A Reproduction of the PARROT Cache Replacement Solution (2024)

Loévan Bost, Andrew J. Beckmann, Alexander Tayyeb, *ECE 492(055)/592(106) Students, NCSU*

***Abstract*— The paper *An Imitation Learning Approach for Cache Replacement* by Liu et al. (2020) proposes a new imitation learning based cache eviction policy, PARROT. This paper focuses on understanding, setting up, and reproducing the results from the PARROT model. This includes configuring the environment for simulation, setting up simulation workloads, applying the PARROT approach, and simulating the cache eviction policy. Furthermore, it involves a deep dive into the core concepts, techniques, and algorithms used, with mathematical explanations and interpretation of the results.**

***Index Terms*—** Markov decision Process (MDP), Long-short term memory (LSTM)

I. Introduction

CACHE replacement policies are very important as they are used to manage which data remains in the cache and which data is evicted. The PARROT approach, which is described in the paper, introduces an imitation learning framework that leverages Belady's optimal policy to make an informed eviction decision. This oracle policy computes the optimal eviction decision given the future cache accesses. While Belady's policy is impractical in real time, PARROT is able to imitate its decision making based on past cache accesses. The approach applies a neural network architecture consisting of LSTM to estimate the best decision.

The purpose of this homework assignment is to implement, reproduce, and analyze the PARROT approach using the provided GitHub repository. The first step is to set up a working environment to run the code. Once the environment is configured, different workloads and simulations can be done to reproduce the baseline results from the paper. This includes running the model against other traditional methods and comparing the performances.

Another focus for the homework assignment is to interpret the results of the code and simulations. To properly comprehend the results, a thorough understanding of the paper and the learning model is required. Studying the structure of the neural network, use of cache accesses, and the imitation learning process is key to mastering all aspects of the homework.

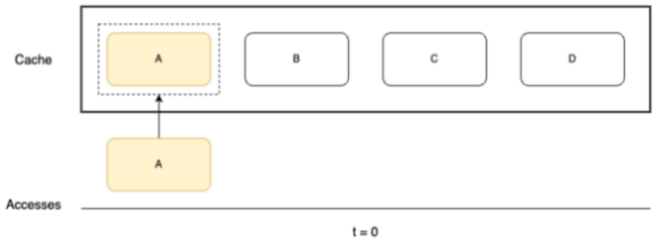
In addition to gaining a technical understanding of the PARROT approach, this homework also allows for an example of a machine learning model and its application in the real world. By studying the imitation learning model and how it is trained, it will provide valuable insights on how models can mimic optimal decision-making policies. A comprehensive understanding of the imitation learning model will be developed, while demonstrating how machine learning models can generate decision making processes based on various data, such as cache access patterns.

II. Background

Efficient memory access is critical for maintaining high computational performance, of those key factors is cache management. Particularly how data is replaced within the cache. Caches are small data storages that provide a fast access to frequently used data. However, when it becomes full, data must be evicted to make room for new data. This decision is made by the cache replacement policy.

The Markov Decision Process contains several key components which include the state space, action space, transition dynamic, and reward function. The state space contains three different elements: the cache state, current access, and the access history. The cache state contains the current contents of the caches which have a memory address stored in each one. The current access includes both the memory address being accessed and the program counter with the instruction. The access history captures the patterns that help the model predict future access behaviors. These elements ensure that the cache replacement policy has access to all relevant information to make an informed decision.

The action space depends on whether the current memory access result is a hit or a miss. A cache hit occurs when the data being requested is already stored in the cache, which allows for fast retrieval. A cache miss, on the other hand, happens when the data requested is not found in the cache (Figure 1). Therefore, an eviction must be made to store the information into the cache. The action space consists of deciding which of the lines in the cache set should be evicted to make space for the new memory address in the case of a cache miss. However, if the result is a cache hit, no eviction is necessary, and no action is taken. Therefore, an eviction is only necessary in the event of a cache miss.



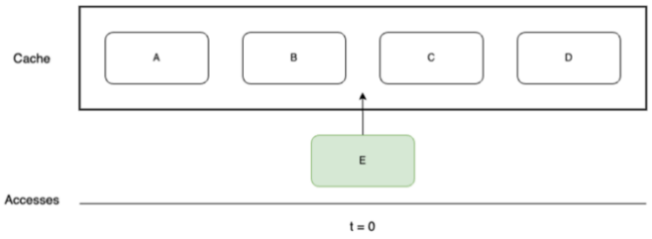


Figure 1. Cache hit and miss. In the top image at t=0, line A is accessed and is already in the cache, causing a cache hit. In the bottom image at t=0, line E is accessed and is not in the cache, causing a cache miss.

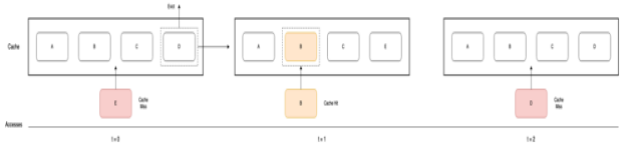


Figure 2. Cache transition and evolution. As a result of a cache miss, the cache will evict a line and insert a new one. Therefore, as t increases, the state of the cache will change.

The cache state will change in response to the memory access and changes made in the cache. When memory access results in a cache hit, the state of cache will not change. But in the result of a cache miss, the policy must evict one of the existing cache lines to make room for the new data. The eviction will depend on the cache replacement policy used. Once the policy decides which line to evict the state of the cache is updated, and the new block is inserted into the cache (Figure 2). Therefore, the transition dynamics are driven by the cache hits and misses which evolve over time as it responds to the memory access and policy eviction decisions.

The reward $R(s_t)$ is 0 for a cache miss and 1 otherwise for a cache hit. The goal is to learn a policy $\pi_\theta(a_t \mid s_t)$ that maximizes the total number of cache hits possible for a sequence of different cache sequences $(m_1, pc_1), \ldots, (m_T, pc_T)$. The reward function is essential for the policy to learn to make eviction decisions that reduce the number of cache misses.

The cache replacement problem involves making a series of decisions over time that affect future states and outcomes. Because of the inherently stochastic nature of cache replacements and the fact that the dynamically changing state space is partially controllable, the cache replacement algorithm is formulated as a Markov-Decision Process (MDP). A Markov-Decision Process, or MDP for short, provides engineers with a framework for sequential decision making where outcomes are partially unknown. In this paper, an imitation learning paradigm utilizes an MDP to derive effective cache management strategies. An MDP is characterized using states (S), actions upon those states (A_s), the reward function (R), and the transition dynamics (P).

III. Cache Simulation Environment

PARROT cast as Imitation Learning

The paper uses MDP states to delineate the current cache, the memory address of the cache and the history of all cache accesses. Figure 3 shows the overall system diagram. The action space holds two decisions, either a cache hit or a cache miss. During a cache miss the PARROT model decides which line to evict based on its generalization of the current cache state. It compares this decision to the expert policy (Belady's) and generates a ranking loss report, which it then uses to update its parameters (θ). During the next memory access (s^a_{t+1}) ,the cache is updated to remove the line ℓ_w and replace when the last memory access. During a cache hit, the reward function is increased by 1 and the cache simply transferred to the next time step (no operation). In both cases, the history is independent of the action A_t meaning the history is updated by taking the last state's memory access history and adding the current memory access during both a cache hit and miss.

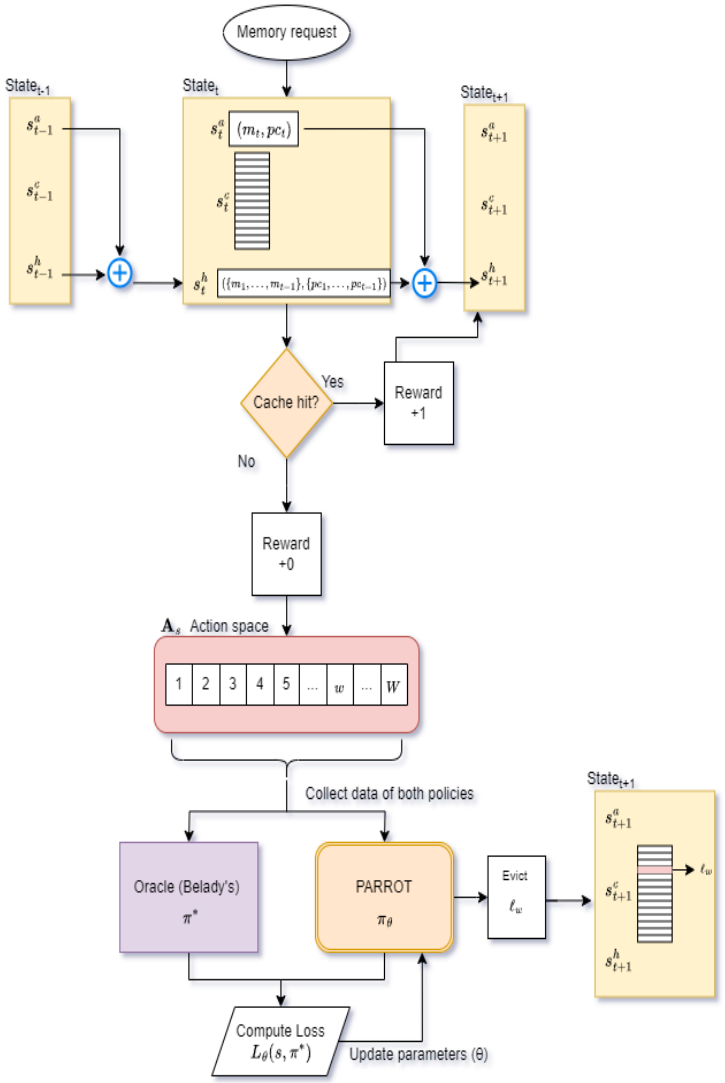


Figure 3. The overall system diagram of the cache replacement problem casted as imitation learning on an episodic Markov decision process.

The paper introduces an imitation learning approach by leveraging Belady's algorithm as an expert policy for which to imitate. Belady's is an oracle policy that computes the optimal cache eviction decision based on knowledge of future cache access queries. Because knowledge of the future is unknown, deciding which cache line to evict can be exceedingly difficult. Imitation learning allows the system to mimic the behavior of an expert to perform tasks, rather than learning from direct feedback like reinforcement learning. Imitation learning trains the agent by providing examples of the correct behavior to generalize for unseen situations, or states.

Cache Setup

To test the performance of a new, imitation learning-based cache eviction policy, the authors simulated a three-level CPU cache architecture involving a 4-way 32 KB L1 cache, an 8-way 256 KB L2 cache, and a 16-way 2 MB last-level cache. The new policy was implemented on the last-level cache and the LRU policy was maintained for the L1 and L2 caches. The SPEC CPU 2006 benchmark workloads were used for trace generation, where raw memory accesses, containing memory addresses m_t and program counters pc_t, were collected and sequentially logged over a 50 second interval for each workload. These sequences were then filtered through the top two cache levels, resulting in the last-level cache accesses. For workload dataset reduction, 64 random last-level cache sets were selected, resulting in around 5M accesses which were then split into 80% for training, 10% for validation, and 10% for testing.

Cache State

The collected sequential cache data was conceptualized and presented as MDP s_t states, s_t, representing the state of the cache at time t. A state, defined as $s_t = (s_t^a, s_t^h, s_t^c)$, includes three components, where $s_t^a = (m_t, pc_t)$. represents the current cache accesses memory address and program counter, $s_t^c = \{l_1, \ldots, l_w\}$, represents the W cache lines in the cache set corresponding to the current cache access, and $s_t^h = (\{m_{t-H+1}, \ldots, m_{t-1}\}, \{pc_{t-H+1}, \ldots, pc_{t-1}\})$, represents the past H cache accesses. A cache hit is defined as $m_t \in s_t^c$, and a cache miss as $m_t \notin s_t^c$. Given a cache hit, no action is taken, as no line must be evicted. However, given a cache miss, the action set A_{s_t} comprised of integers $\{1, \ldots, W\}$, correlating to the current lines in s_t^c, where selecting line w out of A_{s_t} corresponds to evicting line l_w from the cache.

Neural Network Architecture

The PARROT model developed by the authors is a composite model, consisting of an embedder, an LSTM, an attention mechanism, and a small, linear network. A complete diagram (Figure 4) and accompanying description of the model's architecture is shown below.

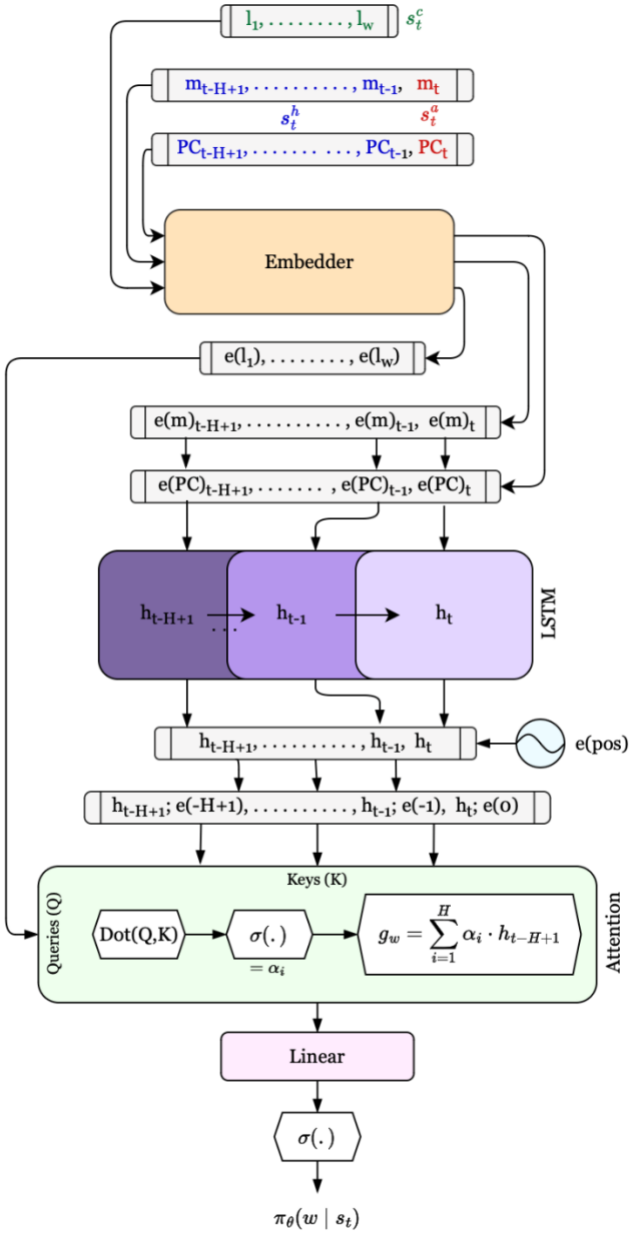


Figure 4. PARROT model architecture showing the data's format and flow through the model.

PARROT Architecture Description:

I. Embedding:

The cache state s_t is available at the time step of a given memory access and its components are used as features in the model. The memory addresses $\{m_{t-H+1}, \ldots, m_t\}$ and program counters $\{pc_{t-H+1}, \ldots, pc_t\}$ are separately embedded into embedding vectors with unique embeddings for all addresses and PCs, resulting in embedded addresses $\{e(m_{t-H+1}), \ldots, e(m_t)\}$ and embedded PCs $\{e(pc_{t-H+1}), \ldots, e(pc_t)\}$. Although the exact architecture of the embedder is not disclosed, the embedding weight matrix W_m is of size $(n_{m+1} \times d_m)$ where n_{m+1} is the number of unique addresses plus one for unseen addresses, and d_m is the length of the embedding vector which an address's information is compressed into. The embedding weight matrix for PCs is defined similarly. The W cache lines in s_t^c are similarly

embedded using the same embedder, giving $\{e(l_1), \ldots, e(l_w)\}$. Utilizing the same embedding architecture for multiple features is beneficial not only for maintaining model size, but for recognizing patterns and creating relationships between features, as they are embedded in a similar vector space.

II. LSTM:

After embedding, the memory address and PC of the current access s_t^a and previous accesses s_t^h are used as inputs into a generic, feed-forward LSTM, generating the cell state c_t and hidden state h_t. The LSTM contains H hidden states for each memory access, where the input to h_t includes the current cache access and the output of the previous hidden layer: $h_t = LSTM([e(m_t); e(pc_t)], h_{t-1})$. The result of the LSTM is a matrix of size $(n_{m+1} \times d_h)$ where d_h is the size of the hidden layer output. The outputs of the hidden layer h_t can be conceptualized as a sort of encoding of the current memory access and its relationship with the past H memory accesses.

III. Positional Embeddings:

Following the generation of the H hidden states, the states are then concatenated with a positional encoding $e(pos)$, developed by *Vaswani et al. (2017)*, to give further information on the timestamp of each state. This form of positional encoding takes advantage of the periodic nature of sine and cosine functions with different frequencies representing different positions in a sequence. This concatenation generates a matrix of size $(d_H \times (d_h + d_{pos}))$, where d_H is the number of hidden layers, d_h is the size of hidden layer, and d_{pos} is the size of the positional encoding. The form of this output is:

$$\begin{bmatrix} h_{t-H+1}; e(-H+1) \\ \vdots \\ h_{t-1}; e(-1) \\ h_t; e(0) \end{bmatrix} \in \mathbb{R}^{(d_H \times (d_h + d_{pos}))}$$

IV. Attention:

The heart of the PARROT model lies in its attention mechanism. The attention mechanism is a sort of combination of the method used by *Vaswani et al. (2017)* and *Luong et al. (2015)*. This mechanism uses the positionally concatenated hidden layers as keys and the embedded cache lines as queries to form contextualized lines g_w. These lines can be thought of as viewing each cache line l_w of s_t^c in terms of the current access and the history of previous accesses. A more mathematical discussion of the attention mechanism is discussed in the *Attention Mechanism* section.

V. Linear Layers + SoftMax:

The contextualized lines $\{g_1, \ldots, g_w\}$ are passed through a final, fully connected linear network where non-linear relationships between contexts can be learned and W output values corresponding to each cache line are produced. These are then passed through a SoftMax layer, where the argument w with the largest probability is then chosen, corresponding to evicting line l_w from the cache.

Training Process

The training process for the PARROT model is used to imitate Belady's optimal cache replacement policy using imitation learning. Belady's policy requires knowledge of future accesses, which is not available during real-time execution. The goal is for the PARROT model to learn a policy that can approximate Belady's decision making process only on past access patterns.

The main idea for the training is to visit a set number of states B and then to update the parameters θ to make the same decision as the optimal policy based on each state $s \in B$ and via the loss function $L_{\theta}(s, \pi^*)$. The first step is to convert a sequence of cache accesses $(m_1, pc_1), \ldots, (m_T, pc_T)$ into states $s0, \ldots, sT$, to calculate the best decisions (Figure 9). The data--- is collected using Belady's optimal policy to build a dataset of different states. Since Belady's has knowledge of the future and which states will be used, it is able to compute the perfect eviction decision at every cache miss. This creates a model of the best decisions for the model to learn from and imitate.

As the training progresses, PARROT transitions to using its learned policy to collect states, ensuring that the model encounters a realistic variety of different cache patterns. This also helps mitigate compounding errors that could arise if the model were only exposed to state under Belady's policy. The DAgger algorithm helps ensure that the collected states evolve as the model improves, to help reduce the chance of errors compounding over time. The PARROT model uses backpropagation over a limited window of past accesses, updating its parameters based on the result of different sequences. The batches of states s_{l-H} $s_{l-H}, \ldots, s_{l+H}$ are sampled and the LSTM hidden states are initialized on the cache accesses of s_{l-H} to s_{l-1}. Then the replacement policy is applied to compute the loss $L_{\theta}(s, \pi^*)$ which encourages to policy to make the best decisions (Figure 9).

In each iteration, the model predicts which cache to evict based on the current state and history, The loss function measures how well the decision matches Belady's optimal decisions. PARROT then utilizes a ranking loss, which penalizes the model if an eviction occurs that will be reused soon. The result of the training is to create a model that can effectively improve cache hit rates compared to traditional models.

DAgger

Compounding errors can be encountered during the training process if the state space (B) is collected using the expert policy (π_*). Collecting under the expert policy can lead to unfamiliar states during testing under the learned policy (π_θ). The learned policy has been observed to make inaccurate eviction decisions due to this phenomenon and continues to generate mistakes exponentially over each iteration of testing.

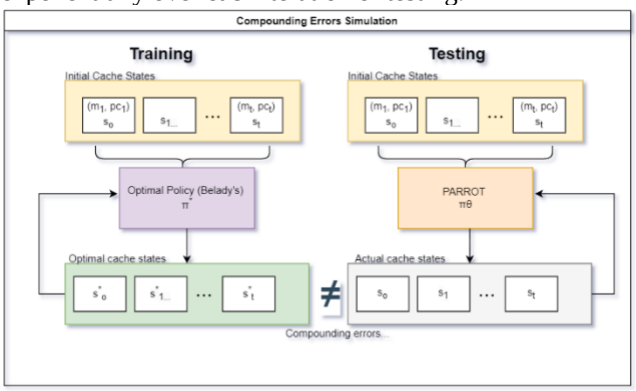


Figure 4. Compounding errors seen during testing

To combat compounding errors, PARROT uses the DAgger algorithm. DAgger avoids unfamiliar testing situations by collecting states using the learned policy instead of the expert policy during training. The current policy updates its parameters and collects the states to an aggregate state space which the policy uses to train for the next iteration. Every 5000 parameter updates the dataset is recollected under the initial policy to mitigate a dataset drift.

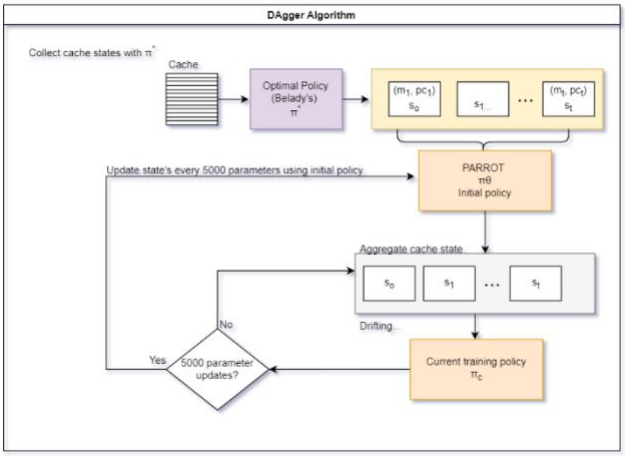


Figure 6. Flow of the DAgger algorithm

The implementation of the DAgger algorithm achieved 9.8% higher normalized cache rates than training on states visited by the expert policy.

Byte Embedder

The byte embedder is utilized to help reduce the memory required to represent memory addresses and program counters. The byte embedder breaks down a memory address into different parts to make them easier to process.

The byte embedder breaks down each memory address into individual bytes. For example, in the diagram the memory address 0xA1B2C3 is broken down into three separate parts which are A1, B2, and C3 (Figure 8). Then, each byte is separated and treated differently. Once they are all separated, they are passed using a small linear layer over their concatenated outputs. The byte embedder learns a hierarchical representation, which separately represents the large memory areas and the smaller objects that have lower bytes.

The byte embedder is an efficient way to reduce the complexity of handling large numbers of unique addresses and PCs, as it breaks it down into smaller, more manageable, components which ultimately helps reduce the amount of memory required.

Attention Mechanism

As mentioned in the model description section, the attention mechanism of the PARROT model is key to developing complex relationships between the cache lines in s_t^c and the current access s_t^a. The process is as follows:

1. Perform matrix multiplication between a weighting matrix W_e and the hidden state matrix: $W_e h_{t-H+i}$.
2. Take the dot product of the queries and the keys: $e(l_w)^T W_e h_{t-H+i}$. The resulting matrix values are known as scores, which tell how much attention a cache line should give attention to a given hidden state. Essentially this allows the model to look at the relationships between each cache line and the history of the model's cache accesses.
3. Take the SoftMax of the dot product: $\alpha_i = \sigma(e(l_w)^T W_e h_{t-H+i})$. Taking the SoftMax accentuates the differences in the amount of attention the model will give to the hidden states and will generate a probability distribution correlating to how much emphasis the model should place on the history of cache accesses.
4. Perform a linear combination of the probabilities with each hidden layer: $g_w = \sum_{i=1}^{H} \alpha_i h_{t-H+i}$. This final linear transformation creates the contextualized cache lines, where the context is built by multiplying each hidden state with its corresponding score (or amount of attention) and summing them together to create a vector that represents the emphasis that the past cache accesses have on each line.

This complete process is shown in Figure 7.

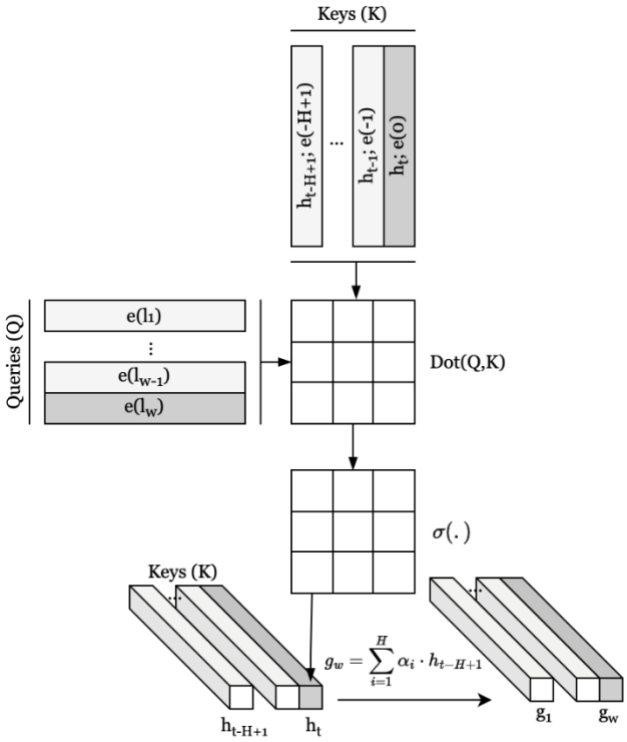


Figure 7. Attention mechanism used in PARROT to find the contextualized cache lines.

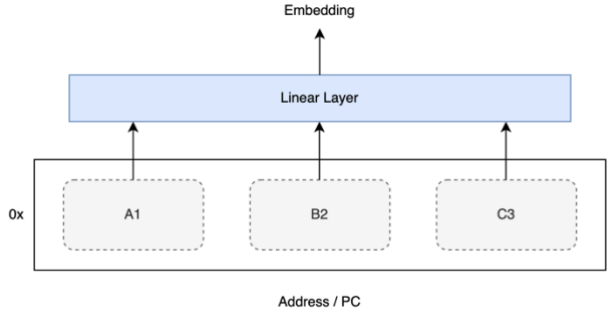


Figure 8. Byte Embedder. Breading down memory addresses into different components.

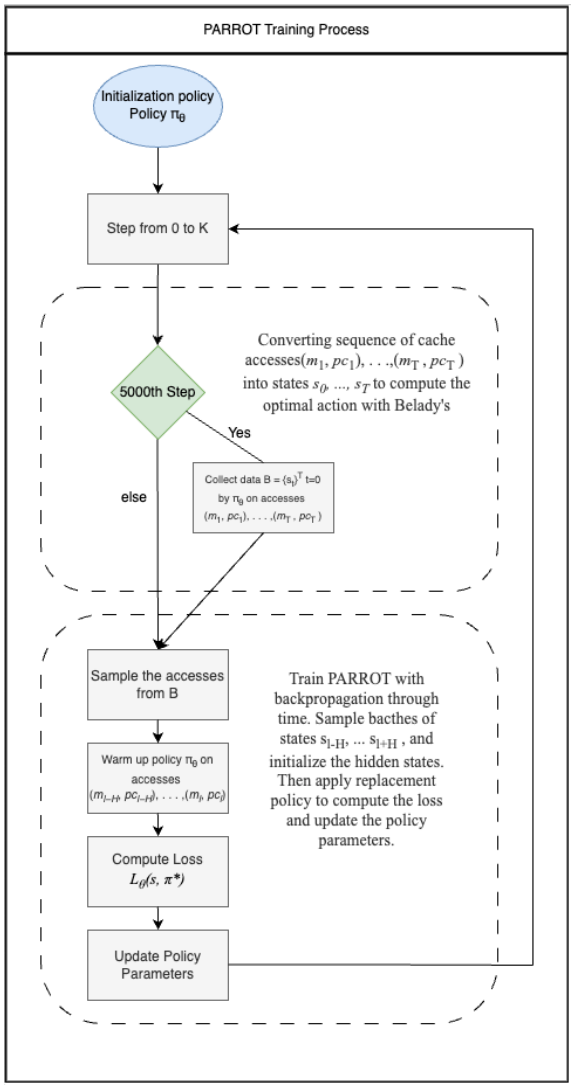


Figure 9. Training process flow chart.

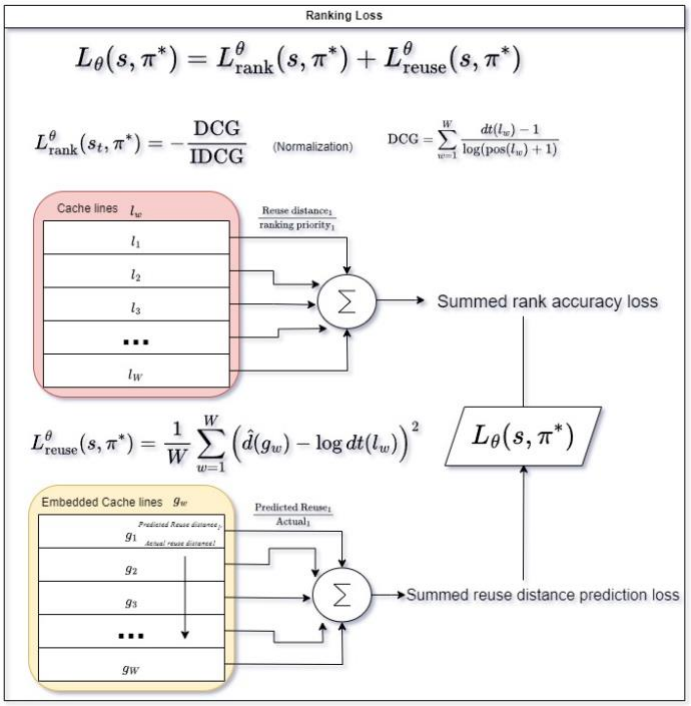


Figure 10. Total loss as a summation of Ranking loss and predicted reuse distance loss

Ranking Loss

Using a normal log-likelihood (LL) encourages policy to match the single optimal action in reference to the expert policy. In this case, more supervision can be provided by embedding probability to the entire distribution of actions. This idea is similar to knowledge distillation (Hinton et al., 2015), where a student model learns from a distribution of possible answers from a teacher, rather than only matching a single optimal action. It helps the model generalize better using the extra information. PARROT makes use of this additional supervision by encouraging the learned policy to prioritize actions in line with the expert policy distribution. Specifically, the loss function in parrot is based on ranking loss (Figure 10) to improve cache hit rates, based on Normalized Discounted Cumulative Gain (NDCG).

$$L^\theta_{\text{rank}}(s_t, \pi^*) = -\frac{\text{DCG}}{\text{IDCG}}$$

$$\text{DCG} = \sum_{w=1}^{W} \frac{dt(l_w) - 1}{\log(\text{pos}(l_w) + 1)}$$

The goal of NDCG is to give more importance to higher-ranked items and discount the value of lower-ranked items. NDCG is a metric commonly used in information retrieval systems to evaluate the relevance of ranked lists. The method observes the ranking eviction probabilities of each cache line that the model outputs and compares that with its reuse distance that Belady's calculates. Effectively, the policy (π_θ) is penalized heavily for placing probability on lines with low reuse distance and less penalized for high reuse distance lines.

To provide better generalization, the authors propose adding a second output to the existing neural network that predicts the reuse distance for each line. The input to the new head is the context embeddings (g_w) for each cache line. These embeddings capture information relevant to the cache line. The output is the predicted log-reuse distance for that particular cache line.

$$L^\theta_{\text{reuse}}(s, \pi^*) = \frac{1}{W} \sum_{w=1}^{W} \left(\hat{d}(g_w) - \log d\, t(l_w)\right)^2$$

The reuse distance loss function penalizes the policy for deviating from the true reuse distance, encouraging better predictions. During training, the model is optimizing for both the ranking loss and the reuse prediction loss.

$$L_\theta(s, \pi^*) = L^\theta_{\text{rank}}(s, \pi^*) + L^\theta_{\text{reuse}}(s, \pi^*)$$

By having both the policy head that decides what line to evict, and the reuse distance prediction head share the same network body, the model can leverage the shared representation to enhance training of the models' parameters, ultimately leading to better performance.

Codebase

The code for the system, developed by the authors, consists of a collection of Python code and JSON configuration files. The main components of the PARROT model are located in '*cache_replacement/policy_learning/cache_model*'. Below we detail the main functions and high-level code implementations of these main components.

Embedder

The code for the Embedder is found in the embed.py file which is located inside of the *cache_model* folder inside of *policy_learning*. The code is responsible for creating an embedder that is specified by the input into an embedding.

```
def from_config(config):
"""Creates an embedder specified by the config.
```

The function is responsible for selecting and creating the correct type of embedder. The function takes in config as an argument which specifies the type of embedder parameters. Depending on the specified type it will create a different embedder.

The Embedder class defines the structure for the specific embedders and defines the attributes. It is used to ensure that that all the subclasses in the code follows the correct structure.

```
class Embedder(nn.Module):
"""Embeds a batch of objects into an embedding space.
```

The rest of the file contains the classes for the different types of embedders which include ByteEmbedder, DynamicVocabEmbedder, and PositionalEmbedder.

```
class ByteEmbedder(Embedder):
"""Embeds each byte and concatenates."""
class DynamicVocabEmbedder(Embedder):
"""Dynamically constructs a vocab, assigning embeddings to new inputs.
class PositionalEmbedder(Embedder):
"""Takes position index and returns a simple fixed embedding."""
```

Byte Embedder

The code for the Byte embedder can be found in the *embed.py* file which is located inside of the *cache_model* folder inside of *policy_learning*. The code contains different classes used to handle the embedders in the program. They are used to handle inputs and break them up into smaller components. The code is also able to take the position index and build from that position. The ByteEmbedder class is used to create an embedding for each value. It has a constructor which takes in two inputs which are bytes_per_entry and embed_dim. Bytes_per_entry is the number of bytes each input should be broken down into and embed_dim is the size of the final embedding.

```
class ByteEmbedder(Embedder):
"""Embeds each byte and concatenates."""
def__init__(self,bytes_per_entry, embed_dim):
"""Embeds entries that have bytes_per_entry many bytes.
```

Each byte is separately embedded, and the embeddings are concatenated to form the final embedding.

LSTM

The code for the LSTM can be found in the *model.py* file which is located inside of the cache_model folder inside of policy_learning. It is ultilzed to track information about memory accesses over time. Information about the previous memory is tracked. Using PyTorch's neural network module, an LSTM unit is utilized.

```
self._lstm_cell = nn.LSTMCell(
pc_embedder.embed_dim + address_embedder.embed_dim,
lstm_hidden_size)
```

In the code snippet, the size of the embedding and the address are given to the unit to update the memory. The hidden size is how much space it has in the memory to store information. This is then utilized to help the model decide what memory data could be evicted based on the passed accesses.

```
next_c, next_h = self._lstm_cell(
torch.cat((pc_embedding, address_embedding), -1), hidden_state)
```

The LSTM unit is used to process the inputs and to update the memory. The outputs next_c and next_h are the new memory states which are used for the next input. The LSTM keeps track of the memory accesses and updates the memory to help the model make the best possible decisions. Each input updates the LSTM information about past actions, which is used to predict future actions.

Attention Mechanism

The code for the attention mechanism is in the *attention.py* file. It is implemented via multiple classes, each with sub definitions. Its base class:

```
class Attention(nn.Module):
```

contains the functions:

```
def forward(self, memory_keys, memory_values, queries,
masks=None):
  """Computes attention weights and the context vector"""
```

and

```
def _score(self, queries, memory_keys):
  """Computes the score function between queries and memory keys.
```

The *forward* function is the core function for a forward pass, in which it gets the score and calculates the SoftMax and context vector. Two types of scores are available. One in:

```
class ScaledDotProductAttention(Attention):
Score(q, k) = <q, k> / sqrt(dim(q))
```

and the other in:

```
class GeneralAttention(Attention):
```

that calculates the general attention score as discussed in *Luong et al. (2017).* The whole attention code has a wrapper:

```
class MultiQueryAttention(nn.Module):
 """Attention with num_queries queries per batch.
```

that takes the base attention mechanism and can process multiple queries at once. This is representative of how a transformers attention mechanism can calculated the attention in parallel.

Loss Function

This code defines custom loss functions in PyTorch for specific machine learning tasks, particularly focusing on ranking metrics like NDCG (Normalized Discounted Cumulative Gain).

def top_1_log_likelihood(probs):
This function calculates the negative log-likelihood of the top-ranked option (highest probability). The loss is computed as -log(p(top option)), where the top option is simply the one with the highest probability in probs[:, 0].

def approx_ndcg(scores, relevances, alpha=10., mask=None):
This function computes an approximation of the NDCG, which is a popular ranking metric. It is differentiable, which makes it suitable for training models using gradient-based optimization methods. The function approximates the position of each item based on the predicted scores using a differentiable sigmoid function. It then computes the DCG (Discounted Cumulative Gain) and IDCG (Ideal DCG), and finally returns the negative NDCG, which can be used as a loss function.

IV. Methodology

Conveniently, the authors of the cache replacement paper provided a GitHub repository, detailing steps to simulate their results and containing their code, as well as an OpenAI Gym environment for simulating their model. These resources are what we used for this paper. Additionally, as the workloads and training of the model was computationally intensive, we utilized NCSU's HPC cluster, Hazel. This cluster used IBM's LSF software for job management and allowed for the use of powerful GPUs for training and testing of the model.

The majority of the model was coded in Python, so our first step was to install all necessary Python packages that were used in the authors' code. The notable packages were Pytorch, which the model was built upon, and TensorFlow with tensorboard, which provided a simple method of model evaluation. The next step was to collect the traces that would be used to train and evaluate the model. Although it is possible to collect custom traces, we used pre-defined workloads from the 2nd Cache Replacement Championship, which were recommend for achieving similar traces to those that were used in the original experiment. As per the original authors' methodology, we then filtered our datasets to use traces only from the 64 cache sets which were used in the original experiment and split these into 80% training, 10% validation, and 10% testing.

The environment created by the authors allowed for an easy training process where a single Python script could be run with variable inputs, allowing for simple configuration of the datasets, cache configurations, aimed policy and loss functions, and various hyperparameters. For reproduction purposes, we maintained the default values for hyperparameters that we used in the final experiments of the original simulation. These hyperparameters are:

- Learning Rate = 0.001
- Address embedding length (d_m) = 64
- PC embedding length (d_p) = 64
- PC embedding vocab size (n_p) = 5000
- Position embedding length (d_{pos}) = 128
- LSTM hidden layer size (d_{LSTM}) = 128
- Frequency of recollection B = 5000
- History length H = 80
- Number of memory address: Variable for # of unique addresses seen in training for each workload

Once a model had been trained on a certain workload, a tensorboard was available to inspect the performance of the training. Upon satisfying training results, the model could then be evaluated and tested on its test set. This also generated a tensorboard file with a single parameter, *raw cache hit rate*, r. However, for evaluation the models performance in comparison to the standard cache replacement policy LRU and the oracle policy Belady's, a *normalized cache hit rate* is calculated. This parameter s defined as: $r_n = \frac{(r - r_{LRU})}{(r_{opt} - r_{LRU})}$, where r_{LRU} is the hit rate of the test set on the LRU policy and r_{opt} is the hit rate of the test set on the optical Belady policy. This normalization sets the LRU normalized hit rate as 0 and Belady's normalized hit rate as 1, where the normalized hit rate r_n represents the performance between these two extremes.

V. Results

We were successfully able to train and evaluate three workloads: astar_313B, bwaves_98B, omentpp_17B. The raw cache hit rates are shown in Table 1. Normalized cache hit rates are shown in Figure 11.

	astar	*bzip*	*omnetpp*
Optimal	38.9%	76.7%	92.5%
LRU	4.4%	60.5%	61.6%
Parrot	32.9%	65.2%	90.8%

Table 1. Raw cache hit rates for each workload under baseline and learned policies.

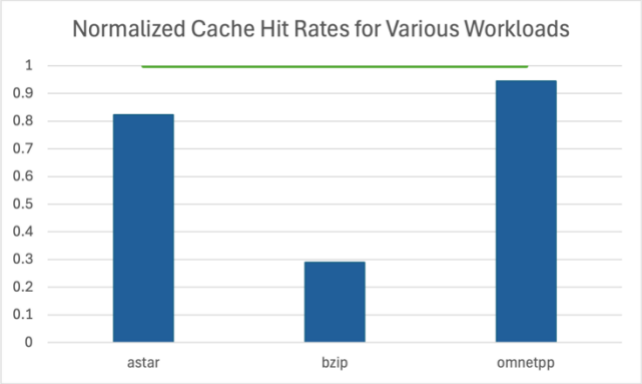


Figure 11. Graph showing normalized cache hit rates for the three simulated workloads. The green bar at the top represents the oracle policy Beladys, with a normalized cache hit rate of 1. The normalized cache hit rate of LRU is 0.

A total of six workloads were trained. Additionally, to those mentioned above, these consisted of bwaves_98B, GemsFDTD_109B, and lbm_94B. The training results for these workloads were inconclusive. The same job submission script and parameters were used to train these models, but each returned declining cache hit rates and non-sensical loss functions and resulted in evaluated cache hit rates of ~.001. The training raw cache hit rate and loss curves for the bwaves and lbm workloads are shown below.

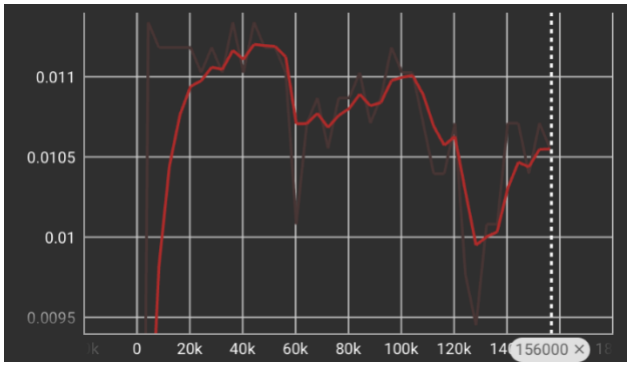


Figure 12. Raw cache hit rate for training on bwaves_98B.

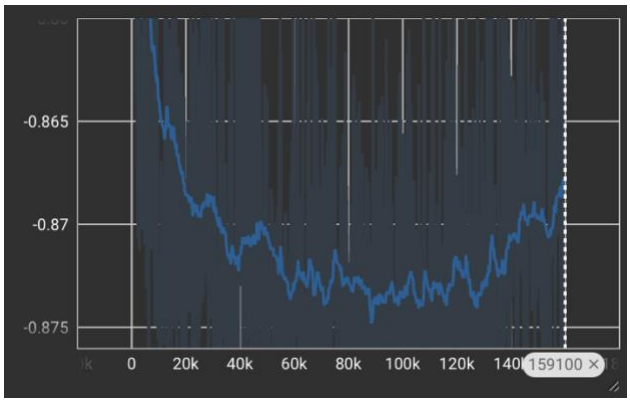


Figure 13. Training loss for bwaves_98B.

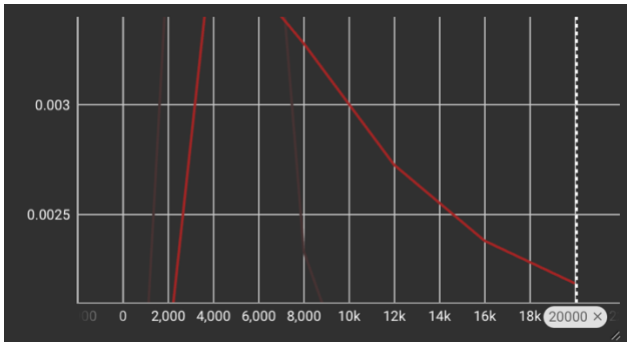


Figure 14. Raw cache hit rate for training on lbm_94B.

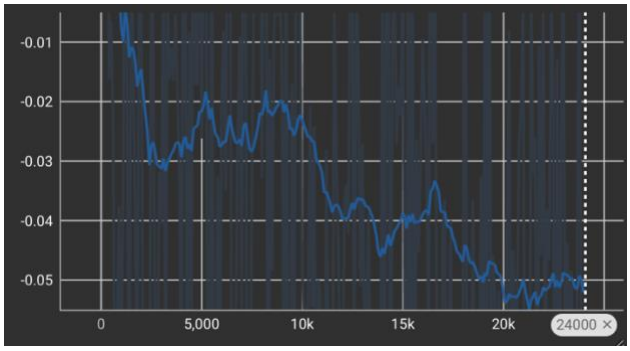


Figure 15. Training loss for lbm_94B.

VI. Discussion

After training and evaluating our workloads, in the least, they are not what we expected. Some of our results came close to the results of the original, but never exactly, such as the bzip workload results. What was rather consistent with the original results, however, was the relative relationship between the performance of the PARROT model on each workload. The order of best performance to least in the original experiment was omnetpp then astar, which were both relatively high, then bzip, which was quite low. This trend is also represented in our results. Besides this pattern, there is not much else about our results that resemble those of the original paper.

Possible reasons for this have been examined but, in general, confusion still remains as to the large deviation in our results and the originals exists. One specifically confusing result is the difference in the LRU simulation of workloads from the original baseline results. As LRU is a deterministic algorithm, it was expected that cache hit rates of this baseline would be extremely consistent. However, this is not the case, and discussion with peers has found that these cache hit rates are inconsistent across multiple iterations. Peer analysis has suggested that errors in extracting the workloads into CSVs has resulted in altered workload datasets, resulting in varying LRU baseline results.

Commenting on the three inconclusive workloads, the reason for these non-sensical trainings is unknown, as all parameters were kept equivalent to the workloads that generated comprehensible results. These results could possibly have stemmed from errors and challenges discussed above and below, or unknown errors in the model trainings that were not considered or discovered. Further work will need to be done for following assignments to understand this process.

We also acknowledge that we do not have a large quantity of workloads that we successfully trained and evaluated on. There are a couple of reasons to this, starting with the configuration of the given environment for simulation. Due to the environment using outdated Python packages in collaboration with a state-of-the-art HPC cluster, many dependency issues arose, and manipulation of the package versions and dependencies were required. Additionally, various errors in function calls and data types arose when initially simulating the model. These were recognized and mitigated as they were found. Although these difficulties were overcome, and the model was run correctly (to the best of our knowledge), these setbacks did reduce the time that would have been useful for training on workloads. These difficulties are not included as to excuse the state of our results, but rather to present a truthful discussion of our process and how our results are a product of it.

VII. Conclusion

Although the results did not exactly match those reported in the paper, we were successfully able to train and evaluate the PARROT model on certain workloads. Further work will need to be done to comprehend the full extent of this models nature. Overall, this provided a valuable understanding of the model and offered a stimulating learning experience for diving into a modern application of AI, as well as evaluating it on practical workloads.

REFERENCES

[1] G. Hinton, J. Dean, and O. Vinyals, Distilling the knowledge in a neural network, https://arxiv.org/pdf/1503.02531 (accessed Oct. 2, 2024).
[2] E. S. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, An imitation learning approach for Cache Replacement, https://arxiv.org/pdf/2006.16239 (accessed Oct. 2, 2024).
[3] A. Vaswani et al., "Attention is all you need," arXiv.org, https://arxiv.org/abs/1706.03762 (accessed Oct. 1, 2024).
[4] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," arXiv.org, https://arxiv.org/abs/1508.04025 (accessed Oct. 1, 2024).

APPENDIX

A. Contributions

Alex Tayyeb: I developed the overall diagram model, the DAgger algorithm model, and the ranking loss model along with the detailed descriptions. I also identified issues with python packages on windows machines and shared them with our class. I developed a solution to our home directories being filled up on the HPC servers and shared my solution with the class via the debugging paper Kaushal shared with us. My role mostly consisted of explaining the overall system in digestible terms and helping format the MDP process into a readable diagram that flowed nicely into our paper.

Loévan Bost: I developed the MDP components, Training process, and Byte embedder diagrams for the homework report, along with detailed descriptions. I also included the abstract, introduction, and background for the report. I helped set up the project environment using the HPC's and conda environment to execute the GitHub instructions and simulations. My role mostly consisted of explaining the training process and byte embedder while helping introduce the MDP components in plain language.

A.J Beckmann: My contribution to this homework included understanding, explaining, and creating diagrams for the neural network architecture, attention mechanism, embedding mechanism and general data flow of the system. I also debugged and configured the environment to be able to train, evaluate, and simulate the PARROT model and the baselines for analysis of our results. I helped my teammates along in this process. I also created and was heavily involved in the class discord to create a space for collective peer debugging and collaboration. Additionally, I helped formulate team meetings and stimulate progress of the project.

B. Meeting Logs

Our meeting statistics including dates, durations, attendance, and work done are documented here: https://oitncsu-my.sharepoint.com/:x:/g/personal/ajbeckm2_ncsu_edu/EVswNSUJHYtKiq_8jZL0IcoBAsNEv8HHKtx6C41w2PabzA?e=7CaUOW