Deep Reinforcement Learning with Long Short Term Memory for Atari Pong

Author #1 Aadil Akhtar (auth#1 301573180) - *auth#1 Computer Science*
Author #2 Diego Huerta (auth#2 301573338) - *auth#2 Computer Science*
Author #3 Joel Del Castillo (auth#3 301573186) - *auth#3 Computer Science*

Supervision

– Supervisor #1 Dr. Hazra Imran (sup#1-himran@sfu.ca)

– Supervisor #2 Hyeon Lee (sup#2-jla465@sfu.ca)

Abstract

Atari Pong is one of the oldest and most popular video games. Despite its simple graphics and gameplay, the effectiveness of an artificial intelligent agent predicting the future path of a moving object is a complex task where Long Short Term Memory (LSTM) Neural networks can help in learning and recognizing patterns. Most of the research done before in Pong agents used Neural Networks extracting features like the relative position of the ball and paddle to correctly estimate the best action to take rather than learning about the trajectory. For that reason, we implemented Deep Reinforcement Learning architectures like Deep SARSA and Deep Q-learning (DQN) using Long Short Term Memory for the game of Pong so that it predicts the ball's trajectory to better estimate the actions to take. We evaluate these methods by testing them over a number of episodes and comparing their rewards returned using Mann-Whitney U tests. Finally, experimental results show that when using LSTM, the policy improves much faster, but the time taken per episode is significantly more.

Introduction

In the past decade, Reinforcement Learning (RL) has been successfully applied to various control tasks, including robotics, game-playing, and decision-making. One of the key benefits of RL is its ability to enable an agent to automatically learn how to perform a task by trial and error without the need for extensive prior knowledge or manual programming. This makes RL particularly well-suited to tasks that are difficult to specify in advance or where the environment is constantly changing.

Besides, the skill of Neural Networks to learn using Long Short Term Memory (LSTM) has made them well-suited to modeling temporal data. Unlike traditional recurrent neural networks, LSTM is able to retain information for small periods of time, which makes it ideal for learning sequences of data. It has been used in a wide range of applications, including speech recognition, language modeling, and machine translation.

Much of the reinforcement learning research has been developed in the context of video games, especially Atari games; they have become one of the most used benchmarks. Within these games, Pong is one of the most used to train and test reinforcement learning algorithms.

However, most of the implementations available do not approach the trajectory prediction of the Pong ball. This is a difficult task because the game is highly non-linear, but LSTM has been shown to be successful in learning such sequences to predict the opponent's next move.

Therefore, teaching an agent to play video games is an important research area because video games are complex high-dimensional problems, and the algorithms used to train agents in these environments could easily generalize into real-world applications.

The research aims to create a functional agent capable of playing Pong to determine the significance of using memory by testing the performance of each algorithm with and without LSTM. The methods this paper considers are DQN, and Deep SARSA, as they are some of the most famous algorithms, and they have shown satisfactory performance in multiple RL domains. These methods will be used to train several agents to play the pong game, and then their performance will be examined and compared to each other by testing them over a number of episodes and comparing their rewards returned using Mann-Whitney U tests.

Finally, our research tries to contribute to the following aspects: First, to implement an LSTM architecture into a Pong agent and determine whether it benefits its performance or not. Second, to compare different RL algorithms and determine the most appropriate for this specific problem. Third, to implement an image pre-processing for efficient calculations.

Our research question is: Will Long Short-Term Memory affect the behavior and performance of the Pong Agent? If so, which algorithm best benefit this architecture?

Main Results

We found that the LSTM implementation performs better than the baseline network in terms of the final performance. However, the benefits of using an LSTM are not always consistent and come with a downfall in time efficiency. When using LSTM, the policy improves earlier but the time taken per episode increases. For 400 episodes, both algorithms, Deep SARSA and DQN, took an average of 1.45 hours more by using LSTM, but it helped them to stabilize and improve their training rewards by an average of +1.08 points.

Whereas the last trained model of each algorithm was tested 100 times to produce their rewards, where the mean DQN, DQN with LSTM, Deep SARSA, and Deep SARSA with LSTM is -18.18, -16.76, -17.29, and -15.17 respectively. Where it shows, in both cases, that SARSA, with and without LSTM, generates better results.

Contributions

- Long Short-Term Memory Implementation for the Deep Learning Pong Agents.
- Efficiency comparison of two main Deep Learning Algorithms: DQN and SARSA in the Pong environment.
- Image prepossessing of every frame of the game for efficient calculations.

Related Work

Long short-term memory was first proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber. It was designed to address the vanishing gradient problem, which is a major issue for traditional recurrent neural networks.(Hochreiter & Schmidhuber, 1997). This Vanishing gradients occur when the error signal (i.e. the gradient) gets progressively smaller as it is back-propagated through time. This can make it very difficult for the network to learn long-term dependencies. LSTMs solve the vanishing gradient problem by using a special type of cell state, which is able to retain information for long periods of time. This enables the network to learn dependencies that are many time steps away. It is this capability that has led to their widespread use in a variety of applications, such as natural language processing, speech recognition, and time series forecasting.

In 2013, Deep Mind researchers presented the first deep learning model to learn to play Atari games successfully. They used a convolutional neural network trained with Stochastic Gradient Descent(SGD), which was updated with a variant of Q-learning. The agent trained in the Pong environment learned to play better than an expert human and outperformed all previous implementations of RL algorithms (Mnih et al., 2013). Later in 2020, Deep Mind published Agent57, which outperformed human experts in 57 atari games (Badia et al., 2020).

In the context of deep reinforcement learning, one common approach is to update the convolutional neural network with Q-learning. However, this is not the only option; there are alternatives such as Sarsa and Advantage actor-critic. In the research by (Zhao, Wang, Shao, & Zhu, 2016), deep reinforcement learning is applied using a neural network with three convolutional layers and two fully connected layers to predict the state-action value function of two Atari games. They used the Q-learning and Sarsa approaches and concluded that

SARSA learning is more suitable for complicated systems as it obtained higher results.

Many recent reinforcement learning approaches are implemented and tested in the context of atari games (including Pong). For example, the paper of (Livne & Cohen, 2020) studies a technique to prune over the neural networks used in deep reinforcement learning; this aims to achieve a solid performance while having a compact representation of the neural network, which usually takes up high computational space. Also, in the research of (Monroy, Stanley, & Miikkulainen, 2006), the Layered Pareto Coevolution Archive is shown to be a practical coevolutionary memory of the neural networks in the game of Pong, presenting a significant storage advantage.

Since the 2000s, there has been a growing interest in developing technology capable of imitating human behavior. This work was more noticeable in 2008 when the IEEE Transactions on Robotics involved Human-Robot interactions and Movement generation with a new hierarchy formation from observing human motion (Kulić, Takano, & Nakamura, 2008).

Years later, RL algorithms have successfully learned various skills for embodied agents, such as robots and simulated characters. While these methods have shown great promise, they are still far from achieving human-like control of motor skills. In the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, the problem was presented to generate a realistic human figure in motion from a given set of input data, typically acquired from video (Martinez, Black, & Romero, 2017). The problem was difficult because of the large number of degrees of freedom involved in human motion to predict these patterns.

Methodology

Reinforcement Learning

Reinforcement learning aims to create agents similar to humans, which learn for themselves by trial-and-error, solely based on rewards or punishments that eventually lead to the largest long-term rewards or minimum long-term punishments.

An RL agent interacts with an environment as follows: At each time t, the agent receives an observation o_t from the set of observation space O or receives a state s_t from the set of state space S, after which it takes action a_t from the set of Action space A defined by a policy $\pi(a_t|s_t)$, and it receives an immediate reward r_t and transitions into a new state s_{t+1} according to the environment. The policy is a map from state to action, and it can be either a deterministic policy $a = \pi(s)$ or a stochastic policy $\pi(a|s) = p[A = a|S = s]$. The model consists of the environment's reward function $R(s, a)$ and the state transition probability matrix $P(s_{t+1}|s_t, a_t)$. The RL agent continues to take steps in the environment until it reaches a terminal state and restarts again. For each step, we define a return

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \text{ (1)}$$

which is the summation of discounted immediate rewards of each step, where the discount factor is defined by $\gamma \in (0, 1]$. The agent aims to maximize the expectation of the long-term return from each state. The value function is an important concept in reinforcement learning which evaluates the goodness/badness of a state or a state-action pair. We can either have a state-value function,

$$V_\pi(s) = E_\pi[R_{t+1} + \gamma R_{t+2} + \ldots | s_t = s]$$

i.e., the expected return for following a policy π from a state s or an action-value function,

$$q_\pi(s, a) = E_\pi[R_{t+1} + \gamma R_{t+2} + ... | s_t = s, a_t = a]$$

i.e., the expected return for following a policy π when selecting an action a in a state s. It is also used as a prediction of expected accumulated discounted future rewards. The agent's goal is to find an optimal policy π^* to maximize the expectation of long-term rewards. The RL problem defined by the 5-tuple (S, A, P, R, γ) can be formulated as a Markov Decision Process(MDP) or a Partially Observable Markov Decision Process(POMDP) (Jeerige, Bein, & Verma, 2019).

Deep Reinforcement Learning

Most successful RL applications have relied on hand-crafted features combined with linear functions or policy representation. The performance of such systems heavily relies on the quality of the feature representation. Deep RL has made it possible to extract high-level features from raw sensory data using a range of neural network architectures, including convolutional networks, multi-layer perceptrons, and recurrent neural networks.(Mnih et al., 2013). For that reason, Deep RL algorithms do not require a dataset in order to learn. Instead, they learn by interacting with their environment without any prior knowledge.

The agents considered for this paper will perform in a Ping-Pong environment, taken from OpenAI Gym, which provides a toolkit for reinforcement learning research. That is, a screen with two paddles that can move up or down and a ball bouncing between them. The agents will know the environment through the RGB image's pixel displayed in every timestamp of the game, i.e., the environment is fully observable MDP. At each time step, the agent selects a discrete action a_t from the set of action space $A = \{Noop, Left, Right, \ldots\}$(Brockman et al., 2016). The action is passed to the emulator and modifies its internal state and the game score. For each step taken, we get an immediate reward of $+1$ for getting the ball to pass the opponent's paddle, the immediate reward of -1 if the ball passes our paddle, and 0 otherwise. We take this game score as our reward function, which the RL agent will try to maximize. That is, the agent will try to score as many points as possible and prevent the opponent from scoring points. The episode terminates when any one of the players has a cumulative score of 21. The total reward in one episode is calculated as the difference between the number of times we win versus the number of times we lose.

As mentioned, the research aims to compare the two reinforcement learning algorithms of Deep Q-Learning and Deep SARSA Learning, including a variation using LSTM neural network for the algorithms. This comparison will be made on the Atari environment game pong, and will aim to understand how the reinforcement learning algorithms maximize the reward in the long term.

Working directly with Atari frames which are 210×160 pixel images with a 128 color palette, can be computationally demanding, so we apply a primary preprocessing step to reduce the input dimensionality. They are preprocessed by first converting their RGB representation to grayscale and cropping a region of the image that roughly captures the playing area. It is then down-sampled to an 80×64 image. The algorithm applies this preprocessing to the last four frames of the history and stacks them to produce the input to the Q-function. The input to the neural network consists of an $80 \times 64 \times 4$ image. The first hidden layer convolves 32 8×8 filters with stride 4 with the input image and applies a RELU activation. The second hidden layer convolves 64 4×4 filters with stride 2, followed by a RELU activation. The third layer convolved 64 3×3 filters with stride 1 with the input image and applies RELU activation. The fourth hidden layer is a fully-connected layer and consists of 256 rectifier units with RELU activation. The final hidden layer is a linear layer and consists of 128 rectifier units with RELU activation. The output layer is a full-connected linear layer with a single output for each valid action (Mnih et al., 2013). This neural network architecture is shown in Figure 1. We train these hyperparameters with different algorithms listed below.

Deep Q-learning

Deep Q-learning (DQN) is a neural network-based reinforcement learning algorithm that attempts to approximate the Q-function, which maps states to action values. The goal of the DQN algorithm is to learn the Q-function so that it can be used to decide which actions to take to maximize the expected reward. It is an off-policy method, and in order to update the current state-action value function, we employ the next state-action value function to estimate it. So even though the next state s' is given to us, the action a' is still unknown to us, and we choose greedily to maximize the Q function (Mnih et al., 2013). The update equation is

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \qquad (2)$$

where α is the learning rate.

We use a Convolutional Neural Network(CNN) as the value function approximation whose input is the raw images from the video game, and the output is the Q values of all actions. Defining θ as the parameters of the CNN, we define

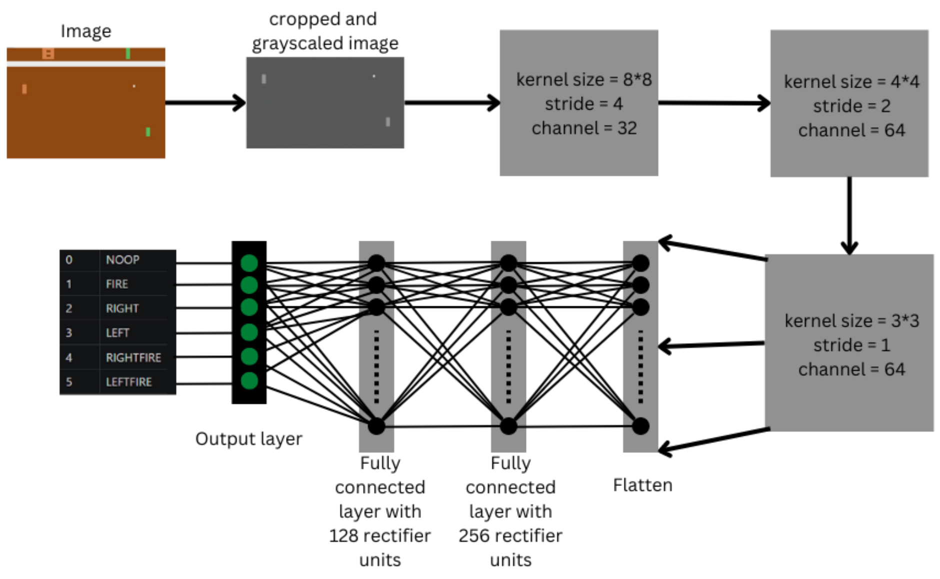


Figure 1: Neural Network Architecture. For neural networks using LSTM, we just replace the first fully connected layer with an LSTM layer

the loss function of the network at i_{th} iteration as

$$L_i(\theta_i) = (y_i - Q(s, a; \theta_i))^2 \qquad (3)$$

where $y_i = r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})$. The main objective is to optimize the loss function $L_i(\theta_i)$. On differentiating (3), we get the gradient of the loss function as

$$\nabla_{\theta_i} L_i(\theta_i) = (r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i))\nabla_{\theta_i} Q(s, a; \theta_i) \qquad (4)$$

The algorithm is designed to work with a replay buffer, a data structure that stores experience tuples (s, a, r, s') to train the neural network. This algorithm has been widely studied to determine its rate of convergence, an extensive analysis of DQN was presented in the work of (Fan, Wang, Xie, & Yang, 2019).

Deep SARSA Learning

SARSA learning is an on-policy method. It means that when updating the current state- action value, the next action a' will be taken. So the update equation of the state-action value is

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)] \qquad (5)$$

where α is the learning rate. We use the ϵ-greedy method for choosing action a given the state s.

Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights for network policy $\theta$
and target policy $\theta^-$
for episode $= 1, \ldots, M$ do
    $\theta^- = \theta$
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, \ldots, T$ 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 the 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 $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        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 (4)
    end for
end for
```

We use a Convolutional Neural Network(CNN) as the value function approximation whose input is the raw images from the video game, and the output is the Q values of all actions. Defining θ as the parameters of the CNN, we define the loss function of the network at i_{th} iteration as

$$L_i(\theta_i) = (y_i - Q(s, a; \theta_i))^2 \qquad (6)$$

where $y_i = r + \gamma Q(s', a'; \theta_{i-1})$. The main objective is to optimize the loss function $L_i(\theta_i)$. On differentiating (6), we get the gradient of the loss function as

$$\nabla L_i(\theta_i) = (r + \gamma Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i))\nabla Q(s, a; \theta_i) \qquad (7)$$

where $\nabla Q(s, a; \theta_i)$ is the gradient of the current state-action value. Thus according to (7), We can optimize the loss function using stochastic gradient descent (SGD) (Zhao et al., 2016).

Each of the above algorithms is trained on the pong game environment using the openAI gym library with the score as our reward. We will compare the two algorithms based on their performance, i.e., the maximum score obtained, which tells us about the long- term reward maximization of each of the algorithms and the time taken to converge cause when using these algorithms in real-life, their performance is largely based on its real-time application.

As mentioned, both of the algorithms considered are neural network-based methods. That is, they make use of a Neural network to optimize the desired results. In particular, both methods use CNN to make computations efficiently

Algorithm 2 Deep Reinforcement Learning based on SARSA

```
Initialise data stack $D$ with size of $N$ and parameters of CNN $\theta$
for episode $= 1, \ldots, M$ do
    Initialise states $s_1 = \{x_1\}$ and preprocess state $\phi_1 = \phi(s_1)$
    select $a_1$ with $\epsilon-$greedy method
    for $t = 1, \ldots, T$ do
        Take action $a_t$, observe next state $x_{t+1}$ and $r_t$,
        $\phi_{t+1} = \phi(s_{t+1})$
        store data $(\phi_t, a_t, r_t, \phi_{t+1})$ into stack $D$ sample data from stack $D$
        select $a'$ with $\epsilon-$ greedy method
        $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ R_j + \gamma Q(\phi_{j+1}, a'; \theta) & \text{otherwise} \end{cases}$
        according to (7), optimize the loss function $L_i(\theta_i)$
        $a_t \leftarrow a'$
    end for
end for
```

with the raw input. Several variants of Neural Networks can be applied to the reinforcement learning algorithms to improve performance. In this research, we use LSTM neural networks to consider new variants of the methods. That is, we make a difference between Deep Q-learning, Deep Q-learning with LSTM, Deep SARSA, and Deep SARSA with LSTM.

Long short-term memory (LSTM)

Long short-term memory (LSTM) is a Neural Network proposed to address the problems of vanishing and exploding gradients, commonly presented when training deep NN. It can process entire data sequences, such as video or speech recognition.

The LSTM architecture consists of a set of connected blocks. Each block is responsible for controlling the flow of information through non-linear functions and maintaining its state over time. More concretely, given the current input signal $x^{(t)}$ and the output of that LSTM unit $y^{(t-1)}$ in the last iteration. The forward pass, i.e. computation of $y^{(t)}$, is described below (Houdt, Mosquera, & Nápoles, 2020).

Block input. It combines the current input and the last output of the unit to update the block input component.

$$z^{(t)} = g(W_z x^{(t)} + R_z y^{(t-1)} + b_z)$$

where W_z and R_z are weights matrices associated with $x^{(t)}$ and $y^{(t-1)}$ respectively, b_z is the bias vector.

Input gate. It combines the current input, the last output of the unit, and the cell value of the last iteration to update the input gate.

$$i^{(t)} = \sigma(W_i x^{(t)} + R_i y^{(t-1)} + p_i \odot c^{(t-1)} + b_i)$$

where W_i, R_i and p_i are the weights associated to $x^{(t)}$, $y^{(t-1)}$ and $c^{(t-1)}$ respectively, b_i is the bias vector, $\odot$ denotes point-wise multiplication of two vectors, and σ is a point-wise non-linear activation function.

Forget gate. It determines which information should be removed from its previous cell states. It is calculated using the values of the current input, the last output, and the cell value of the last iteration.

$$f^{(t)} = \sigma(W_f x^{(t)} + R_f y^{(t-1)} + p_f \odot c^{(t-1)} + b_f)$$

where W_f, R_f and p_f are the weights associated to $x^{(t)}$, $y^{(t-1)}$ and $c^{(t-1)}$ respectively, b_f is the bias vector

Cell. The cell value is computed using the block input, input gate, forget the value and the cell value of the last iteration.

$$c^{(t)} = z^{(t)} \odot i^{(t)} + c^{(t-1)} \odot f^{(t)}$$

Output gate. It combines the current input, the last output of the unit, and the cell value of the last iteration to update the output gate.

$$o^{(t)} = \sigma(W_o x^{(t)} + R_o y^{(t-1)} + p_o \odot c^{(t-1)} + b_o)$$

where W_o, R_o and p_o are the weights associated to $x^{(t)}$, $y^{(t-1)}$ and $c^{(t-1)}$ respectively, b_o is the bias vector

Block output. Finally, the block output is calculated by combining the current cell value and the output gate.

$$y^{(t)} = g(c^{(t)}) \odot o^{(t)}$$

where g is a point-wise non-linear activation function.

In the implementation, instead of using the fourth hidden layer of the fully-connected layer with 256 rectifier units, we used an LSTM layer with 256 units. Everything in the NN architecture except the thing mentioned above is the same for both the LSTM and non-LSTM models.

Training and Testing

We train our model with one NVIDIA GTX 1650ti GPU in Python 3.9 with PyTorch 1.13.0. The learning rate is fixed to 1×10^{-4}, and the training optimizer is *Adam* with mean-squared error as the loss function. We use $\epsilon-$greedy to balance exploration and exploitation. The initial value is set to be $\epsilon = 1$ and decreases to 0.02 linearly when the total training steps reach $200,000$. The discount factor for which the model gives the best result is 0.99. Replay memory of size $10,000$ is used with a batch size of 128 for training.

Once an agent is trained for each of the four models, we will test the performance in the environment in the following way. The agent will play pong against the opponent until one of the players gets a total of 21 points; then, the reward for that game is the number of points of the agent minus the number of points of the opponent. Each agent will be tested for 100 games, collecting 100 rewards. Then, we will use a Mann–Whitney U test and the Exponentially Weighted Moving Average to search for significant differences in the performance of the models.

The Mann–Whitney U test is a nonparametric test on sample data coming from groups x and y; it is used to check if the distribution underlying sample x is the same as the distribution underlying sample y. (Mann & Whitney, 1947) It has the assumption that the observations from both groups are independent of each other and the data is ordinal; the null hypothesis is that the distributions are identical. The alternate hypothesis can be that the distributions are not equal, in which case it is a two-sided test. The alternate hypothesis could also be that one underlying distribution is stochastically less than the other, which is a one-sided test. We use Mann–Whitney U statistical tests to compare the results.

The Mann–Whitney U is performed between exactly two groups, and the number of different methods is four. Therefore, we have to perform a Mann–Whitney U test between each possible pair of models; that is, six different statistical tests. Therefore, we will use a Bonferroni correction to counteract the multiple comparisons problem (Ranstam, 2016). The threshold for rejecting the null and accepting the alternate hypothesis will be $0.05/6 = 0.0083$

Additionally, the Exponentially Weighted Moving Average (EWMA) is a statistical tool used to smooth out data points in a series by giving more weight to recent data points and less weight to older data points. This makes the EWMA more responsive to changes in the data series than a simple moving average, which gives equal weight to all data points.

In our research, EWMA is useful for analyzing the trend of each model and tracking the reward received by the agent at each time step and giving a bonus based on how far away the current reward is from the average. The bonus is exponentially weighted so that recent rewards are given more weight than older rewards.

Results

The code was implemented using PyTorch, which automatically takes care of backpropagation very efficiently. Each model was trained for a total of 400 episodes, having a different number of total steps, and the training time was also distinct. Table 1 shows the amount of training time and steps taken for each one of the four models.

Table 2 shows the EWMA total reward as a function of the episode number of the training process for each one of the four methods considered.

After training, each model was tested for a total of 100 games, producing a reward for each one of them. We take this quantity as a metric of the model performance. The histograms showing the distribution of the rewards for each algorithm can be appreciated in Table 3. The mean reward for DQN, DQN

Method	Steps	Time
DQN	552879	4 hrs 57 mins
DQN with LSTM	640492	6 hrs 52 mins
Deep SARSA	558116	5 hrs 13 mins
Deep SARSA with LSTM	573230	6 hrs 8 mins

Table 1: Summary of 400 episodes of each Agent

with LSTM, SARSA, and SARSA with LSTM is -18.18, -16.76, -17.29, and -15.17, respectively. The frequency of the rewards is the data used for the following analysis.

The comparison is made using Mann–Whitney U tests. Therefore the null hypothesis for each test is that the underlying distribution of the rewards coming from both methods is the same. Moreover, the tests realized were one-sided; that is, using the alternative hypothesis that one underlying distribution is stochastically less than the other. Given a pair of methods to compare, the criteria for choosing which underlying distribution will be tested to be less than the other is the sample mean. The p-values of the test for each pair are given in the Table 4. The alternative hypothesis for each one is that the distribution underlying of rewards of method 1 is stochastically less than the distribution underlying of rewards of method 2. The alpha value for each test is equal to $0.005/6 = 0.833$, coming from a Bonferroni correction of 6 different test and an overall significance value of 0.05.

Technical challenges

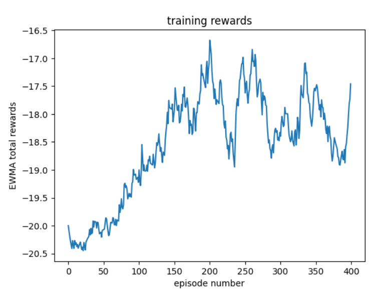
Because of the high-level complexity implementation of the algorithms, we had a number of technical issues, the following points are some of the biggest challenges we faced.

1. Develop a robust reward function that can capture the nuances of the Pong and provide meaningful feedback to the agent.

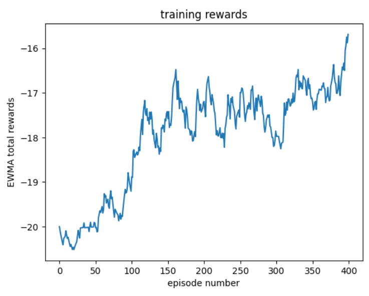
2. Avoid over-fitting the agent to the training data due to the sparse and delayed nature of the feedback signal.

3. Dealing with the exploration-exploitation trade-off where the agent must balance exploration (of new strategies) with exploitation (of known good strategies).

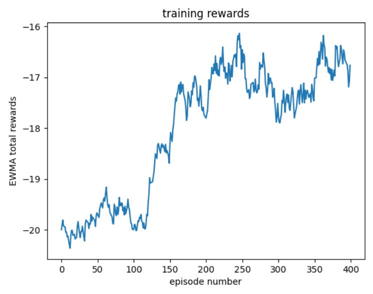
4. Optimizing our computational resources. For every change, we needed at least 30 minutes to check if it was well implemented. Therefore, our capacities were limited by time.



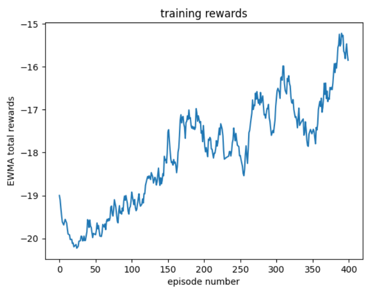
(a) DQN EWMA



(b) DQN LSTM EWMA



(c) Deep SARSA EWMA



(d) Deep SARSA LSTM EWMA

Table 2: Training rewards over episode number for each method

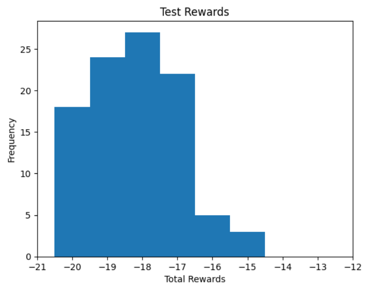
Discussion

The statistical tests performed on the reward test data revealed that, in general, the method for training the model makes a difference in the model performance after training. Namely, based on the data collected for testing. We found that:

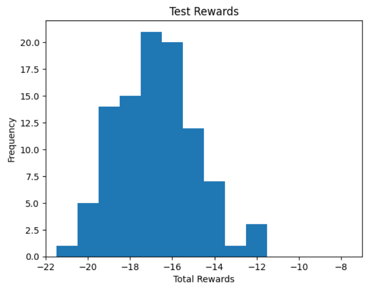
- The model trained with DQN had the worst performance among the four models
- The model trained with Deep SARSA and LSTM had a better performance than the models trained with Deep SARSA and DQN
- We could not find any significant difference between the performance of the model trained with Deep SARSA and the model trained with DQN and LSTM.

From that, the most important things to note are the improvement provided by LSTM and also the supremacy of SARSA over DQN.

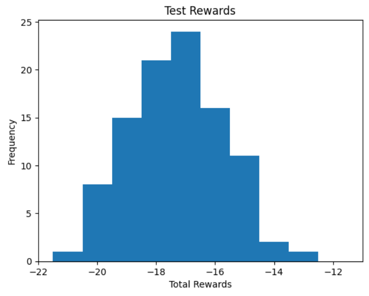
For both DQN and SARSA, the models using LSTM had a better performance than the models without it. This means that the use of LSTM in the training process helps the model to optimize in a better way. Our results match



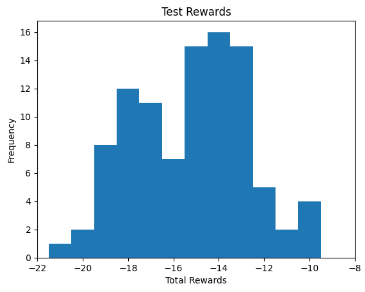
(a) DQN frequency



(b) DQN LSTM frequency



(c) Deep SARSA frequency



(d) Deep SARSA LSTM frequency

Table 3: Test rewards for each method

other investigations that show the effectiveness of LSTM in reinforcement learning applications, and also theoretical results that back up the power of LSTM. Nevertheless, the better performance given by LSTM comes at the cost of more time needed for training, as we can see that for both DQN and SARSA, the LSTM version took considerably longer to train.

Moreover, the models trained using SARSA outperformed the models trained using DQN (in both cases with/without LSTM, separately). This could indicate that the SARSA algorithm is more efficient for the domain of pong. Actually, our results agree with previous investigations that have shown that SARSA learning is more suitable for complicated systems, as it obtains higher results.

Returning to the objectives of the research, we can now conclude that different reinforcement learning algorithms present several differences when applied to high-dimensional problems, specifically the pong environment of atari games. Even for the same amount of episodes, the algorithms need a different amount of time for training, and the number of steps of each one also differs. In terms of the performance achieved, even for 400 episodes, we can see a significant difference in the reward achieved by the models. More importantly, we can see that the application of LSTM neural networks on the optimization problem of DQN and SARSA helps to improve the overall performance of the trained model.

Table 4: P-values for the one-sided Mann–Whitney U tests

Method 1	Method 2	P-value	Reject
DQN	DQN with LSTM	$1.0468054810226425e-08$	True
DQN	SARSA	$2.9320817666609172e-05$	True
DQN	SARSA with LSTM	$7.244713698178875e-17$	True
SARSA	DQN with LSTM	0.026740222364035584	False
SARSA	SARSA with LSTM	$8.154030547674599e-10$	True
DQN with LSTM	SARSA with LSTM	$2.3414024821110084e-06$	True

The analysis of the performance of the algorithms was made considering only one model of each type. Therefore, it is possible that the training process's randomness produced different results than expected.

Moreover, for the agent interaction with the Pong environment, we faced time and processing constraints. For example, for a single episode, it took 35 seconds on average to get the scores. This interaction is typically done through trial and error, which can be very compute-intensive and memory-demanding.

Given that we needed to test many algorithms in a larger period of time, within many episodes, we explored the use of image prepossessing. Even though it was not part of our goals for the project, we managed to improve the learning speed to 10s for each episode without losing the accuracy of the algorithms.

Conclusion

Our motivation for is to prove that LSTM is resistant to the vanishing gradient problem, where the reward function could become small, making it difficult to train the network. In the context of a Pong agent, it can be used to learn the game by observing the trajectory of the ball and predicting the future position. This is important, not only in games like Pong but also in real-life problems like predicting the pattern of a decease where the agent needs to learn from a large number of previous states.

For this research, we focus on implementing and comparing reinforcement learning methods. The environment chosen was the atari game of pong, as it shares some characteristics with real-world problems, such as complexity and high dimensionality.

We focused on DQN and SARSA algorithms, each with a variation using LSTM neural networks in the optimization process. Thus, we considered four different models, and all of them were trained in the pong environment. After the training, we conclude that different algorithms have different training times, and have different performances at the end. Also that there are models more powerful than others; for instance, models with LSTM perform better than models without it.

As future directions, other Neural Networks variations and architectures can be applied to deep reinforcement algorithms. For example, radial basis function networks or modular neural networks can be implemented instead of the LSTM choice made for this research.

During our experiments, we noticed that our Pong agent and all the Reinforcement Learning agents available on the internet have similar random behavior that makes them change their states much quicker than a human could do. For that reason, we encourage the next research to use a different approach where the agent can understand human motor skills by observing and imitating the actions of a human in Pong, creating a new reward function where the high number of changes in time gets punished and using acceleration rather than position states so that the agent can act more smoothly, and precise like a human could do.

References

Badia, A. P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, Z. D., & Blundell, C. (2020, 13–18 Jul). Agent57: Outperforming the Atari human benchmark. In H. D. III & A. Singh (Eds.), *Proceedings of the 37th international conference on machine learning* (Vol. 119, pp. 507–517). PMLR. Retrieved from `https://proceedings.mlr.press/v119/badia20a.html`

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. *CoRR*, *abs/1606.01540*. Retrieved from `http://arxiv.org/abs/1606.01540`

Fan, J., Wang, Z., Xie, Y., & Yang, Z. (2019). *A theoretical analysis of deep q-learning.* arXiv. Retrieved from `https://arxiv.org/abs/1901.00137` doi: 10.48550/ARXIV.1901.00137

Hochreiter, S., & Schmidhuber, J. (1997, 12). Long short-term memory. *Neural computation*, *9*, 1735-80. doi: 10.1162/neco.1997.9.8.1735

Houdt, G. V., Mosquera, C., & Nápoles, G. (2020). A review on the long short-term memory model. *Artificial Intelligence Review*, 1-27.

Jeerige, A., Bein, D., & Verma, A. (2019). Comparison of deep reinforcement learning approaches for intelligent game playing. In *2019 ieee 9th annual computing and communication workshop and conference (ccwc)* (p. 0366-0371). doi: 10.1109/CCWC.2019.8666545

Kulić, D., Takano, W., & Nakamura, Y. (2008). Incremental learning, clustering and hierarchy formation of whole body motion patterns using adaptive hidden markov chains. *The International Journal of Robotics Research*, *27*(7), 761-784. Retrieved from `https://doi.org/10.1177/0278364908091153` doi: 10.1177/0278364908091153

Livne, D., & Cohen, K. (2020). Pops: Policy pruning and shrinking for deep reinforcement learning. *IEEE Journal of Selected Topics in Signal Processing*, *14*(4), 789-801. doi: 10.1109/JSTSP.2020.2967566

Mann, H. B., & Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The Annals* *of Mathematical Statistics*, *18*(1), 50–60. Retrieved 2022-11-19, from `http://www.jstor.org/stable/2236101`

Martinez, J., Black, M. J., & Romero, J. (2017). On human motion prediction using recurrent neural networks. In *2017 ieee conference on computer vision and pattern recognition (cvpr)* (p. 4674-4683). doi: 10.1109/CVPR.2017.497

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, *abs/1312.5602*. Retrieved from `http://arxiv.org/abs/1312.5602`

Monroy, G. A., Stanley, K. O., & Miikkulainen, R. (2006, July). Coevolution of neural networks using a layered pareto archive. In *Proceedings of the genetic and evolutionary computation conference* (p. 329-336). Seattle, Washington: New York, NY: ACM Press. Retrieved from `http://nn.cs.utexas.edu/?monroy:gecco06`

Ranstam, J. (2016, Jan). *Multiple p-values and bonferroni correction.* Elsevier. Retrieved from `https://www.oarsijournal.com/article/S1063-4584(16)00027-3/fulltext`

Zhao, D., Wang, H., Shao, K., & Zhu, Y. (2016, 12). Deep reinforcement learning with experience replay based on sarsa. In (p. 1-6). doi: 10.1109/SSCI.2016.7849837