Navigating Classic Atari Games with Deep Learning

Project Report

Presented to
The Faculty of the Department of Computer Science
San José State University

In Partial Fulfillment
Of the Requirements for the Class
CS 298

By
Ayan Abhiranya Singh
May, 2023
Navigating Classic Atari Games with Deep Learning

By
Ayan Abhiranya Singh

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE
SAN JOSÉ STATE UNIVERSITY
MAY 2023

Dr. Chris Pollett
Department of Computer Science

Dr. Mark Stamp
Department of Computer Science

Prof. Kevin Smith
Department of Computer Science
ACKNOWLEDGEMENTS

I would like to thank my project advisor, Dr. Chris Pollett, for his indulgent and patient discussions with me regarding the content and direction of this project - without which it would not have been possible to complete. I would also like to thank my committee members, Dr. Mark Stamp and Professor Kevin Smith for agreeing to be on my committee and for teaching the most illuminating courses I have taken while at San José State. Finally, I would like to thank my family, whom I will not individually name, but to whom I am grateful for many things beyond what I can put into words.
# TABLE OF CONTENTS

Abstract ................................................................................................................................. 1

I. Introduction .......................................................................................................................... 2

II. Background .......................................................................................................................... 5

   A. Policy Gradient Method .................................................................................................... 6
   B. Q-Learning ......................................................................................................................... 6
   C. Q-Function as a Bellman Recurrence Equation ............................................................... 7
   D. Loss Function .................................................................................................................... 8
   E. Model-Free ......................................................................................................................... 9
   F. Off-Policy ........................................................................................................................ 9
   G. Experience Replay ........................................................................................................... 9
   H. R2D2 ............................................................................................................................... 13
   I. Never Give Up ................................................................................................................ 14
   J. Agent57 ........................................................................................................................... 14

III. Neural Network Architecture and Experimental Setup .............................................. 17

   A. The Convolutional Neural Network ................................................................................. 17

IV. Experiments ....................................................................................................................... 20

   A. Deep Q-Learning with Input Lag .................................................................................. 20
   B. Baselining Agent Performance with Modifiers ............................................................... 23
      i. 2-Frame Cache with Coin Toss ..................................................................................... 24
      ii. 3-Frame Cache with Coin Toss .................................................................................. 24
      iii. 8-Frame Input Buffer ............................................................................................... 24
   C. Experiments to Converge on an Optimal Q-Function .................................................. 26
D. Boost DQN performance with Frame-Diff ............................................... 28
E. Transfer Learning on Different Mazes ...................................................... 32
F. Transfer Learning with User Gameplay ...................................................... 34

V. Conclusion .................................................................................................. 40

A. Guided Training .......................................................................................... 41
B. Game Modification ....................................................................................... 41
C. Transfer Learning from Similar Games ....................................................... 41

References ........................................................................................................ 43
ABSTRACT

Games for the Atari 2600 console provide great environments for testing reinforcement learning algorithms. In reinforcement learning algorithms, an agent typically learns about its environment via the delivery of periodic rewards. Deep Q-Learning, a variant of Q-Learning, utilizes neural networks which train a Q-function to predict the highest future reward given an input state and action. Deep Q-learning has shown great results in training agents to play Atari 2600 games like Space Invaders and Breakout. However, Deep Q-Learning has historically struggled with learning how to play games with greater emphasis on exploration and delayed rewards, like Ms. PacMan. In this project, we train a neural network that learns control policies for Ms. PacMan. We experiment with various methods to boost Q-agent performance. Through our novel training method, Frame-Diff, we are able to not only optimize, but beat, known benchmarks for reinforcement learning agents for Ms. PacMan by up to 50%.

**Keywords:** Deep Q-Learning, Video games, Atari, Reinforcement learning
I. INTRODUCTION

For my project, I trained a deep neural network (DNN) to play the classic Atari game, Ms. PacMan, using the Q-Learning reinforcement learning (RL) algorithm. The paper by Mnih et al. [1] from 2015 provides a foundation for the model I have developed for this project. [1] state that their model was able to reach superhuman levels on games like Space Invaders, Boxing, Star Gunner and Breakout while not being able to reach average human benchmarks on games like Ms. PacMan, Asteroids and Montezuma’s Revenge. In this project, I aim to leverage the research behind the model in [1] to build an agent that can play Ms. PacMan. The work done as part of this project also acts as a test for the robustness of the model described in [1], by testing it under various simulations of stress, such as latency or lag. For example, one application of such a model could be to train an agent to play Atari games by streaming video over an internet connection, which often suffer from latency and stuttering.

DNNs can be used to interpret game frames as a set of raw pixels to learn a control policy that achieves the maximum amount of reward [1]. The policy dictates which action an agent should take to maximize its reward given a specific state within the game, i.e., a set of pixel values. The reinforcement learning feedback loop works such that an agent is presented with an observation of the environment at a given time step. The agent takes an action on the environment based on this state and the environment returns a reward. Mnih et al. [1] attempt to learn control policies for classic Atari 2600 video games leveraging a take on the Q-Learning algorithm, called “Deep Q-Learning”. They use a neural network, called a “Deep Q-Network” (DQN), to learn an optimal rewards function for the Atari agent. This algorithm makes clever use of what is known as an “experience replay” buffer, to minimize loss on the Q-Function, which predicts the total future reward. The Q-Learning algorithm, the Deep Q-Learning algorithm and
the experience replay buffer are all explained in depth in Chapter II. The Deep Q-Learning algorithm proposed in their paper was used to train an agent to master games like Space Invaders, Breakout, Pong, Beam Rider and Seaquest. However, the original paper still struggled to provide comparable performance on certain games like Montezuma’s Revenge, Asteroid, Private Eye and Ms. PacMan. The reason for this is the exploitation (using the DNN) versus exploration (going off on a new path) tradeoff. As some games drag on for longer, the need for exploration might actually increase. The agent they trained was reported to score 2311 on Ms. PacMan [18].

DQN was followed by R2D2 [22] in 2019, a distributed computing reinforcement learning algorithm that was able to beat the DQN benchmarks by a huge margin. This model used histories of observations instead of just the observation of a state and the next action. As a result, this is a far more compute and memory intensive learning model than DQN. This agent was able to score up to an average of 44281.7 on Ms. PacMan, which is well above human average.

Never Give Up [23], published in 2020 by DeepMind, was the successor to R2D2 and the direct predecessor to Agent57 [14]. This algorithm experiments with introducing rewards outside of those naturally found in the environment, to encourage the agent to explore previously unexplored states in the game. While there is no published benchmark on Ms. PacMan it is reported that it was able to reach better-than-human levels on all 57 Atari games.

Agent57 [14], published in 2020, was DeepMind’s follow up to their own DQN. It amalgamated the DQN with the distributed approach of R2D2 and Never Give Up. Curiosity, represented as an “exploration rate” was added to the Q-Function as an adjustable parameter. Overall, it was able to outperform both and scored an average of 63944.44 on Ms. PacMan.
While R2D2, Never Give Up and Agent57 expand the performance benchmarks beyond that of DQN for Atari games, it is important to note that all three approaches require high-performance computing (HPC) and are heavily reliant on distributed worker agents. Due to this, and the fact that this paper is a single-researcher effort, the DQN Ms. PacMan score is the benchmark that will be aimed for with the neural network developed in this project.

The organization of the rest of this paper is as follows: the next chapter discusses [1] in depth and provides a comprehensive explanation of the theory behind Q-Learning, as well as more discussion on popular RL agents today. The following chapter, Chapter III, is focused on the architecture of the neural network we implement for this project and the experimental setup. Chapter IV describes all the experiments conducted on our neural network – ranging from input lag, extending the state buffer, adjusting the learning rate to converge on an optimal Q-Function, and transfer learning. Of note is our novel training technique, Frame-Diff, also discussed in Chapter IV which provides improved results over the DQN. Finally, Chapter V comprises the conclusion to this project, summarizing the learnings and offering insight on potential for future research in this field.
II. BACKGROUND

Mnih et al. [1] published their paper regarding Deep Q-Learning for Atari games in 2013. This is one of the seminal works in the field of modern reinforcement learning. Essentially, they developed the first deep learning model that was able to learn efficient control policies for classic Atari games from high-dimensional input, i.e., image frames from the game. The original paper published by DeepMind was able to learn control policies for 7 games, including Space Invaders and Breakout. For our project, we built an agent that can play Ms. PacMan. Our model leverages Deep Q-Learning as well, so a comprehensive understanding of this algorithm is required before we proceed with our implementation.

The underlying issue that deep reinforcement learning is trying to solve is that of credit assignment, that trials of these games (or episodes) usually run very long (sometimes to the order of tens of thousands of frames) and the reward returned, per-frame, often becomes very sparse. Credit assignment is a big challenge under this condition, i.e., learning what the right action should be given an observation. To solve this problem, Mnih et al. propose Deep Q-Learning, which involves a combination of the Q-Learning algorithm with convolutional neural networks (CNNs).

Figure 1. A sample frame of Ms. Pac-Man on Atari 2600, rendered via the Arcade Learning Environment [1]. The reward at this terminal state of an episode would be the total reward earned thus far, i.e., 1700 points.
The idea behind their approach is to record observations and rewards at each time step. The Q-Agent interacts with the game’s environment through actions, which modify the environment and return an observation and a reward (which could be zero). Formally, they want to maximize that reward value at each time-step, $t$. Additionally, we want to discount future rewards with a discount factor, $\gamma$, that signifies that rewards in the future are not as important as rewards earned immediately or near-immediately. Given an action $a$ and state $s$ at time $t$, the total future rewards from time $t'$ onwards, $R_t$, is given by the equation

$$ R_t = \sum_{t'=t}^{T} (\gamma^{t'-t} r_t) $$

This is the value which needs to be maximized under the definition of this problem. There are two approaches to maximizing the future reward:

A. Policy-Gradient Method:

This approach entails developing a policy gradient function that outputs an action based on a given state. A neural network can be trained to do this by providing observations as input and having as many output neurons as valid actions for the problem. This sets a good baseline for the Q-Learning approach for future rewards maximization.

B. Q-Learning:

This approach entails training a Q-function that accepts an input combining state and action and outputs the predicted reward if action $a$ is taken at state $s$. The function $Q(s, a)$ is given by:

$$ Q(s, a) \rightarrow R $$

Where $R$ is the total future reward after an action $a$ has been taken [5].
It follows that a perfect Q-function will exactly predict the reward from a given time step till the end of the episode. Q-functions are essentially conditioned on the policy itself. A neural network architecture similar to the one described above for the Policy Gradient approach could be used for this problem. The network accepts the state of the game as input and then provides an estimation of future rewards as output. The state of the game is represented with pixel values and the output is the Q-value for each legal action in the environment. By maximizing the Q-value choice of action at each time step, we can play to maximize future rewards for the agent. However, the drawback with this approach is that the neural network must be called for every action.

Since the perfect Q-function does not exist in practice, the Q-function for this neural network must be trained. Minh et al. [1] propose a take on an online reinforcement learning algorithm to do this. They postulate that, given a state $s$, the Q-function can be estimated by simply playing through an episode and recording all the rewards. While the game is being played, the DNN is trained at the same time. Over time, if this training process is conducted iteratively, Minh et al. propose that the Q-function will converge upon an optimal Q-function for the given environment [1].

C. Q-Function as a Bellman Recurrence:

The Q-function models the Bellman recurrence equation. This means that the Q-function for a state $s$ can be represented in terms of Q functions for the successive state, $s'$, and onwards. The star policy, $Q^*(s, a) \rightarrow R$, returns the highest Q-value possible given an action and state. This means the agent performs action $a$ and then plays optimally henceforth. Per Russell and Norvig [5], the entire star policy can be defined as:
\[ Q^*(s, a) = \mathbb{E}_{s' \sim \epsilon} \left[ r + \gamma \max_{a'} Q^*(s', a') \mid s, a \right] \] (3)

Essentially, the equation describes the agent in state \( s \), performing action \( a \) to get reward \( r \) and state \( s' \). In state \( s' \), provided the optimal action is performed by following the star policy, the same equation holds true for the next state of the game’s environment. Thus, we have a recurrence relation where the optimal policy \( Q^*(s, a) \) may be framed in terms of \( Q^*(s, a) \) for subsequent states. This recurrence relation lays the basis on which we can train the neural network.

**D. Loss Function:**

The loss function for the neural network is defined as such, per [1]:

\[ L_t(\theta_i) = \mathbb{E}_{s, a \sim \rho_i} \left[ (y_t - Q(s, a; \theta_i))^2 \right] \] (4)

\( Q(s, a; \theta_i) \) represents the Q-function for state \( s \) and action \( a \). It outputs the predicted total future reward for that action, i.e., the predicted total reward from that time step till the end of the episode. \( y_t \) is the expected reward after playing the entire episode. From Equation (1), we know that the total future rewards can be represented as a function of the current reward plus the reward obtained for all subsequent time steps. This intuition can be used to compute the “true” labels for the loss function. The parameters from one time step prior could be used to obtain a “true” label for the current time step and then compute the loss as squared error between that label and the Q-function for the current time step. This implies that

\[ y_t = \mathbb{E}_{s' \sim \epsilon} \left[ r + \gamma \max_{a'} Q^*(s', a' ; \theta_{t-1}) \mid s, a \right] \] (5)
is the target reward for time step $i$ [1]. $\rho(s, a)$ is a probability distribution over the states $(s, a)$, known as the behavior distribution. Thus, the DQN itself is used to compute both true and predicted labels to compute the squared loss in this deep reinforcement learning algorithm.

**E. Model-Free:**

Additionally, [1] state that this algorithm is model-free. This means that the algorithms itself simply learns a function that outputs a Q-value for each state-action tuple. There is no interaction with external components like the model or an emulator, to learn specifics about the reinforcement learning task.

**F. Off-Policy:**

The model learns what is known as the $\epsilon$-greedy strategy:

$$ a = \max_a Q(s, a; \theta) $$

(6)

While following a behavioral distribution that allows the agent to explore the environment to a reasonable degree. The agent will randomly explore the environment with $\epsilon$ degree of probability and request Q-values from the DNN the remaining $1 - \epsilon$ amount of time. This helps mitigate exploitation of the neural network and can speed up the training phase by reducing the number of calls to obtain Q-values.

**G. Experience Replay:**

Mnih et al. [1] find that successive frames across the game’s environment are often extremely similar and correlated. In Chapter IV, we experiment to calculate the actual time between frames, which works out to around 0.08 seconds. For this reason, they propose that instead of using data samples as they are generated, a better approach to boost training performance might be to store observations, actions and rewards in an “experience replay buffer”
and samples those at random. Inherently, this implies some samples might never get selected for play, while others might get selected more than once. A good analogy for this is the process of randomly selecting or shuffling data prior to training a model under supervised learning.

The input state to the neural network comprises of a buffer of 4 frames (observations). These images are concatenated together because multiple images often carry more information about the state of the game, such as trajectories for agents and ghosts. Additionally, this helps mitigate the flickering issue that many classic Atari games suffer from. The Q-function is initialized to random weights. Following the $\epsilon$-greedy strategy from above, the agent either takes a random action with probability $\epsilon$, or the model suggests the action to be taken that has the highest future reward. Once this action is executed, the next state of the game’s environment is obtained, with a new observation and reward. This transition, from state $s_t$ to state $s_{t+1}$ is stored in the experience replay buffer.

This buffer is randomly sampled periodically. Since the selection of state transitions is random, the model is receiving loosely correlated states as its input tensor. If the state transition was a terminal state, then the target reward label for training the model is the last reward received. If it isn’t, then the target reward should be the last reward received, plus the total predicted future reward from the Q-Function. The Deep Q-learning algorithm, per [1] is documented in the diagram below.
Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \( \mathcal{D} \) to capacity \( N \)
Initialize action-value function \( Q \) with random weights

for episode = 1, \( M \) do
  Initialise sequence \( s_1 = \{ x_1 \} \) and preprocessed sequenced \( \phi_1 = \phi(s_1) \)
  for \( t = 1, T \) do
    With probability \( \epsilon \) select a random action \( a_t \)
    otherwise select \( a_t = \max_{a} Q^*(\phi(s_t), a; \theta) \)
    Execute action \( a_t \) in emulator and observe reward \( r_t \) and image \( x_{t+1} \)
    Set \( s_{t+1} = s_t, a_t, x_{t+1} \) and preprocess \( \phi_{t+1} = \phi(s_{t+1}) \)
    Store transition \( (\phi_t, a_t, r_t, \phi_{t+1}) \) in \( \mathcal{D} \)
    Sample random minibatch of transitions \( (\phi_j, a_j, r_j, \phi_{j+1}) \) from \( \mathcal{D} \)
    Set \( y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases} \)
    Perform a gradient descent step on \( (y_j - Q(\phi_j, a_j; \theta))^2 \) according to equation 3
  end for
end for

Figure 2. The Deep Q-learning algorithm, per [1], combining the Experience Replay approach.

The training approach proposed by [1] was a general form of reinforcement learning that was applied to 7 Atari games. The agent is re-trained on each of the games, however, the base model or neural network architecture is not altered. Of even more curious note is that the neural network architecture utilized by them is not extremely complicated, particularly when compared to popular convolutional neural networks (CNNs) that are used today, like VGG16 [15], ResNet-152 [16] and Inception-v4 [17].

Minh et al. [1] evaluate their model by conducting experiments on the Atari games Beam Rider, Breakout, Enduro, Pong, Q*bert, Seaquest and Space Invaders. The results for their experiments on Breakout and Seaquest are documented below, in Figure 3.
Breakout is probably considered the single biggest success of the DQN. While the agent does show improvement on Seaquest over an increasing number of training epochs, the agent does not perform as well as on Breakout, overall. The agent for Breakout is particularly memorable because it was able to learn an actual efficient human strategy - that of breaking a small hole in the wall, pushing the ball through, and just letting it destroy the wall from the other side - just through the network and experience replay. While they did achieve fantastic results on Space Invaders and Breakout (among others), they were unable to produce similar results on other games like Montezuma’s Revenge and Ms. PacMan. The results for the rest of the games that they tested the DQN on are shown below, in Figure 4. Those games place more emphasis on exploration and taking “risks”, while the rewards get sparser and sparser as the game progresses.
H. R2D2

Kapturowski et al. [22] published their paper on the R2D2 distributed algorithm in 2019. This algorithm, an upgrade over the DQN, leveraged long short term memory (LSTMs) and gated recurrent units (GRUs) to enhance the performance of their agent. Essentially, they pass a history of observation-action tuples as input to the DNN. The distributed computing aspect modifies the experience replay to aggregate learning from multiple worker units. Again R2D2, while a level up on the base DQN, still struggles somewhat with risk taking and exploration.
I. Never Give Up

Badia et al. [23] published their paper on the Never Give Up algorithm in 2020. This algorithm aims to model "curiosity" in the agent to motivate it to take "risks". They do this by adding rewards of their own making, dubbed "intrinsic rewards", to the environment to push the agent to behave a certain way or to take specific actions under some given conditions. The reason this is called "risk-taking" is that the agent, under these conditions, might ignore immediate reward to instead explore states that it has not encountered before and thus create new memories for the experience replay. Like [23], this method also relies on distributed training, and is an example of model parallelism. Never Give Up proposes an interesting take on the long-credit assignment problem, however, it sacrifices the established performance baseline for the games that the DQN was able to master completely.

J. Agent57

DeepMind followed up the DQN with Agent57 [14], a new RL agent that could achieve superhuman performance on all 57 classic Atari 2600 games. Per Badia et al. [14], they aimed to solve the sparse reward problem and the exploration/exploitation conflict with this paper. Agent57 [14] improves upon Never Give Up, by proposing a means to learn the correct exploration versus exploitation policies for each game. The first modification they suggest is to parameterize the Q-Function according to extrinsic rewards and intrinsic rewards. An exploration rate factor, $\beta$, is applied to intrinsic rewards to adjust for whether you prioritize intrinsic or extrinsic rewards in the current game. For example, prioritizing intrinsic rewards by setting the
exploration rate factor high could set the agent off on taking more risks instead of prioritizing short term rewards. The modified Q-Function equation is as follows:

\[
Q(x, a, j; \theta) = Q(x, a, j; \theta^e) + \beta_j Q(x, a, j; \theta^i)
\]  

(7)

Where \(\theta^e\) and \(\theta^i\) represent the extrinsic and intrinsic components of the Q-function, respectively. The weights for both components parameterize two separate neural networks that that share their overall architecture. In this manner, the extrinsic and intrinsic reward parameters are optimized separately, even though the overall goal to maximize total future reward remains the same [14].

The second modification is regulating the discount factor, \(\gamma\), which was used in the original DQN to increase or reduce the value of future reward in the game. Setting this value low is ideal for games which prioritize immediate rewards, while setting it high would imply that the agent should play the "long game" so to speak, and attempt to look for future rewards as much as immediate rewards.

Thus, there could be different permutations of these two factors that optimize the agent's performance on a given game. Agent57 does exactly this - it adjusts these factors over the course of training to converge on an optimal solution. They propose that a meta controller overlooks the agents as it interacts with its environment. The agent's observation-reward cycle is much the same as in the DQN, however, in this case the meta-controller modulates the exploration rate factor and discount factors as the training progresses. This itself turns into a reinforcement learning problem similar to the DQN, which aimed to maximize future rewards with each action. In this case, the meta-controller looks to maximize future reward by curating its selection of the exploration rate and discount factors. Finally, they do report a score of 63994.44 on Ms. PacMan, which obliterates the previous marks of any agent till date.
The next chapter draws from the Deep Q-Learning algorithm described in this chapter and covers our implementation of the DQN as well as the learning algorithm.
III. NEURAL NETWORK ARCHITECTURE AND EXPERIMENTAL SETUP

Our initial working hypothesis for this project is to train a deep learning model, that is able to maximize its score on Ms. Pac-Man or win a maze outright. The approach will be to build a DQN that takes inspiration from the model described in [1] and then apply that to the game of Ms. PacMan.

The neural network for this task has 3 layers, each fully connected to the next. The first 2 layers both contain 32 neurons, with the first layer accepting an input of the game screen (n x n grid). The final fully connected layer has 5 outputs, corresponding to the actions that an agent can take within the environment (no-operation, up, down, left or right).

A. The Convolutional Neural Network

Convolutional neural networks (CNNs) are neural networks that work great with problems involving learning local structures within the input, for example, object detection in images. The CNN consists of convolutional layers, which act as filter matrices of some set size, where the elements of the filters are the weights for that layer. These filters then convolve across the image at a set stride, compute dot products with the original image and output a new matrix, known as the feature map. These convolutional layers are usually followed by an activation function, which determines what the final output will be from one neuron in the network to the next, or, in some cases whether the neuron is on or off. The ReLU (rectified linear unit) activation function is a popular activation function that has a gradient of 1 and determines whether the neuron is on or off based on whether the input is positive or not.

As an example, the first convolutional layer might take an image as input. The filter applied on it may extract simple shapes like lines, diagonals or triangles. Subsequent convolutional layers will then be applied on the output from the previous convolutional layer. In this manner,
the CNN can extract increasingly abstract features from the input image. The final layer of the CNN is fully connected and enables it to make classifications.

The filter weights and biases of the CNN are learned as part of the training. Since these convolutions or filters are usually initialized randomly the CNNs could potentially learn different things each time they are trained.

The network implemented is a convolutional neural network and consists of:

1. 3 convolutional layers:
   
i) 32 filters of 8x8, stride of 4, activated by ReLU
   
ii) 64 filters of 4x4, stride of 2, activated by ReLU
   
iii) 64 filters of 3x3, stride of 1, activated by ReLU

2. 2 dense layers:
   
i) 512 neurons, activated by ReLU
   
ii) 5 neurons (size of the action space for the agent), with a linear activation function

The model for this project was developed in Python and the Keras API in TensorFlow was used to implement the deep neural network described above. The training dataset is composed of frames of images from playing Atari games using the ALE.

The hardware specifications are as follows. The model is trained on a MacBook Air M1 with 8 cores (4 performance and 4 efficiency) and 8 GB of LPDDR4 memory. TensorFlow executes computations on the M1 GPU, which has another 8 cores and supports the Apple Metal graphics optimization technology.

Per Mnih et al. [1], the network takes the raw pixels of the game screen as input. Our approach to training the agent is much the same as the approach outlined in the DeepMind paper. Per Bellemare at al. [4], the Arcade Learning Environment (ALE) can be used to simulate the Ms
PacMan game within a Python kernel. The OpenAI Gym library [3] provides a wrapper that enables communication between the neural net and the environment rendered in ALE. The ALE wrapper for Ms Pacman allows for actions to be provided on a frame-by-frame basis. Once an action is played in the simulated ALE environment for Ms. PacMan, an observation (the next frame of the game) is returned. We use the OpenCV library to convert this frame to greyscale and reduce its size to 84 by 84 pixels. The reduction to greyscale gets rid of any noise the network would have to otherwise deal with in case there were colors in the game and the reduced size speeds up the training process. A deque is used to maintain a queue of the latest 4 frames of the game. This queue is provided to the neural net as input when training and to obtain predictions on the optimal move to play. As an expansion of the $\varepsilon$-greedy strategy described in the previous chapter, we also utilize an $\varepsilon$-decay parameter. This parameter reduces the exploration probability, $\varepsilon$, as the amount of frames that the agent has been trained on increases. This parameter essentially reduces the amount of random exploration the agent does over time, as the agent’s Q-function converges further towards optimal. Figure 5 below outlines the flow of constructing the input buffer for the Deep Q-Network.

![Diagram](image.png)

Figure 5. Action-observation flow.
IV. **Experiments**

Over the course of this chapter, we discuss the various experiments conducted on the agent to measure performance under lag, extending the state buffer, improving performance with the Frame-Diff approach and transfer learning. The experiments with lag are intended to set baselines for what to expect with the different forms of transfer learning we try, while the Frame-Diff approach utilizes our findings of high correlation between frames within a small time quanta. The experiments also provide us a general idea of the scalability of the model we have developed.

**A. Deep Q-Learning with Input Lag**

One use case for an AI agent like this might be to deploy it over an internet connection to play similar games or even different mazes within PacMan. A setup like this would involve two major concepts to incorporate into the existing Q-Network solution, namely, those of input lag and transfer learning. Later, in this project, we discuss training an agent with user gameplay videos. Most scenarios involving processing video input or even multiple frames over the internet would require some sort of exploratory analysis into packet loss and stuttering, which explains our motivations for the experiments in this section.

To simulate lag within the program environment, we first need to experiment with the frames of Ms. PacMan running on the ALE [4]. The ALE is a framework that renders classic Atari 2600 games as sandbox environments to test reinforcement learning algorithms such as what we intend to do. OpenAI’s Gym [3] API provides a wrapper over these games, which enables development of agents that can interact with them to produce new observations. The env.reset() function within Gym resets the environment (game) to the initial state (the start of the game). The input buffer, for the Q-learning algorithm consists of a deque of frames of size 4. This deque
is initialized by duplicating the first frame 4 times. A snapshot of the input buffer after playing one action is shown below (Figure 6). The env.step(action) function plays a move within the environment and returns a single observation, i.e., 1 frame per step. At each time step, the input buffer contains the last k number of frames, meaning it pops the oldest and pushes in the new frame. Simply put, the step() function plays one action and then appends the latest frame to the current queue of frames.

![Input state buffer after executing one action](image)

Figure 6. Input state buffer after executing one action. A single new frame is appended to end of the queue.

Measuring the time elapsed per frame is of particular importance in this experiment as that is the single controllable parameter that measures how much of lag we are inserting into the simulation. The approach is to time gameplay in the Stella Atari 2600 emulator (which is the emulator that powers the ALE rendering of Atari games) versus how many frames the agent in the Gym environment takes to move the same distance. The initial state of the Ms. PacMan level 1 maze is shown below (Figure 7).

![Initial state of Ms. PacMan](image)

Figure 7. Initial state of Ms. PacMan, rendered on ALE.
For example, it takes 3.70s to move from the bottom left of the screen to the bottom right (45 frames, works out to around 1 frame per 0.0822 of a second). It takes 5.70s from game launch to reach the first left wall (68 frames, around 1 frame per 0.0853 of a second). Similarly, the agent is measured moving across the top and other various sections of the map and timed against actual gameplay in Stella. One frame comes to represent approximately 0.085 seconds of gameplay. Thus, if we simulate a frame skip of 2 frames, we are effectively simulating a loss of 0.17 seconds of user gameplay.

In this preliminary experiment, input is provided to the net in the same manner as training. If frame 1 is the observation for the first time step and frame 2 is the observation for the second time step, then the progression of input to the net (the input buffer) looks like [1111] → [1112] → [1123] → [1234], and so on. The agent is trained over 5000 episodes, working on the reduced move set (NO-OP, UP, DOWN, LEFT, RIGHT). Moves like UPRIGHT, UPLEFT, etc, which make more sense on a joystick control, are not considered as valid actions for this agent. Under the conditions above, the agent scores 850 points, a decent return after training the net for 21 minutes and 35 seconds. This sets a baseline model for the next experiment.

The next experiment is to observe model performance under basic conditions of latency. In this experiment, we duplicate frames for every pair of 2 moves. For example, if frame 1 is the observation for the first time step and frame 2 is the observation for the second time step, then the progression of input to the net looks like [1111] → [1122] → [2233] → [3344], and so on. This essentially mimics a set up where the agent receives new information (a new observation of the game’s environment) every 0.17 seconds, as opposed to 0.085 in the base model. The agent is again trained over 5000 episodes, working on the reduced move set (NO-OP, UP, DOWN, LEFT, RIGHT). Under the condition of latency, the agent scores 530 points, a considerable drop-
off of 37.65% from the previous score of 850 which was obtained by the base model. This tells us that the agent needs training for more episodes and perhaps with a “gap” between frames as an adjustable training parameter.

**B. Baselining Agent Performance with Modifiers**

Frame skipping is an important hyperparameter used in training the agent to play Atari games. The frame skip value represents the number of frames that should be “skipped”, or ignored, between the agent’s actions, during the training phase. Our motivation behind selecting a frame skip value is that it speeds up training the agent, since the time between frames is quite small. Frame-skipping could potentially also improve performance with an input at an inconsistent frame rate, as playing through every successive frame could result in frames without a reward signal (when the agent is moving in between pieces of food), which becomes noise as far as training the agent is concerned. Mnih et al. [1] experimented with a Frame-Skip parameter of 4, i.e., 4 frames between actions – when they trained their DQN to play games like Breakout and Space Invaders. Of note is the score reported for an agent playing Ms. PacMan but taking actions randomly instead of using a Q-Function. In this set-up, the agent is able to score an average of 307 points [18] per trial. This is the average we hope to beat over our experiments, as this would demonstrate an agent that is learning over time and playing better than simply moving randomly across the board.

Choosing a frame skip value of 4 seems like a good place to start for the purposes of Ms. PacMan. To set a baseline for the experiments being conducted, an agent is trained over 5000 episodes, with a frame-skip parameter of 4. Over 10 trials, the agent scores an average of 902 points. This agent will serve as the baseline for the following experiments.
i. 2-Frame Cache with Coin Toss:

To study the performance of the agent under various performance modifiers, the performance of the agent may be analyzed with a random selection of frames. In this experiment, a coin is tossed every 2 frames to decide whether a new frame should be selected to update the input buffer. Between coin tosses, frames that are generated (but not added to the input buffer) are cached. Based on the result of the coin toss, either the first or the second frame is selected. Under these conditions, selecting 1 of every 2 successive frames, the agent scores an average of 492 points over 10 trials. Thus, there is a degradation of 45.45% with an input “stutter” of 1 out of every 2 frames.

ii. 3-Frame Cache with Coin Toss:

The same experiment is repeated for a cache of 3 frames, i.e., a cache of 3 frames is stored; every third moves 1 of the 3 frames is added to the input buffer. Each move has 33% probability to be chosen. Over 10 trials, the agent scores an average of 650 points under these conditions. Both the 2-Frame cache and 3-Frame cache experiments are conducted on a 4-Frame input buffer.

iii. 8-Frame Input Buffer:

Finally, to experiment with obtaining an improvement in agent performance without increasing the training episodes, we increase the size of the replay buffer. The size of the replay buffer is increased from 4 frames to 8 frames, with no other modifications. Over 10 trials, the agent scores an average of 804 points, which is a drop of 10.86% from the base model.

The results of these experiments are promising. The base model with a 4-frame replay buffer
outperformed the other models on average over 10 trials. This is expected as this is model is providing predictions on frames at exactly the same rate as it was trained. Among the modified models, model 3 (8 frame replay) appears to win out. Again, the drop in performance is expected as the frame rate is inconsistent. However, the agent is still able to play the game and despite the loss in signal, is able to score an average of 650 points (in the case of the 3-frame cache). Interestingly, the 3-frame cache performs better than 2-frame cache even though the chances of updating the replay buffer in the former is less. Figure 8 below maps the performance of the 3 experiments with the baseline model, over 10 trials.

![Comparative analysis of models](image_url)

**Figure 8.** Comparative analysis of the preliminary experiments with the baseline model.

We observe that the basic 4-Frame replay buffer performs better than the 8-frame replay buffer, by around 100 points, on average. The results for the 8-Frame buffer are still important, as they set a baseline for us to conduct experiments on learning rate. We can test various learning rates with the 8-Frame buffer as the 8-Frame buffer is not showing a major drop in the agent’s
average score from the 4-Frame buffer and can help to set a minimum baseline for performance.

Table 1 below documents the results of each of the experiments above, over 10 trials.

Table 1. Comparative analysis of the preliminary experiments with the baseline model.

<table>
<thead>
<tr>
<th></th>
<th>3 Frame Cache With Coin-Toss</th>
<th>2 Frame Cache With Coin-Toss</th>
<th>8-Frame Input Buffer</th>
<th>4-Frame Input Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial 1</td>
<td>470</td>
<td>510</td>
<td>1310</td>
<td>590</td>
</tr>
<tr>
<td>Trial 2</td>
<td>370</td>
<td>470</td>
<td>870</td>
<td>1740</td>
</tr>
<tr>
<td>Trial 3</td>
<td>810</td>
<td>440</td>
<td>730</td>
<td>700</td>
</tr>
<tr>
<td>Trial 4</td>
<td>990</td>
<td>470</td>
<td>350</td>
<td>970</td>
</tr>
<tr>
<td>Trial 5</td>
<td>540</td>
<td>470</td>
<td>600</td>
<td>440</td>
</tr>
<tr>
<td>Trial 6</td>
<td>450</td>
<td>660</td>
<td>710</td>
<td>1050</td>
</tr>
<tr>
<td>Trial 7</td>
<td>430</td>
<td>550</td>
<td>710</td>
<td>590</td>
</tr>
<tr>
<td>Trial 8</td>
<td>1140</td>
<td>520</td>
<td>1140</td>
<td>1300</td>
</tr>
<tr>
<td>Trial 9</td>
<td>770</td>
<td>520</td>
<td>750</td>
<td>740</td>
</tr>
<tr>
<td>Trial 10</td>
<td>530</td>
<td>310</td>
<td>870</td>
<td>900</td>
</tr>
<tr>
<td>Average</td>
<td>650</td>
<td>492</td>
<td>804</td>
<td>902</td>
</tr>
<tr>
<td>STDEV</td>
<td>262.7630957</td>
<td>88.6691729</td>
<td>269.0394436</td>
<td>389.0672607</td>
</tr>
</tbody>
</table>

C. Experiments to Converge on an Optimal Q-Function

Through stochastic gradient descent, we aim to minimize the loss function over the iterations of the training process. The learning rate dictates how large of a “step” is taken towards the minimum. For the DQN training process, since we aim to learn an optimal Q-function, the gradient descent is minimizing loss on the Q-function. By changing the learning rate, we change the weight updates per iteration of the training process. The goal of the experiments in this section is to find an adequate learning rate to converge upon the optimal Q-function, thereby ensuring an efficient training process and high-performance agent.

Since the 8-frame buffer appears to be performing well, we run some experiments to see how robust the model really is. There are a few reasons why we adjust the learning rate. One of the goals is to improve performance of the agent with an 8-frame input buffer, by increasing the
learning rate while keeping the number of episodes trained for constant. Additionally, we explore the effect of learning rate on the size of the input buffer, and whether there is a more ideal buffer we can use other than the 4-frame buffer we have been using thus far.

Separate versions of the agent are trained by varying the starting learning rate from 0.0001 to 0.00025. Figure 9 below outlines the performance of the agent under each.

![Performance of 8-frame replay buffer](image)

**Figure 9.** Comparative analysis of the preliminary experiments with the baseline model.

The learning rate of 0.00025 still performs the best. Gains when increasing the learning rate beyond 0.00025 pertaining to average agent score were marginal at best, while dropping the frame rate below 0.0001 was causing the agent to do nothing meaningful and just score 70 points by moving to a corner and then cease to move. The agent had its single best run at learning rate 0.001. However, the other 9 trials were too close to the random-action baseline for this learning rate to be considered the best. Additionally, the standard deviation was the highest for this learning rate, which implies the agent’s performances were not consistent at all. The overall best performance over 10 trials is at learning rate 0.00025. Table 2 below documents the performance of the agent over multiple learning rates, each agent being trained for 10,000 episodes.
Table 2. Varying starting learning rate with an 8-frame input buffer.

<table>
<thead>
<tr>
<th>Trial</th>
<th>5k eps, LR=0.001</th>
<th>5k eps, LR=0.0001</th>
<th>5k eps, LR=0.00005</th>
<th>10k eps, LR=0.0001</th>
<th>5k eps, LR=0.000025</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial 1</td>
<td>330</td>
<td>550</td>
<td>70</td>
<td>390</td>
<td>1310</td>
</tr>
<tr>
<td>Trial 2</td>
<td>1820</td>
<td>400</td>
<td>70</td>
<td>990</td>
<td>870</td>
</tr>
<tr>
<td>Trial 3</td>
<td>370</td>
<td>950</td>
<td>70</td>
<td>940</td>
<td>730</td>
</tr>
<tr>
<td>Trial 4</td>
<td>330</td>
<td>550</td>
<td>70</td>
<td>480</td>
<td>350</td>
</tr>
<tr>
<td>Trial 5</td>
<td>330</td>
<td>680</td>
<td>70</td>
<td>490</td>
<td>600</td>
</tr>
<tr>
<td>Trial 6</td>
<td>370</td>
<td>800</td>
<td>70</td>
<td>420</td>
<td>710</td>
</tr>
<tr>
<td>Trial 7</td>
<td>330</td>
<td>850</td>
<td>70</td>
<td>990</td>
<td>710</td>
</tr>
<tr>
<td>Trial 8</td>
<td>340</td>
<td>560</td>
<td>70</td>
<td>520</td>
<td>1140</td>
</tr>
<tr>
<td>Trial 9</td>
<td>370</td>
<td>390</td>
<td>70</td>
<td>600</td>
<td>750</td>
</tr>
<tr>
<td>Trial 10</td>
<td>330</td>
<td>520</td>
<td>70</td>
<td>520</td>
<td>870</td>
</tr>
<tr>
<td>Average</td>
<td>492</td>
<td>625</td>
<td>70</td>
<td>634</td>
<td>804</td>
</tr>
<tr>
<td>STDEV</td>
<td>466.9713291</td>
<td>189.22356</td>
<td>0</td>
<td>241.3020238</td>
<td>269.0394436</td>
</tr>
</tbody>
</table>

Since the agent performs best with a learning rate of 0.00025, we will proceed with that hyperparameter value for the subsequent experiments.

**D. Boost DQN Performance with Frame-Diff**

We present a novel approach to boost performance while training the neural network, called “Frame-Diff”. This approach involves building a quasi-attention technique for the CNN model by computing the difference between successive frames. Attention is a technique used when training DNNs where the net is made to give priority to specific parts of the input data compared to other parts. By computing the difference between successive frames, we often obtain a result (due to the similarity between successive frames) where the frames only highlight what changed in the time lapse between them. Further, in this manner, the states that are saved in the experience replay buffer consist only of the pixel values across frames that have changed – thereby reducing noise to the neural network.
Initially, in the base model, the input buffer contains the first frame (F1) duplicated 4 times. The Frame-Diff approach dictates that the frames should be the difference between the current and previous frames, so we initialize the buffer to contain 4 frames that contain no data, since each is the difference of the first frame and itself (F1 – F1). Starting from when the first action is played, and frame F2 is generated, the derived frame F2 – F1 is entered into the input buffer. Figure 10 below shows what such a derived frame would look like.

![Frame 1 (F1), Frame 2 (F2), Frame 2 - Frame 1 (F2 - F1)](image1)

Figure 10. Sample of a derived frame using the Frame-Diff approach.

A sample of the input buffer taken mid-game is shown below, in Figure 11. Observe that the noise in the inputs is greatly reduced, as the neural net can now focus directly on subtle changes between frame and learn more about temporal distinctions between frames.

![Frame Diff State Buffer](image2)

Figure 11. Sample Frame-Diff input buffer
This version of the deep learning model also uses Frame-Skipping. The frame-skip parameter selected is 4, as this has worked well in previous experiments and provides an adequate speedup to the training process. We train the model for 10,000 episodes and test it against the base model (also trained for 10,000 episodes). Initially, we experiment with the learning rate at 0.0001 and 0.00025 for the 10,000 episode model. The model is then trained for 30,000 and 50,000 episodes respectively, at a learning rate of 0.00025. The results for the experiments are documented below in Table 3.

<table>
<thead>
<tr>
<th>Trial</th>
<th>Base model, 10k eps, LR=0.00025</th>
<th>Frame-Diff, 10k eps, LR=0.0001</th>
<th>Frame-Diff, 10k eps, LR=0.00025</th>
<th>Frame-Diff, 30k eps, LR=0.00025</th>
<th>Frame-Diff, 50k eps, LR=0.00025</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial 1</td>
<td>660</td>
<td>350</td>
<td>2370</td>
<td>1210</td>
<td>2020</td>
</tr>
<tr>
<td>Trial 2</td>
<td>2070</td>
<td>820</td>
<td>1960</td>
<td>1410</td>
<td>2400</td>
</tr>
<tr>
<td>Trial 3</td>
<td>1070</td>
<td>380</td>
<td>1770</td>
<td>1800</td>
<td>2300</td>
</tr>
<tr>
<td>Trial 4</td>
<td>910</td>
<td>570</td>
<td>1710</td>
<td>1400</td>
<td>3730</td>
</tr>
<tr>
<td>Trial 5</td>
<td>940</td>
<td>460</td>
<td>1470</td>
<td>1400</td>
<td>2390</td>
</tr>
<tr>
<td>Trial 6</td>
<td>1210</td>
<td>480</td>
<td>2540</td>
<td>2000</td>
<td>2560</td>
</tr>
<tr>
<td>Trial 7</td>
<td>1360</td>
<td>670</td>
<td>2370</td>
<td>1380</td>
<td>2390</td>
</tr>
<tr>
<td>Trial 8</td>
<td>2170</td>
<td>460</td>
<td>1170</td>
<td>1840</td>
<td>2990</td>
</tr>
<tr>
<td>Trial 9</td>
<td>1170</td>
<td>460</td>
<td>990</td>
<td>1780</td>
<td>1240</td>
</tr>
<tr>
<td>Trial 10</td>
<td>670</td>
<td>570</td>
<td>2060</td>
<td>1100</td>
<td>3260</td>
</tr>
<tr>
<td>Average</td>
<td>1223</td>
<td>522</td>
<td>1841</td>
<td>1532</td>
<td>2528</td>
</tr>
<tr>
<td>STDEV</td>
<td>522.8352194</td>
<td>140.7756292</td>
<td>521.1834823</td>
<td>300.0666593</td>
<td>684.6377793</td>
</tr>
</tbody>
</table>

Frame-Diff comes through with some stellar results. The base model, trained at a learning rate of 0.00025 is able to score 1223 points on average over 10 trials. The Frame-Diff model, at the same learning rate, is able to score 1841 points on average while maintaining a mostly similar standard deviation of 521 among scores. This tells us that the model is performing with similar variation in scores (mostly consistent) across 10 trials, except far better per trial. This
equates to a 50.53% increase in average agent performance, which is highly promising since the model was only trained for around 1 hour for 10,000 episodes. After training the Frame-Diff model for 50,000 episodes, the agent obtains a high score of 3730. This is of particular importance as it is almost able to win the first map outright, but the agent is also prioritizing good gameplay over just racing to clear the map. The first maze can be cleared with 1700 points, by consuming all of the 170 food pellets without chasing after power ups. This version of the agent is able to shatter that 1700-point barrier. With adequate GPU memory and compute power, if the agent were to be trained past 50,000 episodes, a superior model is possible.

<table>
<thead>
<tr>
<th>RL Method</th>
<th>Average Score</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame-Diff</td>
<td>2528</td>
<td></td>
</tr>
<tr>
<td>DQN</td>
<td>2311</td>
<td>Minh et al. [18]</td>
</tr>
<tr>
<td>Dueling Architecture DQN (DuDQN)</td>
<td>2250</td>
<td>Wang et al. [21]</td>
</tr>
<tr>
<td>Deep Recurrent Q-Network (DRQN)</td>
<td>2048</td>
<td>Hausknecht and Stone [19]</td>
</tr>
<tr>
<td>Proximal Policy Optimization Algorithm (PPO)</td>
<td>2096.5</td>
<td>Schulman et al. [20]</td>
</tr>
<tr>
<td>DQN with Linear Q-Function</td>
<td>1692</td>
<td>Minh et al. [18]</td>
</tr>
</tbody>
</table>

From Table 4, we can see that the Frame-Diff approach compares favorably with other known reinforcement learning agents for Ms. Pac-Man. Frame-Diff is able to beat the DQN [1] by around 9.39%, which is impressive considering the Frame-Diff agent is not trained for as long as the original DQN was. Frame-Diff also beats the DQN with Linear Q-Function by nearly 50%, which is expected, since the DQN model performs better than that. However, even the Frame-Diff model trained for 10,000 episodes is able to beat the Linear DQN – which is trained in less than an hour. Considering the overall time and resource consumption of training Frame-Diff, the model performs superbly against the field.
Another important takeaway from this is not only the improvement in agent performance
over these other models, but that it was achieved in fewer total frames of training, implying the
model played less yet learned more. The DQN [18] is trained over 10 million frames. Our agent
is trained on an average of 364151 frames every 10,000 episodes, which is around 1.82 million
frames for 50,000 episodes. This works out to around an 81.8% improvement solely by the
number of training frames, which is enormous, especially as the number of training episodes
increase.

E. Transfer Learning on Different Mazes

Transfer learning refers to the problem of using a machine learning model in a different
context than what it was trained on. As the weights of the CNN are learned as part of the training
process, it is possible to “connect” different types of input to the CNN and observe the results.
Transfer learning is time and resource efficient as pre-trained models can be reused and some
deep learning models take an extremely long time to train.

One approach to experimenting with transfer learning in Ms. PacMan is to train an agent on
the first maze of Ms. PacMan and then deploy that agent on subsequent mazes to evaluate its
performance on mazes it has not encountered before. Maze 2 is an improved version of Maze 1
with faster ghosts that are also better at swarming the agent. Maze 3 is a completely different
maze altogether. Figure 12 below shows frames of both mazes.
The agent is trained over 10,000 episodes on maze 1, taking the difference between adjacent frames, the agent scores a high score of 1150 on maze 2. On average, the agent scores 612 points on Maze 2, compared to 1223 for Maze 1, which is a drop of around 50% in average performance. The standard deviation between scores is far lower as well. With Maze 3, with different entry/exit points and power-up layouts, the average score is even lower. Table 5 below outlines the agent’s performance over 10 trials, across Maze 1, Maze 2 and Maze 3.
Table 5. Comparing the performance of the agent across Maze 1 and Maze 2.

<table>
<thead>
<tr>
<th></th>
<th>Maze 1, 10k eps, LR=0.00025</th>
<th>Maze 2, 10k eps, LR=0.00025</th>
<th>Maze 3, 10k eps, LR = 0.00025</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial 1</td>
<td>660</td>
<td>620</td>
<td>470</td>
</tr>
<tr>
<td>Trial 2</td>
<td>2070</td>
<td>720</td>
<td>390</td>
</tr>
<tr>
<td>Trial 3</td>
<td>1070</td>
<td>1150</td>
<td>280</td>
</tr>
<tr>
<td>Trial 4</td>
<td>910</td>
<td>540</td>
<td>240</td>
</tr>
<tr>
<td>Trial 5</td>
<td>940</td>
<td>490</td>
<td>390</td>
</tr>
<tr>
<td>Trial 6</td>
<td>1210</td>
<td>710</td>
<td>300</td>
</tr>
<tr>
<td>Trial 7</td>
<td>1360</td>
<td>430</td>
<td>200</td>
</tr>
<tr>
<td>Trial 8</td>
<td>2170</td>
<td>380</td>
<td>150</td>
</tr>
<tr>
<td>Trial 9</td>
<td>1170</td>
<td>470</td>
<td>340</td>
</tr>
<tr>
<td>Trial 10</td>
<td>670</td>
<td>610</td>
<td>300</td>
</tr>
<tr>
<td>Average</td>
<td>1223</td>
<td>612</td>
<td>306</td>
</tr>
<tr>
<td>STDEV</td>
<td>522.8352194</td>
<td>220.5951546</td>
<td>95.93979594</td>
</tr>
</tbody>
</table>

F. Transfer Learning with User Gameplay

To further test the robustness of the DQN implementation, we experiment by transferring learning from user gameplay. As an expansion of our initial idea from Chapter II about playing PacMan with an agent over an internet connection, we address the question – *is it possible to train the agent by making it “watch” the user play?* That is, can the Q-agent learn an efficient control policy from gameplay videos? To address this question, we look to leverage computer vision to try and learn the input, reward and completion signals for a gameplay video of Ms PacMan.

This approach entails isolating and contouring the Ms PacMan character and tracing its movement across the screen. The first order of business is to isolate the PacMan object on the
screen, i.e., remove all other ghosts, maze walls, food pellets, score and life indicators, etc. Before removing the other objects on the screen, the input frame should be cropped to remove the score and life section of the screen. Removing all characters except for Ms PacMan, entails developing what’s known as a “yellow mask”, i.e., isolate the yellow color on the screen so that everything disappears except for the Ms. PacMan character. Converting the image from Red-Green-Blue (RGB) values to Hue Saturation Values (HSV) is helpful since HSV represents colors in a cylindrical model, which means we can specify ranges easily. For the purposes of this experiment, we experiment with a yellow mask that ranges from [20, 100, 100] to [30, 255, 255]. This will detect and isolate pixels that have that have values in those ranges. Figure 13 below samples a frame from the ALE rendering of Ms. PacMan, supported with the same frame yellow masked.

![Figure 13. Yellow masking a frame of Ms. PacMan to track the character.](image)

After the yellow masking step, the Ms PacMan character is isolated moving across the screen. OpenCV’s Canny contouring library provides a handy API for edge detection in images. Figure 5 below shows the result after applying the contouring API on the yellow masked image obtained in Figure 14.
Once the contours are obtained, computing the largest contour by area returns the edge for the biggest shape detected on the screen – which, if the yellow mask has been implemented correctly, is Ms. PacMan on the screen. The OpenCV moments function returns the center of a given contour. By passing the largest contour to the moments function, we can obtain the center of the Ms PacMan object. By taking the difference in the position of the center between successive frames, we can track the direction the object has moved.

To obtain a reward signal, the PyTesseract library is used to convert the score to text. The episode completion signal, in the naïve case, is when the frames have all been exhausted. Once the action, completion and rewards signals are set up, the agent can be trained using frames from gameplay by assembling a deque of 4 frames just like the base DeepMind model does. The results of training the agent through user gameplay is shown below, in Table 6.
Table 6. 10-trial average performance of the user-trained agent over 60 episodes.

<table>
<thead>
<tr>
<th></th>
<th>User-trained, 1 episode</th>
<th>User-trained, 5 episodes</th>
<th>User-trained, 10 episodes</th>
<th>User-trained, 60 episodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial 1</td>
<td>250</td>
<td>210</td>
<td>310</td>
<td>420</td>
</tr>
<tr>
<td>Trial 2</td>
<td>340</td>
<td>230</td>
<td>450</td>
<td>300</td>
</tr>
<tr>
<td>Trial 3</td>
<td>260</td>
<td>230</td>
<td>250</td>
<td>300</td>
</tr>
<tr>
<td>Trial 4</td>
<td>210</td>
<td>230</td>
<td>310</td>
<td>1000</td>
</tr>
<tr>
<td>Trial 5</td>
<td>240</td>
<td>230</td>
<td>260</td>
<td>380</td>
</tr>
<tr>
<td>Trial 6</td>
<td>240</td>
<td>210</td>
<td>250</td>
<td>360</td>
</tr>
<tr>
<td>Trial 7</td>
<td>340</td>
<td>120</td>
<td>250</td>
<td>360</td>
</tr>
<tr>
<td>Trial 8</td>
<td>310</td>
<td>230</td>
<td>450</td>
<td>180</td>
</tr>
<tr>
<td>Trial 9</td>
<td>340</td>
<td>120</td>
<td>310</td>
<td>180</td>
</tr>
<tr>
<td>Trial 10</td>
<td>340</td>
<td>210</td>
<td>310</td>
<td>290</td>
</tr>
<tr>
<td>Average</td>
<td>287</td>
<td>202</td>
<td>315</td>
<td>377</td>
</tr>
<tr>
<td>STDEV</td>
<td>51.86520992</td>
<td>44.1713834</td>
<td>76.19419634</td>
<td>232.7635901</td>
</tr>
</tbody>
</table>

A steady increase in average score is observed with an increase in the number of user-game play trials. The user-trained model outperforms the base model by factors of 1.63068182 and 1.578125 over 1 episode and 5 episodes of training, respectively. The divide grows further as the user-trained models outperforms the base model over 10 episodes of training by a factor of 2.37, which is indicative that the user trained model is learning to play well faster due to more reward-driven moves being taken by the user during training, instead of using a random action with some exploration probability value of occurring. Table 7 below outlines the base model’s performance over 60 episodes of training.
Table 7. 10-trial average performance of the base agent over 60 episodes.

<table>
<thead>
<tr>
<th>Trial</th>
<th>Base model, 1 episode</th>
<th>Base model, 5 episodes</th>
<th>Base model, 10 episodes</th>
<th>Base model, 60 episodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>130</td>
<td>90</td>
<td>120</td>
<td>320</td>
</tr>
<tr>
<td>2</td>
<td>130</td>
<td>220</td>
<td>120</td>
<td>220</td>
</tr>
<tr>
<td>3</td>
<td>230</td>
<td>160</td>
<td>140</td>
<td>700</td>
</tr>
<tr>
<td>4</td>
<td>130</td>
<td>90</td>
<td>190</td>
<td>210</td>
</tr>
<tr>
<td>5</td>
<td>170</td>
<td>180</td>
<td>110</td>
<td>210</td>
</tr>
<tr>
<td>6</td>
<td>210</td>
<td>70</td>
<td>110</td>
<td>260</td>
</tr>
<tr>
<td>7</td>
<td>170</td>
<td>90</td>
<td>120</td>
<td>210</td>
</tr>
<tr>
<td>8</td>
<td>230</td>
<td>70</td>
<td>190</td>
<td>330</td>
</tr>
<tr>
<td>9</td>
<td>190</td>
<td>90</td>
<td>110</td>
<td>210</td>
</tr>
<tr>
<td>10</td>
<td>170</td>
<td>220</td>
<td>120</td>
<td>190</td>
</tr>
<tr>
<td>Average</td>
<td>176</td>
<td>128</td>
<td>133</td>
<td>286</td>
</tr>
<tr>
<td>STDEV</td>
<td>38.93013686</td>
<td>60.69962475</td>
<td>31.28720008</td>
<td>153.4202652</td>
</tr>
</tbody>
</table>

The user-trained scores, while trending in the right direction, are not conclusive. There is a very little training data to fully evaluate the robustness of the model. The average scores, while better than a randomly-acting agent, are not that far better within 60 episodes to make a definite statement about the scalability of this type of training. The user-trained agents are learning from gameplay that tends to play a lot more perfectly than the $\varepsilon$-greedy agent, which means the rewards as the game draws on longer remains a lot more consistent. The agent may not be able to replicate this when it plays its own games, after training for more episodes. All the agents discussed in this project were trained on thousands of episodes so that a meaningful Q-function could be trained to evaluate what the agent learned and how it has improved over time. With just 60 episodes, scalability remains a major question. Recording games using the ALE is quite a time-consuming and resource-intensive task as the games need to be played through frame by
frame, providing input for approximately every 0.08 seconds of gameplay. However, this project does work as a proof-of-concept and could benefit from data augmentation.
V. Conclusion

Classic Atari 2600 games provide great environments for the application of reinforcement learning algorithms. As discussed in this report, there have been several such projects, especially over the last decade or so. However, some of the Atari 2600 games with heavier emphasis on exploration and “risk-taking” have proven challenging for these agents to solve. Our results in implementing a Deep Q-Agent for one such game, Ms. Pac-Man, has shown promising results.

The agent is able to score more than 3700 points on Ms. Pac-Man and is able to nearly win the first stage with only 6 pieces of food remaining. Due to limitations in compute power on the MacBook Air M1, we are currently unable to train the model past 50000 episodes. However, observing the trend in increasing performance against a higher number of training episodes, it is safe to say the game (at least on the first map) is solvable.

Further, while the agent suffers a drop in performance under conditions of latency and network lag, it is still able to perform reasonably well enough. These results are also scalable, so the model will improve as it is trained for more episodes. The same holds for transferring the learning of the agent onto different mazes of the game, as the agent is able to demonstrate it is playing to reach a high score and not moving randomly.

Lastly, we analyze the results of training the agent on user gameplay. By comparing the results of a user-trained agent vs the base model trained for the same number of episodes, we established that the user trained agent is performing slightly better on average. However, there is a paucity of training data (recorded gameplay of Ms. PacMan) here.

The results also bode well for further research in this field. There are a number of key areas highlighted in this project which provide exciting avenues of research.
A. Guided Training
As an expansion of the user-gameplay training idea covered in this project, there is scope to experiment with augmenting training data to obtain more gameplay videos. One possibility might be to train a generative deep Q network that is trained over far more episodes on a high-performance computing (HPC) cluster. That agent could then play hundreds if not thousands of games of Ms. PacMan, and those gameplay sessions could be recorded. Those videos could be sampled frame by frame using the computer vision approach described in this paper and used to train a new Q-agent to see if the performance is improved with a more consistent reward signal. An alternative could be to record the move and reward history and just manually train the agent on gameplay videos via that, instead of using any computer vision component.

B. Game Modification
There are a number of interesting ideas that could be explored if the game itself is modified in some way. For example – if the ghosts were controllable - could dueling Q-agents be trained for the ghosts and the Ms. PacMan character, fighting each other on one game at one time? What if we train the Q-agent on different mazes, maybe even custom mazes, and then try to transfer that to the base PacMan game?

C. Transfer learning from Similar Games
Another interesting idea that arises out of this project is that of scraping user gameplay videos of similar games like the original PacMan (1982), or the console version of Ms. PacMan and trying to train an agent on those games. Transferring learning from those games onto the base Atari 2600 Ms. PacMan could be an exciting project on its own.
Ultimately, there are several thrilling applications of reinforcement learning to solve problems in the real world, not just limited to the use cases that have been described in this project. Our experiments in this project have shown favorable results in adapting DNNs to learn control policies for Ms. PacMan. Further research and innovation in this domain of artificial intelligence only needs a confluence of curious minds and determined effort.
REFERENCES


[8] "Mobile Object Detection using Tensorflow Lite and Transfer Learning.". Alsing, Oscar. 2018


