as LRU and LFU, the work presented here has been shown to outperform them in a reasonable number of settings and tends to perform better as cache size gets larger.

When compared to adaptive cache policy techniques, the reinforcement learning approaches presented here can learn much more complex learning policies. These adaptive

approaches are only able to learn simple combinations and mixtures of standard heuristic policies. This limits the performance that can be achieved using these approaches. The approach here can learn almost any possible policy based on the state inputs. Additionally, the learning algorithm here can be changed dramatically to fit more diverse data distributions.

When comparing with supervised learning approaches, the reinforcement learning approach presented here is far better able to learn long-term importance of decision making. The approach described in this thesis is modeled as an MDP to maximize cumulative reward. The reward function encourages reducing the number of cache misses and encourages maximizing the long-term cache hit ratio. Supervised learning approaches struggle to model the long-term dependencies of cache replacement decisions and only considered the short-term choice. This is shown in this work by the performance of the supervised learning algorithm used as initialization. The choices it learned were not productive in terms of improving performance. Reinforcement learning is much better suited to solve this problem than supervised learning.

The most direct comparison to make with this work is to other reinforcement learning approaches. Older reinforcement learning approaches did not use dramatically effective function approximation techniques. The older approaches did not use neural networks which prevent the algorithm from being able to learn overly complex state spaces and larger action spaces. The deep learning approach used in this thesis enables the algorithm to use large state inputs and large action sizes. Other deep reinforcement learning approaches applied to cache replacement learning use different state representations and define the Markov Decision Process differently. The state defined in this work is better than those described in other works because it is purely index-based and quite expressive. The state of the cache never depends on the content in the cache and only about the access histories of each index in the cache. This means that even if the algorithm is run on different sets of data, it will be able to perform well if the underlying access pattern is similar. Using this purely index-based representation, it is able to achieve comparable performance to other deep reinforcement learning approaches. Additionally, the state described in this work allows any combination of policies such as LRU and LFU because the state is very expressive. The algorithm can learn complex trends in the correlation of accesses.

Additionally, the use of imitation learning following the optimal policy is a novel idea that shows promise for helping initialize complex neural networks. Previous works have not explored the use of Bélády's algorithm as an expert to learn from using neural networks [32]. GAIL is a state-of-the-art technique that had not been previously applied to cache replacement policy. This algorithm has shown to be able to perform well in settings and rival reinforcement learning approaches.

# CHAPTER 8: CONCLUSION

In this thesis, reinforcement learning is used to attempt to improve cache replacement policy. The problem of cache replacement policy is presented as a partially known Markov decision process. This work presents a novel state representation for the cache. The cache state is represented as a history of index cache accesses. The state does not consider the content in the cache and simply is based on the history of access at each index of the cache.

Using this new state representation, recent state-of-the-art deep reinforcement learning algorithms were explored to find optimal policies to solve this problem. Several approaches were used including value-based algorithms such as Double Deep Q-learning and policy gradient based algorithms such as Advantage Actor-Critic.

In addition to reinforcement learning, imitation learning was used as a novel approach to improve initialization of standard reinforcement learning algorithms. The clairvoyant Bélády's algorithm was used as an expert agent for algorithms to attempt to mimic and achieve higher performance. For imitation learning algorithms, supervised learning and the Generative Adversarial Imitation Learning algorithm were used to mimic this expert agent. These algorithms were directly explored and used as initialization for reinforcement learning algorithms

The proposed methods were then tested in simulated environments. A simple cache model was implemented to test performance of these algorithms on improving cache hit ratio. Simulated data access patterns were generated following the Zipf distribution to mimic real data and test performance of these algorithms. As a point of comparison, the algorithms were compared to baseline cache replacement algorithms such as least frequently used and least recently used. The results show that in most settings the algorithms are able to achieve performance comparable or better than the baseline methods. The Advantage Actor-Critic algorithm was able to consistently outperform the baseline methods in terms of cache hit ratio in a variety of experimental settings. The imitation learning algorithms were somewhat effective in mimicking the optimal algorithm and acting as initialization for reinforcement learning algorithms.

Based on the results in this work, these algorithms can be applied to current caching sys-

tems. In databases with consistent data distributions, the algorithms and methods described in this work can be used to train policies that outperform currently used heuristic-based cache replacement policies. After training these algorithms in an offline setting, the algorithms could be used as cache replacement policies that improve cache hit ratio and therefore improve overall cache performance. This result could be quite beneficial as improving cache performance could have significant impact on a wide range of applications such as speeding up web traffic and reducing database access times.

For future work, the algorithms and approaches described here should be extended and explored in real database cache systems. The experimental process in this work used a simulated and simplified cache model. The model was meant to simulate the real-world problems as much as possible, but performance should be explored in real-world settings. The algorithms seem to show greater performance relative to baselines as the problem gets more complex. Additionally, the index-based cache representation and neural network approximators should facilitate effective scalability to large scale cache problems with complicated data access patterns. These two factors seem to indicate that with further exploration on more challenging cache problems, the work presented here could provide even better results.

## REFERENCES


[1] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: Weschester Publishing Services, 2018.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

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

[4] K.-Y. Wong, "Web cache replacement policies: a pragmatic approach," *IEEE Network*, vol. 20, no. 1, pp. 28–34, 2006.

[5] T. Koskela, J. Heikkonen, and K. Kaski, "Web cache optimization with nonlinear model using object features," *Computer Networks*, vol. 43, no. 6, pp. 805–817, 2003.

[6] J. Cobb and H. ElAarag, "Web proxy cache replacement scheme based on back-propagation neural network," *Journal of Systems and Software*, vol. 81, no. 9, pp. 1539–1558, 2008.

[7] S. Podlipnig and L. Böszörmenyi, "A survey of web cache replacement strategies," *ACM Computing Surveys (CSUR)*, vol. 35, no. 4, pp. 374–398, 2003.

[8] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[9] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[10] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[11] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE transactions on Computers*, no. 12, pp. 1352–1361, 2001.

[12] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton et al., "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[13] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

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

[15] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, pp. 1928–1937.

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

[17] K. Janocha and W. M. Czarnecki, "On loss functions for deep neural networks in classification," *arXiv preprint arXiv:1702.05659*, 2017.

[18] J. Ho and S. Ermon, "Generative adversarial imitation learning," in *Advances in Neural Information Processing Systems*, 2016, pp. 4565–4573.

[19] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

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

[21] E. Jones, T. Oliphant, P. Peterson et al., "NumPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[22] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1. IEEE, 1999, pp. 126–134.

[23] C. Cunha, A. Bestavros, and M. Crovella, "Characteristics of www client-based traces," Boston, MA, USA, Tech. Rep., 1995.

[24] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[25] N. Megiddo and D. S. Modha, "Outperforming lru with an adaptive replacement cache algorithm," *Computer*, vol. 37, no. 4, pp. 58–65, 2004.

[26] I. Ari, A. Amer, R. B. Gramacy, E. L. Miller, S. A. Brandt, and D. D. Long, "Acme: Adaptive caching using multiple experts." in *WDAS*, 2002, pp. 143–158.

[27] S. Romano and H. ElAarag, "A neural network proxy cache replacement strategy and its implementation in the squid proxy server," *Neural computing and Applications*, vol. 20, no. 1, pp. 59–78, 2011.

[28] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3. IEEE Computer Society, 2008, pp. 39–50.

[29] P. Blasco and D. Gündüz, "Learning-based optimization of cache content in a small cell base station," in *Communications (ICC), 2014 IEEE International Conference on*. IEEE, 2014, pp. 1897–1903.

[30] M. S. ElBamby, M. Bennis, W. Saad, and M. Latva-Aho, "Content-aware user clustering and caching in wireless small cell networks," *arXiv preprint arXiv:1409.3413*, 2014.

[31] C. Zhong, M. C. Gursoy, and S. Velipasalar, "A deep reinforcement learning-based framework for content caching," in *Information Sciences and Systems (CISS), 2018 52nd Annual Conference on*. IEEE, 2018, pp. 1–6.

[32] A. Jain and C. Lin, "Back to the future: leveraging belady's algorithm for improved cache replacement," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 78–89.