Advanced Machine Learning (KSADMAL1KU): 1999 words (https://wordcounter.net/)

PAPER: PLAYING ATARI BASED ON HUMAN-LEVEL CONTROL THROUGH DEEP REINFORCEMENT LEARNING

Thor V.A.N. Olesen
Department of Computer Science
IT University of Copenhagen
Copenhagen, 2300
`tvao@itu.dk`

Alexander Ramos
Department of Computer Science
IT University of Copenhagen
Copenhagen, 2300
`aara@itu.dk`

Dennis Thinh Tan Nguyen
Department of Computer Science
IT University of Copenhagen
Copenhagen, 2300
`dttn@itu.dk`

August 3, 2021

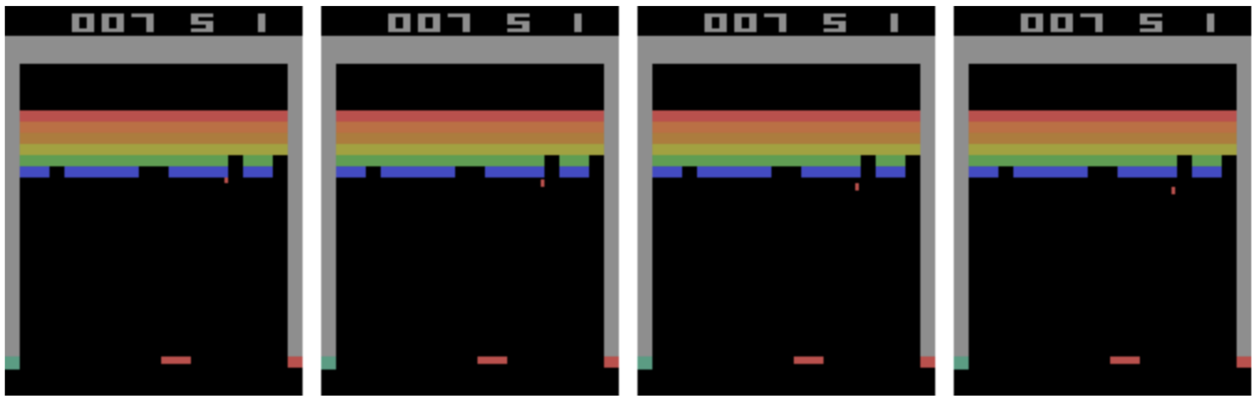


Figure 1: 4 consecutive Atari Breakout frames (single state observation)

1 Introduction

This project is based on the "H*uman-level control through deep reinforcement learning*" DeepMind paper [1], which demonstrates a system that can learn to play any Atari game from scratch better than humans, using only raw pixels as inputs and no prior knowledge of the game rules. The OpenAI gym library is used to represent the Atari environment and Tensorflow is used to implement the Deep Q-Network Agent (DQN) that does Approximate Q-Learning by applying Deep Learning methods (Convolutional Neural Networks) to the field of Reinforcement Learning (Q-Learning). This is improved with experience replay and target networks to create an agent that successfully learns to maximize rewards.

2 Methods

2.1 Preprossessing

A video frame from the Atari environment is represented as a 210 (height) x 160 (width) x 3 (channel) image. This is a very large state space so the image observations are converted to grayscale (1 channel), downsampled to 84 x 84 pixels and stored with the uint8 type to reduce the memory it takes to store all frames in memory. Notice that the images are only normalized with the float type right before doing predictions in the convolutional neural network model.

2.2 Reinforcement Learning: Q-learning

In **Reinforcement Learning (RL)**, an agent makes *observations* and takes *actions* within an *environment*, and in return it receives *rewards*. The objective is to learn to act in a way that will maximize its expected rewards over time:

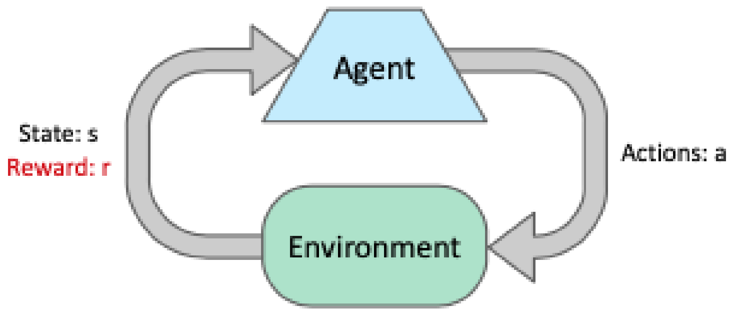


Figure 2: Reinforcement Learning Feedback Loop

The optimal value of a q-state $(s, a), Q * (s, a)$ (also known as the q-value of an action-state), is the maximum expected value of the future discounted total reward an agent receives after starting in s, taking action a, then acting optimally:

$$Q^*(s,a) = E[r + \gamma max_{a'} Q^*(s',a')|s,a] = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma max_{a'} Q^*(s',a')]$$

Q-learning is an active (feedback-based) model-free reinforcement learning algorithm that learns an optimal policy by watching an agent play (e.g. randomly) and gradually improving its estimates of the Q-values calculated from the Bellman equation that maximize the expected value of the total future discounted reward starting from the current state.

We use **Approximate Q-learning** by combining Q-learning with a deep convolutional neural network as a non-linear function approximator to estimate the Q-Value of any state-action pair (s,a) $Q^*(s,a) \approx Q(s,a,\theta)$ where θ is a vector of model parameters in the **Q-network**. A Deep Neural Network (DNN) used to estimate Q-Values is called a **Deep Q-Network** (DQN), and using a DQN for Approximate Q-Learning is called **Deep Q-Learning** [2].

According to the Bellman equation, we want the approximate Q-Value computed by the DQN for a given state-action pair (s,a) to be as close as possible to the reward r that we observe after playing action a in state s, plus the discounted value of playing optimally in the future. Thus, to estimate the sum of future discounted rewards, we can execute the DQN on the next state s' and get an approximate future Q-value for each possible action. In order to play, optimally, we discount the highest Q-value and set that to be the estimate of the sum of future discounted rewards. The target Q-value y(s,a) for the state-action pair (s,a) is obtained by summing the reward r and the future discounted value estimate:

$$Q_{target}(s,a) = r + \gamma \cdot max_{a'} Q_\theta(s',a')$$

This target Q-value is used to do a training step in the agent replay method using a Gradient-Descent algorithm (RMSProp in the paper). We then strive to minimize the squared error between the estimated Q-value $Q_\theta(s,a)$ and the target Q-Value (the Huber loss [3] is used to reduce sensitivity to large errors) at each training iteration i:

$$L_i(\theta_i) = E_{(s,a,r,s')\sim U(D)}[(r + \gamma \cdot max_{a'} Q(s',a',\theta_i^-)) - Q(s,a,\theta_i))^2]$$

where the target Q-value is $y_i = r + \gamma \cdot max_{a'} Q(s',a',\theta_i^-)$ and the approximated Q-value is $f_i(x) = Q(s,a,\theta_i)$

Experience replay is done by storing experiences (s,a,r,s') at each time step t in a data set $D_t\{e_1,...,e_t\}$. to make gradient updates less expensive and speed up training by using a minibatch of 32 experiences, which are drawn uniformly from the replay memory, $\sim U(D)$. During learning, Q-learning updates are applied on these samples. The Q-learning update at iteration i uses the loss function above. The advantage of the experience replay buffer is that the agent does not forget previous experiences and the sampling ensures that experiences are less correlated to cope with their inherent sequential order.

The paper uses two DQNs instead of one; the first network is the *online model* that learns at each step and is used to predict Q values, and the other network is the **target network model** used only to define the targets to stabilize training. A replay buffer of 1 million experiences is used to reduce correlation between experiences and avoid *catastrophic forgetting* where the agent starts to replace what it learned in the past with what it learns now in policy updates. The agent follows an **ϵ-greedy policy** to incentivize it to initially explore and later exploit by decaying the exploration rate over time.

2.3 Convolutional Neural Networks (CNNs)

CNNs solve the problem of the large parameter space in deep neural networks with fully connected layers for image recognition tasks by using partially connected layers and weight sharing. The *convolutional layer* is a layer in the neural network where neurons are not fully connected with the input image, but only to pixels in their receptive fields (local regions). The architecture helps the network focus on low-level features in the first hidden layer, gradually making them higher-level features in the following hidden layers [2].

Images from the Atari environment are inherently non-linear and hierarchical so we use a deep convolutional neural network to detect regional patterns and a nonlinear ReLu activation function $max(0, x)$ to introduce nonlinearity. A **filter** is then a small set of learn-able weights that store a single pattern represented by a small image. The size of the receptive field is the region of the input space that affects a particular unit of the network. A set of filters (*convolution kernels*) is used to slide over the original image to learn different hierarchical structures in the image. *Each filter convolves (slides) over every X by X block of image pixels and computes the dot product of the filter matrix with pixel block values as the output*. The output is a **feature map**, which highlights areas in an image that activate the filter the most, which are combined into more complex patterns.

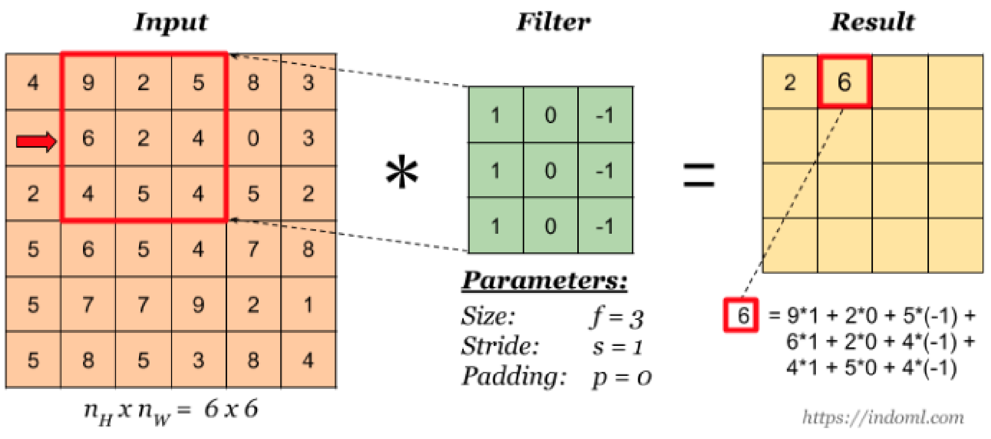


Figure 3: Filter convolves or slides over image to learn a local region of important features. Taken from `https://www.coursera.org/learn/convolutional-neural-networks`

Our convolutional neural network takes in a preprocessed input of 84 x 84 x 4 stacked frames and consists of three convolution layers and two fully connected layers. The exact structure, filters, strides and padding is shown below:

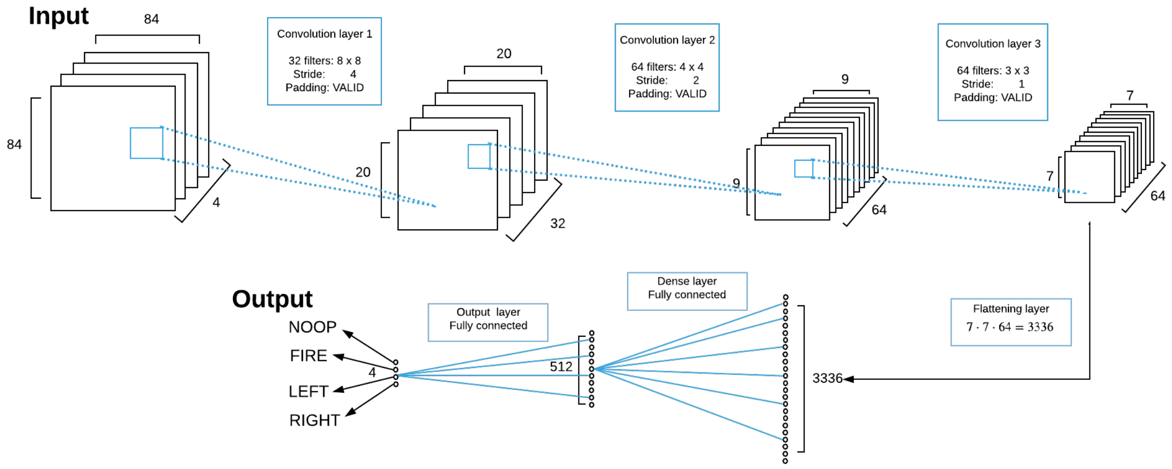


Figure 4: Convolutional Neural Network Architecture

3 Experiments

We tested our implementation using same hyper parameter settings as in the *DeepMind* paper on Atari Breakout. Furthermore, we experimented with the choice of weight initialization, learning rate, optimizer, replay memory size.

General Setup and Constraints

The experiments were run on *Google Colab*, which provides free access to Tesla K80 GPUs. Unfortunately, we are restricted to using it a maximum of 12 consecutive hours. However, we were able to run the experiments for an extended period of time beyond 12 hours using a custom save and restore feature. Nonetheless, we were only able to train our models for $\sim$ 36 consecutive hours with possible disconnects and replay memory issues, whereas *DeepMind* spent 38 days = 912 hours training on 50 million frames. In 912 hours *DeepMind* trained 200 epochs, which means we approximately trained $(36/912)$ hours $\cdot\, 200$ epochs ≈ 7.99 epochs. *DeepMind* does not provide a graph of average reward over time for Breakout in their *Human-Level Control* paper so we used the one from "*Playing Atari with Deep Reinforcement Learning*" [4] to compare our results in a more meaningful way.

3.1 Experiment A

Experiment A replicates the hyper parameters of *DeepMind* [1]. We trained for $\sim$36 hours $\approx$ 8 epochs using *Google Colab*. We achieved a maximum reward of 15, and saw a steady increase in average reward. However, at the time in which the experiment was performed, we were unable to store the agent's replay buffer after a Colab interruption.

3.2 Experiment B

Experiment B uses the Adam optimizer in addition to the original hyper parameters, which is an extension of RMSProp and AdaGrad. This provides a more stable optimizer by adapting a learning rate to each network parameter as training unfolds [5].We use the Xavier [6] weight initializer that controls the variance of our initial weight values to avoid the vanishing gradient problem. The model was trained for $\sim$24 hours $\approx$ 6 epochs using *Google Colab*. We were able to store the replay buffer and hence restore the Agent's replay buffer in the event of system failures to mitigate *catastrophic forgetting*. We achieved a maximum reward of 10 and saw a steady increase in average reward.

3.3 Best Hyper parameters

The hyper parameters that yielded best overall results are similar to the ones found in the paper except that we used the Adam optimizer, which proved to be more stable.

- Minibatch Size: 32
- Replay memory size: 1,000,000
- Agent history length: 4
- Target network update frequency: 10000 (the frequency at which the Primary Deep Q-Network trainable parameters are copied to the Target Deep Q-Network)
- Discount factor: 0.99
- Action repeat: 4 (frame skip)
- Learning rate: 0,00025
- Initial exploration: 1
- Final exploration: 0.1
- Final exploration frame 250,000 (instead of 1 million due to 4 frame skip)
- Replay start size 50000
- No Op Max: 30 (maximum number of "do-noting" actions to be performed by agent at the start of an episode)

4 Discussion

4.1 Results

Figure 5 shows average reward over time, for DeepMinds early version. The red square in *DeepMinds* plot roughly indicates the time frame displayed in our experiments.

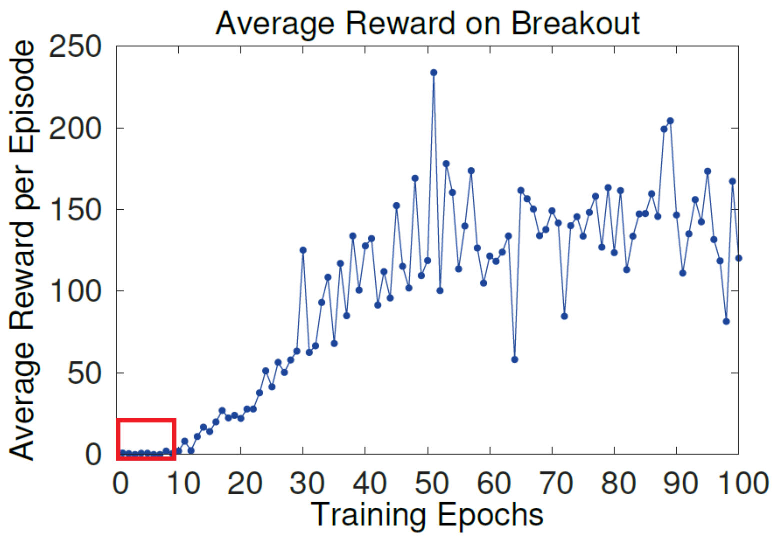
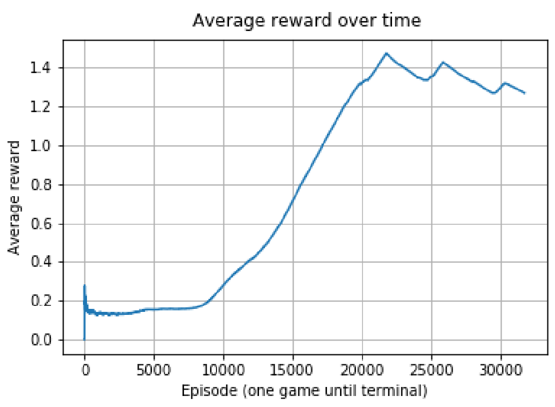
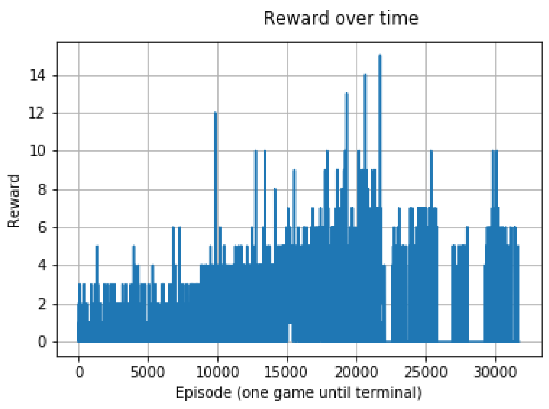


Figure 5: Average reward over epochs from DeepMind [4] - We lie within red box

Figure 6 shows average reward over time (on the left) and rewards over time (on the right) from experiment A. Note that it trains for 35.000 episodes which corresponds to ∼8 epochs



(a) Average reward over episodes

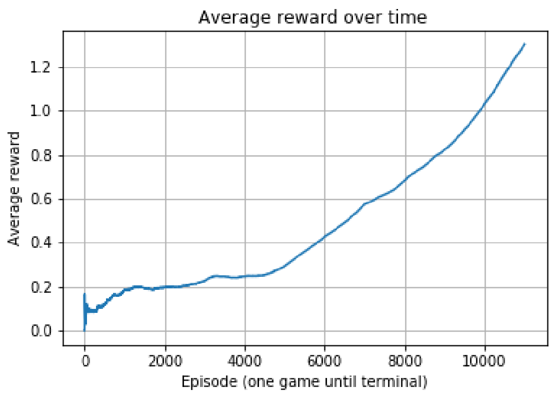


(b) Reward over episodes

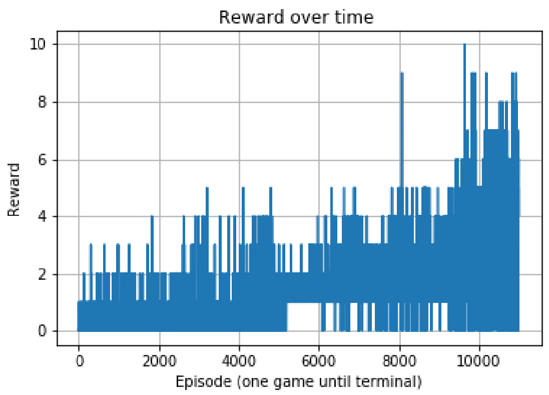
Figure 6: Experiment A

Comparing the model in experiment A's with DeepMind's, it has not trained long enough to achieve major improvements in reward gain. However, after approximately 6000 episodes the average reward starts to increase and grows exponentially after 10000 episodes. Looking at rewards in figure 6b, the agent starts with random play, achieving between 0-4 rewards. As the agent learns, it gradually becomes better and consistently achieves 6-10 rewards, peaking at 15. The sudden decrease in rewards around 24000 episodes is due to system failures that results in the replay buffer being lost, as we were unable to store the replay buffer for this experiment. Thus, the agent is forced to gather new experiences upon restart, which most likely causes catastrophic forgetting as its most recent experiences are lost.

Figure 7 shows average reward over time (on the left) and rewards over time (on the right) from experiment B. Note that it trains for 10000 episodes which corresponds to ∼6 epochs



(a) Average reward over episodes



(b) Reward over episodes

Figure 7: Experiment B

Similar to experiment A, the model was not trained long enough to gain any substantial rewards. However, the average reward increases much earlier and faster compared to experiment A. This is most notable around episode 8000 where the average reward for experiment B is roughly 0.7 in contrast to experiment A, the same average reward is reached around episode 15000. This may be due to the use of the Adam optimizer, and the ability to restore the replay buffer in experiment B hence avoiding catastrophic forgetting which likely occurred in experiment A. The agent seems to find a policy with guaranteed reward beyond 1 around episode 5000-6000.

4.2 Improvements

The *Rainbow: Combining Improvement in Deep Reinforcement Learning* paper suggests a series of further improvements of the DQN including *Double Q-Learning* proposed by Hasselt et. al.(2010) [7], *Prioritized Experience Replay* (Schaul 2016 [8]) and *Dueling Networks* (Wang 2015) [9].

5 Conclusion

We have successfully implemented the Deep Reinforcement Learning algorithm by DeepMind used to teach an agent to play Atari Breakout from raw pixel inputs only. This was done by combining Q-Learning with Deep Neural Networks in order to construct a Deep Q-Network to do Approximate Q-Learning given the large state space. We used Experience Replay and a Target Deep Q-Network to cope with instabilities of using a nonlinear neural network function approximator to represent the action-value Q function. The agent is capable of improving its policy over time, but due to lack of efficient hardware and time restrictions, we were unable to replicate *DeepMinds* results. Future improvement relies on access to optimized hardware and longer training times since most of the promising results first occur after weeks of training. We have realized how hard it is to reproduce this kind of breakthroughs and find it concerning that the scientific results in Machine Learning are hard to reproduce. In conclusion, Reinforcement Learning is notoriously difficult because of the training instabilities, scalability issues, and huge sensitivity to the choice of hyperparameter values.

References

[1] Kavukcuoglu K. Silver D. et al. Mnih, V. Human-level control through deep reinforcement learning. *Nature*, 2015.

[2] Richard S. Sutton and Andrew G. Barto. In *Reinforcement Learning (2nd edition, 2018)*, page 349. The MIT Press, 2018.

[3] Robert; Friedman Jerome Hastie, Trevor; Tibshirani. In *The Elements of Statistical Learning*, page 349. Springer, New York, 2009.

[4] Kavukcuoglu K. Silver D. et al. Mnih, V. Playing atari with deep reinforcement learning. 2013.

[5] Jimmy Ba Diederik P. Kingma. Adam: A method for stochastic optimization. 2015.

[6] Yoshua Bengio Xavier Glorot. Understanding the difficulty of training deep feedforward neural networks. 2010.

[7] Guez A. Silver D Hasselt, H. Deep reinforcement learning with double q-learning. 2010.

[8] Quan J. Antonoglou D. Silver David Schaul, T. Prioritized experience replay. 2010.

[9] Schaul T. Hessel M. Hasselt H. Lanctot M . Freitas N Wang, Z. Dueling network architectures for deep reinforcement learning. 2015.

6 Appendix

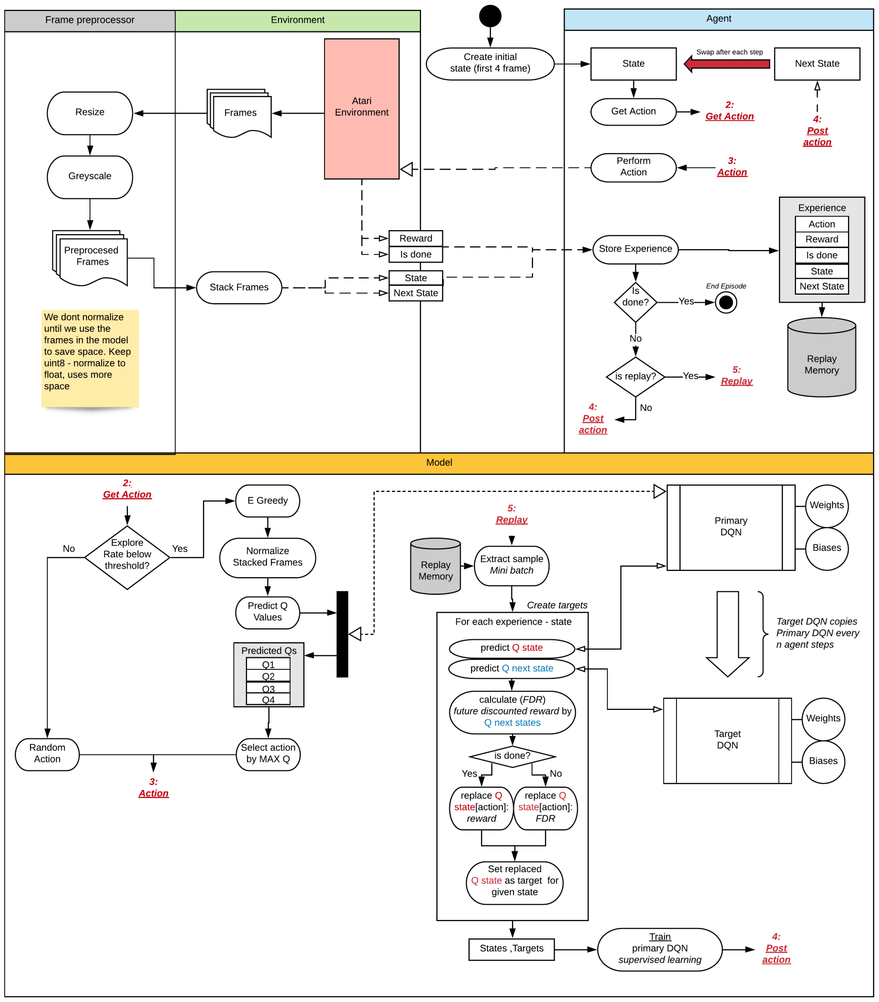


Figure 8: Flow chart of the learning process

Code snippets

Layers

```
def convolutional_2d_layer(self, inputs, filter_weights, biases, strides=1):
    output = tf.nn.conv2d(self, inputs, filter_weights, strides, padding=PADDING)
    output_with_bias = tf.nn.bias_add(output, biases)
    activation = tf.nn.relu(output_with_bias)
    return activation
```

```
def dense_layer(self, inputs, weights, biases):
    output = tf.nn.bias_add(tf.matmul(inputs, weights), biases)
    output_with_bias = tf.nn.bias_add(output, biases)
    dense_activation = tf.nn.relu(output)
    return dense_activation
```

```
def output_layer(self, input, weights, biases):
        linear_output = tf.nn.bias_add(tf.matmul(input, weights), biases)
        return linear_output
```

Training

```
def train(self, inputs, targets):
        with tf.GradientTape() as tape:
            predictions = self.predict(inputs)
            current_loss = self.huber_error_loss(predictions, targets)

        trainable_variables = list(self.weights.values()) + list(self.biases.
                                                 values())

        gradients = tape.gradient(current_loss, trainable_variables)

        optimizer.apply_gradients(zip(gradients, trainable_variables))

        return tf.reduce_mean(current_loss)
```

Experience Replay

```
 def replay(self):
        experiences = self.experiences.sample(MINIBATCH_SIZE)
        states, actions, rewards, next_states, dones = experiences

        next_Q_values = model.predict(states)
        next_Q_values_target = model.predict(next_states, is_target = True)

        current_q_predictions = self.model.predict(states)
        next_q_predictions_target = self.model.predict(next_states, is_target=True
                                                 )

        targets = np.array(shape=(MINIBATCH_SIZE, self.number_of_actions))

        for i, (state, action, current_reward, next_state, done) in memory_batch:
            current_Q_values = next_Q_values[i]
            next_Q_values_target = next_Q_values_target[i]
            future_discounted_reward = DISCOUNT_FACTOR * tf_reduce_max(
                                                    next_q_values_target)
            current_q_values[action] = current_reward if done else current_reward
                                                    + future_discounted_reward
            targets[i] = current_q_values
        average_loss = self.model.train(states, targets)
        return average_loss
```