Advanced Machine Learning (KSADMAL1KU): 3705 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`

December 19, 2019

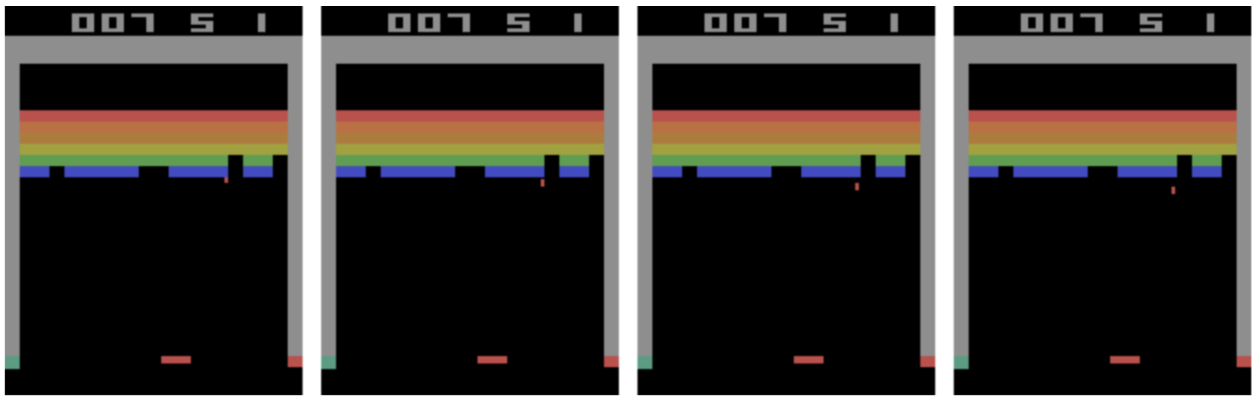


Figure 1: 4 consecutive Atari Breakout frames (single state observation)

1 Introduction

The 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 (see appendix 6.1). The environment in which the agent is placed may subject the agent's actions to being **nondeterministic**, which means there may be multiple successor states that can results from an action taken in some state. This kind of problem where the world poses a degree of uncertainty and the agent has to plan (search) for multiple outcomes of its actions that may fail, is a *nondeterministic search problem* and can be solved with models known as **Markov decision processes** (**MDP**).

The act of playing Atari Breakout is a sequential decision making problem that can be formalized as such a discrete-time finite Markov Decision Process (MDP). Here, the set of states is the set of all possible preprocessed frames ($255^{84\times84}$), $A = \{0 : \text{NOOP}, 1 : \text{FIRE}, 2 : \text{LEFT}, 3 : \text{RIGHT}\}$ is the set of actions, the initial state s_0 is the initial frame, the terminal state(s) G is the set of frames in which the agent dies or has destroyed all bricks, $r \in \{0, 1, 2, 4, 7\}$ is the set of rewards and $\gamma = 0.99$ is the discount factor. The MDP is then an extension of Markov chains that adds actions (allowing choice) and rewards (giving motivation) to support the agent's goal of taking an action that will maximize rewards over time

Unlike supervised learning, rewards are often **delayed** so the reward of an agent action do not depend on a single decision performed in a state, but rather on the whole sequence of an agent's actions. The **discount factor** and a **finite horizon** is used to place time constraints on the number of timesteps for which the agent can take actions and collect rewards. The agent is given a "lifetime", which is some number of timesteps or episodes (games) to accumulate as much reward as possible before being automatically terminated. The discount factor is used to model an exponential decay in the value of rewards over time. The discount factor determines how much the agent cares about rewards in the future where a value of 0 favors immediate reward and a value of 1 will go on for future rewards which may lead to infinity.

Markov decision processes satisfy the **Markov property** (memoryless), which states that the future and past are conditionally independent, given the present:

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ..., a_0, s_0) = P(s_{t+1}|s_t, a_t)$$

Where $s_0...s_t, a_0...a_t$ are state action sequences.

The memoryless probabilities above are encoded by the transition function so $T(s, a, s') = P(s'|s, a)$, which represents the probability that an agent taking an action $a \in A$ from a state $s \in S$ ends up in a state $s' \in S$ (s_{t+1}).

Where other MDPs may be partially observable, Atari Breakout is fully observable and we use a version that is deterministic, meaning the agents actions are guaranteed to succeed. This transforms our MDP transition function from a probability function to a simple mapping from a state and an action to another state $T : SA \to S'$.

Solving an MDP means finding an optimal **policy** $\pi^* : S \to A$, a function mapping each state $s \in S$ to an action $a \in A$, such that it yields the maximum expected total reward. The MDP is solved using the **Bellman equation**, which recursively describes the expected rewards. The Bellman equation is a **dynamic programming** equation, since it decomposes a problem into smaller subproblems through its inherent recursive structure. The goal is acting in a way that maximizes future discounted rewards:

$$R_t = \sum_{t=0}^{T} \gamma^t \cdot R(s_t, a_t, s_{t+1}),\ 0 \leq \gamma < 1$$

The optimal value of a state s, $V*(s)$, is the maximum expected value of the future discounted total reward an optimally-behaving agent that starts in s will receive, over the rest of its lifetime where $T(s, a, s') = P(s'|s, a)$:

$$V^*(s) = max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V*(s')]$$

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:

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

This allows us to reexpress the Bellman equation:

$$V^*(s) = max_a Q*(s,a)$$

In a stochastic environment, the reward that the agent can expect to receive on average conditioned on taken action a in state s is an expectation. This becomes a weighted sum over all future states (outcomes) with discounted rewards multiplied transition probabilities.

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.

Model-free learning attempts to estimate the q-values of states directly, without using any memory to construct a model of the rewards and transitions in the MDP. Q-learning learns the q-values of states directly, bypassing the need to know any values, transition functions or reward-functions. Q-learning iteratively updates the action q-values function by using Bellman equation to perform **q-value iteration** by computing Q_{t+1} for all states s of the current time step t in the environment.

Q-learning will acquire **q-value samples**: $sample = R(s,a,s') + \gamma max_{a'} Q(s',a')$ during agent play and do simple value iteration updates, using the weighted average of the old value and new learned value:

$$sample = R(s,a,s') \gamma max_{a'} Q(s',a')$$

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \cdot sample$$

Assuming we spend enough time in exploration and decrease the learning rate α appropriately, Q-learning learns the optimal q-values for every q-state even by taking suboptimal or random actions (**off-policy learning**), $Q_t \to Q^*$ as $i \to \infty$. However, we need to do this for each sequence of actions and states, which is intractable given the exponentially large state space that we need to store and compute q-values of, since we do not have enough memory and cannot visit all states during training. Thus, 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**. This was previously done by extracting linear combinations of handcrafted features from the state to estimate Q-Values. However, DeepMind showed in this paper that using deep neural networks works much better for complex problems and it does not require any feature engineering. 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 actually 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 $f_i(x) = Q(s,a,\theta_i)$

In order to make gradient updates less expensive and speed up training, a sample or minibatch of 32 observations is used instead of the full expectation. The experiences $e_t(s_t, a_t, r_T, s_{t+1}$ at time step t consist of a state s, action a, reward r and next state s' that are drawn randomly from a replay memory of stored samples (experiences). This is made possible by **experience replay** where the agent's experiences are stored at each time-step t in a data set $D_t\{e_1, ..., e_t\}$. During learning, Q-learning updates are applied on the samples (or mini-batches) of experience $(s,a,r,s') \sim U(D)$, drawn uniformly at random from the pool of stored 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, since the video frames are otherwise extremely correlated due to their inherent sequential order.

Finally, in the original DeepMind paper on Deep Reinforcement Learning with Atari, the neural network model was used to both make predictions and to set its own targets. This is analogous to a dog chasing its own tail which can make the network unstable. The solution in the paper, is to use two DQNs instead of one; the first network is the *online model* that learns at each step and is used to move the agent around, and the other network is the **target network model** used only to define the targets. In the paper, the target model is only updated every 10.000 steps to stabilise the Q-value targets, a tiny learning rate of 0.00025 is used to stabilize training and a very large replay buffer of 1 million experiences is used to reduce the 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 its policy updates.

The agent follows an **ϵ-greedy policy** based on some probability $0 \leq \epsilon \leq 1$, and acts randomly to explore with probability ϵ. This ensures that the environment is sufficiently explored by forcing the agent to switch between following its current policy (exploit) with probability $(1 - \epsilon)$ or acting randomly to explore the environment with probability ϵ. The exploration rate is decayed over time to incentivize the agent to exploit after sufficient exploration.

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 the neurons are not connected to every single pixel in the input image, but only to pixels in their receptive fields (local regions). In continuation, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer. As a result, 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. This is great for image recognition, since images have shaped objects, edges, lines, colors and pixels are usually more similar to neighbors pixels in local regions of the image [2].

In our case, sensory image data in the form of raw pixels from the Atari game environment are fed to the model, whose patterns 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 to the model. A **filter** is then a small set of learnable weights that store a single pattern represented by a small image, the size of the receptive field, which is the region of the input space that affects a particular unit of the network. A set of filters (or *convolution kernels*) is used to slide over the original image data to learn different hierarchical structures in the data. **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 the areas in an image that activate the filter the most, which are combined into more complex patterns. The **stride** is the number of pixel shifts over the input and **zero padding** may be used to to retain the dimensions of the input by adding zeros around the inputs. This is done before the convolution, which is the process by which a filter is moving over the image to find the most important features. **Max pooling** may be used to further down-sample the input by reducing its dimensionality using a max filter.

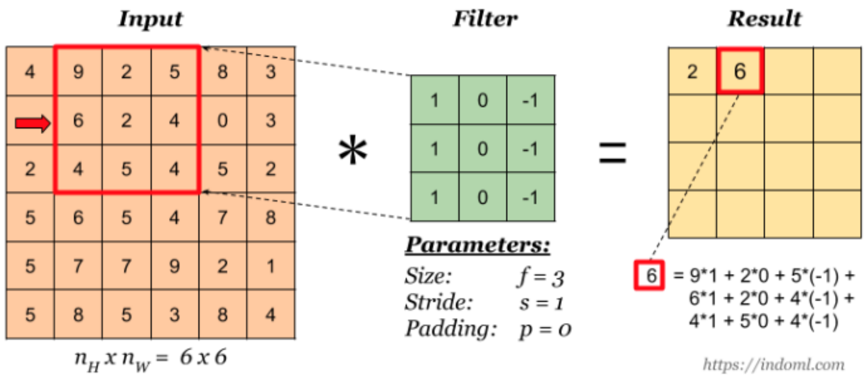


Figure 2: Filter convolves or slides over image to learn a local region of important features. Taken from `https://www.coursera.org/learn/convolutional-neural-networks`

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 in appendix (6.2).

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 a Tesla K80 GPU. 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 their earlier paper "*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 (Root Mean Squared Propagation) and AdaGrad (Adaptive Gradient Algorithm). This provides a more stable optimizer by adapting a learning rate to each network parameter as training unfolds [5]. In addition, we use the Xavier (GlorotUniform version) or He (VarianceScaling with scale 2 for ReLU) weight initializer that controls the variance of our initial weight values to avoid the vanishing (or exploding) gradient problem. Unfortunately, the choice of weight initializer is not mentioned in the chosen paper but recent results indicate that the Xavier [6] or He [7] initializers work particularly well by reducing the variance of the weights and the negative saturation of the ReLU activation function. The model was trained for $\sim$24 hours $\approx$ 6 epochs using *Google Colab*. In this experiment, we were able to store the replay buffer and hence we were able to 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. Again, we compare our results of Breakout with those of DeepMinds early version.

3.3 Best Hyper parameters

The hyper parameters that yielded best results overall are similar to the ones that can be 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 3 shows average reward over time, for DeepMinds early version. The red square in *DeepMinds* plot roughly indicates the time frame displayed in our experiments.

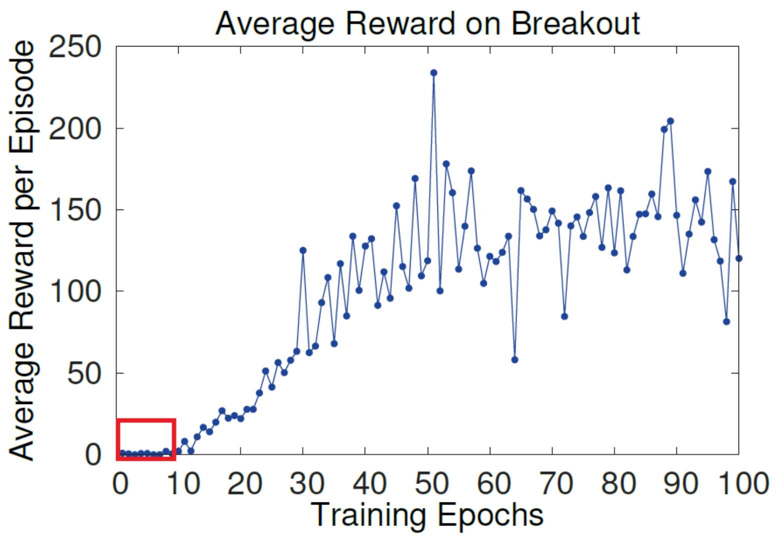
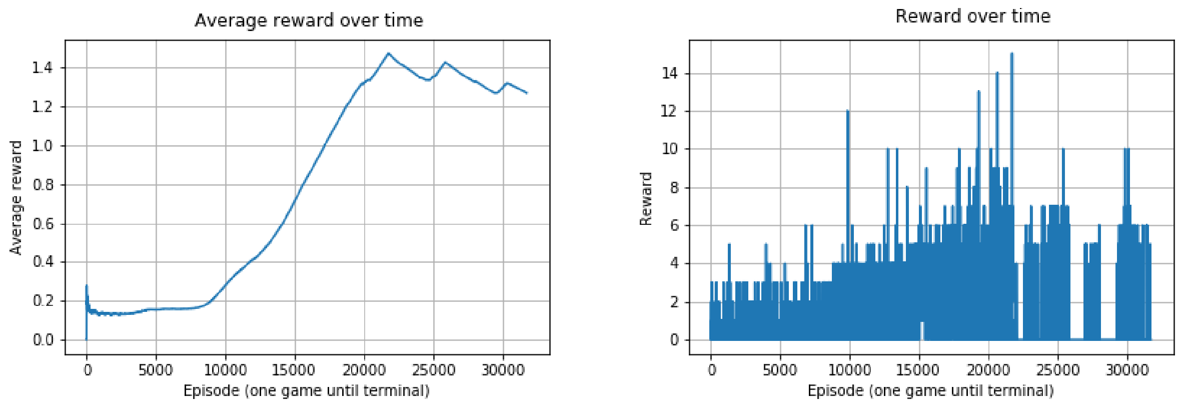


Figure 3: Average reward over epochs from DeepMind [4] - We lie within red box

Figure 4 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



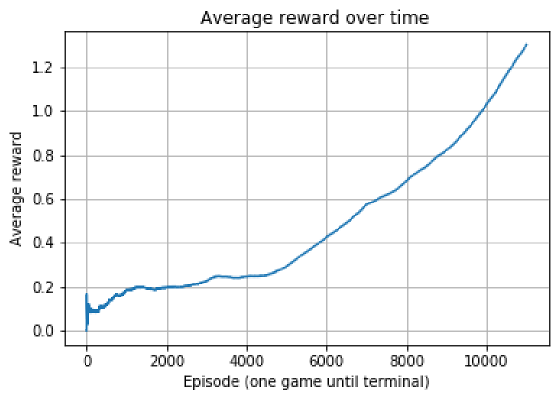
(a) Average reward over episodes

(b) Reward over episodes

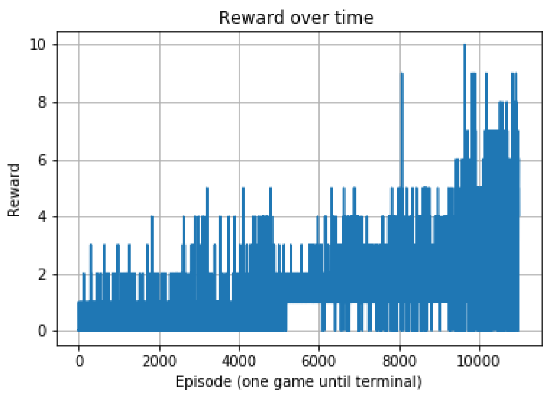
Figure 4: Experiment A

Comparing the model in experiment A's with DeepMind's, it's clear that 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 4b we see that it 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 both plots around 24000 episodes is due to system failures and hence the replay buffer is 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 5 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 ∼8 epochs



(a) Average reward over episodes



(b) Reward over episodes

Figure 5: Experiment B

Similar to experiment A, the model was not trained long enough to have any substantial reward gains. 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 which reaches the same average reward around episode 15000. The reason may partly be due to the use of the Adam optimizer, and partly 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.

Loss as performance metric

The loss is a poor indicator of the model, since it might decrease, yet the agent perform worse. This happens when the agent gets stuck in a small region of the environment by over-fitting. On the other hand, the loss may go up, yet the agent might perform better. This happens when the DQN is underestimating the Q-values, and starts correcting them by increasing its predictions. As a result, the agent may get better rewards although the loss may increase because the DQN sets the targets, which become larger too. ([8] p. 638).

4.2 Improvements

After *DeepMinds* second DQN paper in 2015, they published "*Rainbow: Combining Improvement in Deep Reinforcement Learning*" [9] in 2017, which further improves the Atari DQN using new Reinforcement Learning methods.

Double Q-Learning

Double Q-Learning proposed by Hasselt et. al.(2010) [10] addresses the issue of the maximization bias introduced by repeatedly taking the maximum of estimated values from a DQN. The suggested solution is to use two separate Q value estimators, which are used to update each other. The result is a decoupling of the action selection and its evaluation, ultimately reducing harmful overestimation, which is present in our implementation.

Prioritized Experience Replay

We uniformly sample experiences from memory, which is not very efficient. Instead we want to prioritize more recent experiences, from which there is much to learn. Scahul et. al. (2015) proposes Prioritized Experience Replay which sample experiences from memory based on their "learning potential". Newer experiences are inserted into memory with higher priority, meaning there's a higher probability for these to be sampled.

Reward clipping

We have not enabled reward clipping to generalize across other Atari 2600 games, since it may render the agent unable to distinguish between rewards of different magnitudes [1]. Thus, we decided not to enable this as our main focus was to first successfully implement a DQN in the Atari Breakout game.

5 Conclusion

In this project we have successfully replicated the implementation of the DeepMind Reinforcement Learning algorithm used to teach an artificial 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 enable Approximate Q-Learning given the large state space. We used Experience Replay and a Target Deep Q-Network to cope with the instabilities of using a nonlinear neural network function approximator to represent the action-value Q function. Our results show that our 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 would rely on access to hardware optimized for Machine Learning models and a longer training time frame, since most of the promising results first occur after weeks of training. Based on our findings, we have realized how hard it is to reproduce these kind of academic breakthroughs and find it concerning that these kind of scientific results in the world of 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] Shaoqing Ren Jian Sun Kaiming He, Xiangyu Zhang. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. 2015.

[8] Aurélien Géron. In *Hands-On Machine Learning with Scikit-learn, Keras, and TensorFlow - Second Edition*. O'Reilly, 2019.

[9] Modayil J. Hasselt H. et al. Hessel, M. Rainbow: Combining improvements in deep reinforcement learning. 2017.

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

6 Appendix

6.1 RL Loop

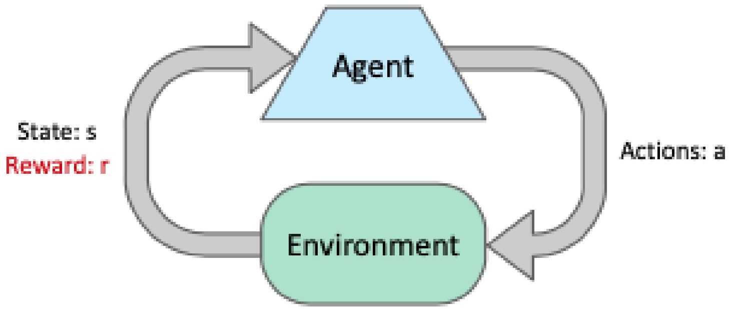


Figure 6: Reinforcement Learning Feedback Loop

6.2 Convolutional Neural Network Structure

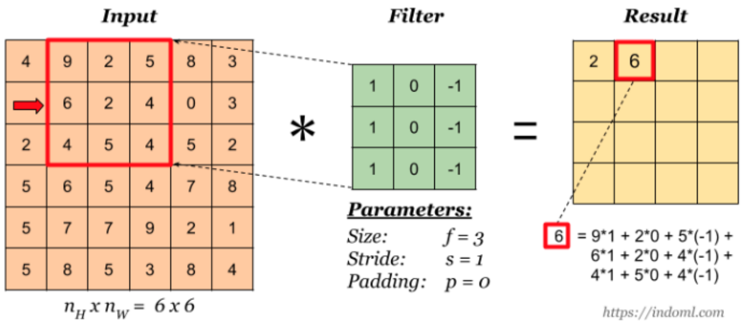


Figure 7: Filter convolves or slides over input image to learn important hierarchical features

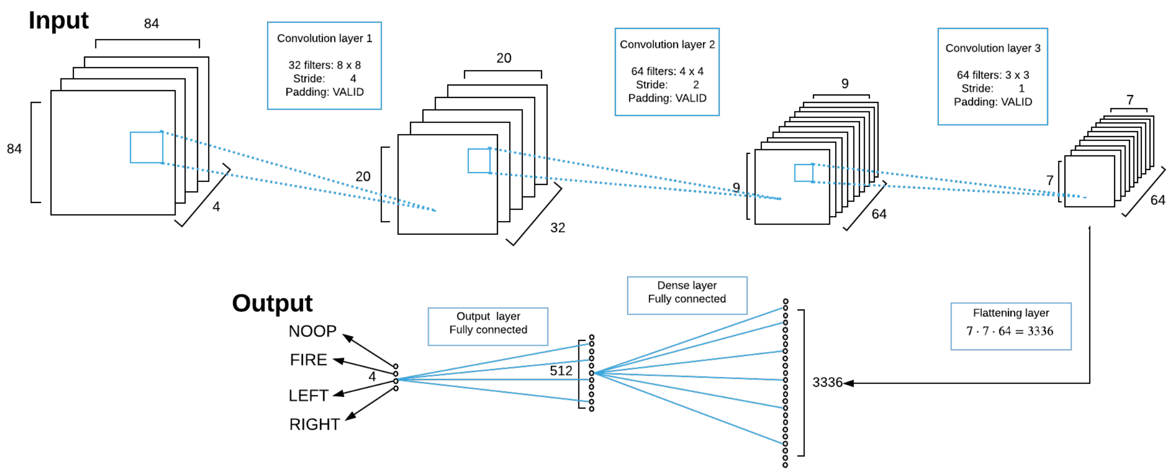


Figure 8: Convolutional Neural Network Architecture

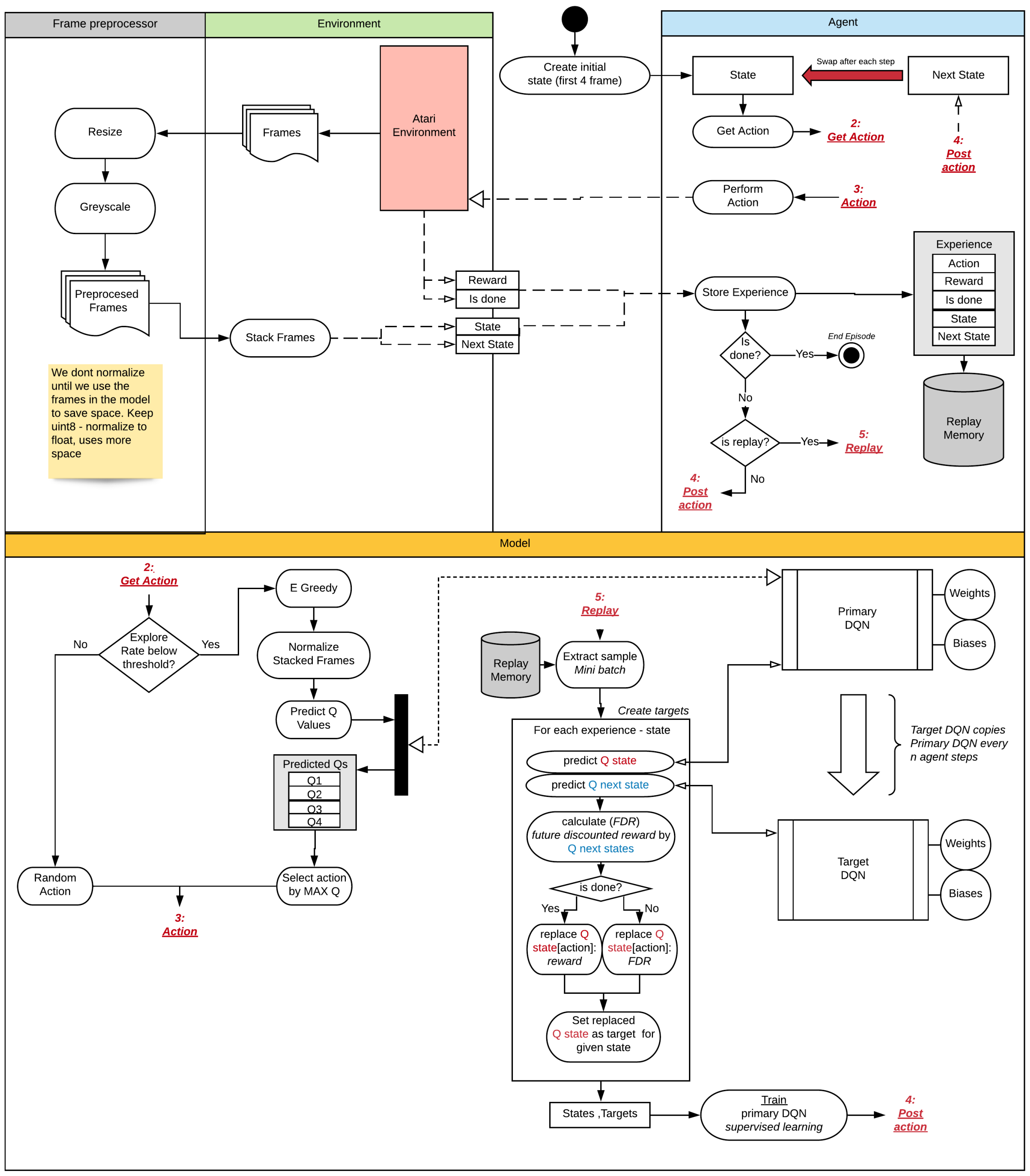


Figure 9: Flow chart of the learning process