Converging towards optimal policy using Deep Reinforcement Learning

Case study: Atari Breakout

Milan Djuric
Department for Computing and Control
Faculty of Technical Sciences
University of Novi Sad
millan.djuric@hotmail.com

***Abstract* – This paper represents three different methods for training a reinforcement learning agent to play a challenging game of Atari Breakout. The methods covered are deep Q network, double deep Q network and dueling architectures. All three algorithms provide an end-to-end reinforcement learning solution in which an agent learns directly from pixels. Due to the hardware limitations, each model was trained over only 2 million frames, which is considerably less then what was reported in some state-of-the-art solutions. The average reward per game over last 100 episodes was used as the evaluation metric. I conclude that the single stream methods (deep Q network and double deep Q network) are more efficient and adequate for Atari Breakout as they obtain an average reward that is nearly double the size of the average reward obtained by dueling architectures after the same number of training frames.**

***Keywords* – Approximate Q Learning, Deep Q Network, Double Deep Q Network, Dueling Architectures, Convolutional Neural Networks**

I. INTRODUCTION

One of the core topics in machine learning is the problem of sequential decision making (SDM). In SDM, an agent needs to learn a sequence of actions to perform in an environment in order to achieve certain tasks. For many years, the most popular framework that offers a formalization of SDM problems has been reinforcement learning (RL).

The main reason for RL popularity in recent years was the ability of RL algorithms to employ state-of-the-art computer vision techniques. Recent advances in computer vision and deep learning have made it possible to extract features from high-dimensional input such as images and video. This has considerably extenuated and improved the process of training a RL agent, since the responsibility of feature detection and design has shifted from humans to a deep neural network. Unfortunately, training a neural network for a reinforcement learning problem imposes several additional challenges that are usually not present in the context of supervaised learning. In part III, we provide an in-depth explanation of these challenges, as well as the proposed solutions.

In this paper, we will demonstrate three different RL algorithms to train our agent to play Atari Breakout. These algorithms are known as deep Q network (DQN, A), double DQN (G) and dueling architectures (H). They provide an end-to-end RL solution: the agent learns to play the game directly from visual inputs in form of pixels, without any information or domain knowledge previously being hardcoded to our model. All three solutions are model-free, off-policy and online methods (Table 1) that rely on deep neural networks as Q-function approximators. They employ state of the art computer vision techniques, such as convolutional layers. I also utilize the ideas such as *experience replay* (C) and *target network* (F) to achieve greater training stability (Mnih et al. 2013 [3]).

To test the three models, we calculate the average reward of each model during the last 100 evaluation episodes, after initially training every model for a total of 2 million frames. The results suggest that the single stream methods (DQN and double DQN) outperform the dueling architectures. I conclude that the gap in the score between the algorithms is explained by the small number of actions available in Atari Breakout, which is 4. With such a small action space, the dueling architectures algorithm does not create any significant benefit. On the contrary, it introduces more computational overhead, as well as the increased potential for overfitting.

TABLE 1: REINFORCEMENT LEARNING TERMINOLOGY

model-free	Opposite of model-based. This means that our agent does not use any approximation of the transition nor the reward function of the underlying Markov decision process.
off-policy	Opposite of on-policy. This means that our agent does not start with a given policy (a mapping from state to actions) and tries to evaluate it, but rather finds one by following a greedy strategy of Q-value calculation in the landing state (the max operator in the Bellman equation).
online learning	Opposite of offline. This means that we train using a minibatch and not the full batch of training information.

II. BACKGROUND

In traditional RL, the environment is represented as a set of states, where each state is associated with a set of valid actions the agent can perform. By performing an action a in state s, the agent receives a feedback signal from the environment in form of a reward r and next state s'. The overall goal of the agent is to learn the best action to perform in each state of the environment. The best action is one that maximizes the expected cumulative reward. We refer to the quality of taking a certain action *a* in state *s* as the Q-value (or Q-function, *Q (s, a)*).

In most scenarios, RL problems usually involve an agent that initially has no knowledge about the environment and learns by means of trial-and-error. Unfortunately, the environment is often represented as a high-dimensional (possibly even continuous) state-action space that cannot be fully visited by our agent, let alone thoroughly explored. In such a setting, many traditional RL solutions, such as the *online Q-learning* (Watkins and Dayan, 1992 [1]), become inapplicable, due to the fact that they rely on dynamic programming techniques on in-memory data structures that store the learned Q-values of each state. These solutions cannot scale well for the aforementioned environments, such as the environment of Atari Breakout. It quickly becomes obvious that in order to solve problems that include a high-dimensional environment, we need to be able to generalize between different states of the environment. This idea gives rise to a new family of *Q-learning* algorithms known as *approximate Q-Learning.*

In classical *approximate Q-Learning*, we define our Q-function as a linear combination of handcrafted features. By doing so, we manage to reduce the dimensionality of our environment by mapping different state-action pairs to the same Q values. The goal of our algorithm now is to learn the optimal weights of these features. There are two major deficiencies of this approach. Firstly, due to the fact that the features are handcrafted, the quality of our Q-function, and therefore the success of our algorithm, becomes heavily dependent on our domain knowledge. For many problems, even the greatest domain expertise would not be enough to design and encode usable, efficient features. Secondly, many complex problems are much better described with nonlinear relationships. To tackle these issues, we resort to a special class of *approximate Q-learning* algorithms called *deep reinforcement learning* (DRL).

III. Deep Reinforcement Learning

In essence, all DRL algorithms employ the central idea introduced by *approximate Q-Learning*, and that is to approximate the Q-function. However, DRL algorithms attempt to overcome the deficiencies of previous solutions by learning a nonlinear function approximator, such as a deep neural network. By parametrizing our Q-function with the weights of a deep neural network, we refine the idea of a whole class of *approximate Q-learning* algorithms. Instead of relying on handcrafted features for our Q-function representation and using *Q-learning* only to learn the weights of these features, we use a deep neural network to identify both the features and their associated weights.

Although there were previously relatively successful RL algorithms that employed neural networks as Q-function approximators, such as Neural Fitted Q-Learning (Riedmiller 2005 [2]), these methods were inferior to DRL algorithms performance-wise because they used the strategy of offline learning. The reason why offline learning is so detrimental to performance is because it implies that the weight updates are performed in batches that include the entire training information. On the other hand, most DRL solutions perform updates online, which means that stochastic gradient descent is applied over a small minibatch of training information. Therefore, online DRL methods are computationally less demanding and scale better to larger data sets. However, there are various challenges that arise in online training. Since the environment is usually represented as a huge state-action space, performing updates online may take a huge number of episodes to learn the optimal Q-function. This is due to the fact that the network quickly "forgets" what it learned before, because each experience has a small chance of being resampled for a minibatch update. Additionally, the agent receives information from the environment in a sequential manner which can lead to a set of highly correlated training data. Finally, unlike in classical supervaised learning scenarios, we encounter the "moving target" problem – the target variable in our loss function is also dependent on the weights of our model. These issues significantly impair training stability and can even cause a DRL model to diverge.

We now divert our attention to different ways to tackle the aforementioned issues. The first of three algorithms we will consider is called DQN.

A. Deep Q network and Model Architecture

A deep Q network (DQN) is a deep neural network that for a given input state *s* outputs a vector of action values Q(*s, ·, θ*), where *θ* are the trainable parameters of the network. For an n-dimensional state space and an action space containing m actions, the DQN is a function from R^n to R^m. At its foundation, DQN utilizes the idea of *online Q-learning* (Watkins and Dayan, 1992 [1], equation 1), and that is to use Bellman equations for calculations of our Q-values:

$$\begin{aligned} Q^*_{target} &= r + \gamma \cdot max_{a'} Q^*_i(s', a') \\ Q^*_{i+1}(s,a) &= Q^*_i(s,a) + \alpha \cdot (Q^*_{target} - Q^*_i(s,a)) \end{aligned} \quad 1$$

Online Q-learning. Note that a is the action performed in current state s that leads to the landing state s' with immediate reward r. Q is the tabular Q function, γ the discount factor, and α is the learning rate.*

In DQN, the tabular Q^* function from 1 is replaced with our parametrized *Q(s, ·, θ)* function. We calculate the weights *θ* by iteratively optimizing a specific loss function. Note that we chose Huber loss for training our Q model, however, now we will only consider quadratic loss *L* for the sake of simplicity:

$$L = \frac{1}{2} \cdot (Q_{prediction} - Q_{target}) \quad 2$$

$Q_{prediction}$ represents the estimated Q-value of the performed action *a* in our last state *s*. We obtain it by performing a single forward pass through our main Q network:

$$Q_{PREDICTION} = Q(S, A, \Theta) \quad 3$$

On the other hand, Q_{target} is calculated according to the Bellman equation. It is the sum of the immediate reward *r* received for performing action *a* in state *s* and the discounted maximum Q-value over all possible actions *a'* in the landing state *s'*:

$$Q_{target} = r + \gamma \cdot max_{a'} Q(s', a', \theta) \quad 4$$

The aforementioned loss function (2) is optimized by iteratively performing stochastic gradient descent with respect to weights θ. This ensures that the predicted Q-value for action *a* in state *s* (3) regresses towards its target value (4). Recall that this is an online method, which means that the updates are performed in a sequence of minibatches. This can raise issues regarding training stability that are later addressed by introducing *experience replay* and *target network* (Mnih et al. 2013, [3]).

Another important aspect is the exploration-exploitation tradeoff. In essence, every RL agent needs to make a right balance between the exploration of the environment and the exploitation of the learned behavior. Resorting to exploitation to quickly may cause the agent to learn a sub-optimal behavior, whereas excessive exploration may be computationally overwhelming. This tradeoff is later tackled using a *ε-greedy strategy* for action selection (section E).

Additionally, we also need to determine how to represent the state of the environment. Since the goal is to enable the agent to learn directly from pixels, it is obvious that we need to use an image as an input to the Q-function. However, one image does not contain the information about the movement of the entities represented in that image. Consequently, all three models that were trained use a stack of last four frames received from the environment as the state representation and the input to the neural network.

As far as the architecture of the deep neural network is concerned, it does not differ from the one used in Mnih et al. (2015, [6]). It consists of three convolutional layer with one hidden, fully-connected layer followed by an output layer that has a separate neuron for each valid action. The first convolutional layer uses 32 filters with kernel size 8 and stride 4. The second and the third layer use 64 filters each with kernel size 4 and 3 respectively, as well as strides of 2 and 1 respectively. All convolutional layers are followed by a rectified linear unit. The final hidden layer is fully-connected and consists of 512 rectifier units. Since there are 4 valid actions in each state of Atari Breakout, the output layer consists of 4 neurons.

B. *Frame preprocessing*

Recall from previous paragraph that the model that was trained used four stacked images as an input to the Q-function. Each of the images was expected to be of dimensions 84×84×1, where 1 indicates the grayscale channel. On the other hand, the Atari environment provides 210×160×3 pixel images, where 3 represents the standard RGB color channels. Consequently, an additional preprocessing step was required to convert the images returned from the environment to 84×84×1 format. The exact same preprocessing steps were used in Mnih et al. (2013 [3]) and were aimed at reducing the input dimensionality.

Algorithm 1: Main procedure of DQN

```
procedure DQN
  initialize main network with weights $\theta$
  initialize target network with weights $\theta'$
  initialize empty replay memory $D$
  $K \leftarrow$ maximum number of games
  $\phi \leftarrow$ frame processor
  for $i \leftarrow 0$ to $K$ do
    current frame $\leftarrow$ get initial frame from the environment
    current state $\leftarrow$ repeat $\phi$(current frame) 4 times
    while game has not finished do
      action $\leftarrow$ $\epsilon$-greedy action chooser(current state, length of D)
      next frame, reward, terminal $\leftarrow$ preform action(current state, action)
      Store (current frame, action, reward, new frame, terminal) in D
      Remove first frame from current state and add $\phi$(next frame)
      Sample random mini batch of transitions $(s_j, a_j, r_j, s_{j+1})$ from $D$
      if $s_{j+1}$ is terminal then
        $Q_{target} \leftarrow r_j$
      if $s_{j+1}$ is not terminal then
        $Q_{target} \leftarrow r_j + y * max_{a'} Q(s_{j+1}, a', \theta')$
      Perform a gradient descent step on $(Q(s_j, a_j, \theta) - Q_{target})^2$
      if it is time to update target network then
        $\theta' = \theta$
```

C. *Experience Replay*

One of the strongest efforts in recent years in RL were primarily concerned with designing techniques that would stabilize training. The most common scenario in the majority of supervised learning algorithms is that data samples are independent and identically distributed (IID). Unfortunately, we cannot make such an assumption in most RL applications. In RL, data samples usually come as correlated sequences, and these correlations represent one of the biggest obstacles to stable training. Breaking these correlations is a necessity for algorithm convergence.

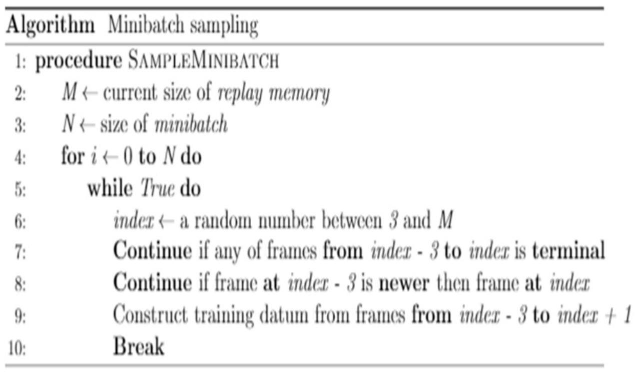
Another problem that occurs in RL is that the agent tends to forget its past experience. This is due to the fact that the environment keeps feeding the agent with experiences that are more common, whereas other, rarer experiences tend to be neglected, as the agent only gets an opportunity to see them a small number of times. This issue is especially present in games, where earlier stages of the game are visited to a greater extent. Therefore, it is important to design a strategy in which the agent could continuously learn over the entire set of experiences, not just the experiences that are most prevalent in the environment. Moreover, we also want to learn online (in minibatches), as it is computationally less demanding.

Experience replay represents one solution to this problem. Instead of performing gradient descent on every single action performed in the environment, we store all of the transitions in a data structure (*replay memory*) from which we, occasionally, randomly sample minibatches and perform weight updates. Each transition is one experience and is represented as a tuple e = (s_i, a, r, s_j, t), where s_i is the original state (a 84×84×1 frame), a is the action performed in s_i, r stands for the associated reward, s_j the landing state (again a 84×84×1 frame) and t is the terminal flag that signals the end of the game. With *experience replay* we tackle both of the aforementioned problems. Firstly, „the agent now gets a chance to refresh what it has learned before [4]", and secondly, „randomizing the samples breaks the correlations and therefore reduces the variance of the updates [3]." Furthermore, for the sake of sampling simplicity, the *replay memory* is implemented as a circular buffer of fixed size *M*. This means that older observations are eventually overwritten by new ones.

D. Sampling Training Data

Recall that the model is fed with a stack of four 84×84×1 frames. Therefore, if the size of the minibatch is *N*, for each training datum we need to sample four consecutive transitions from the *replay memory*. However, given the fact that the *replay memory* is a circular buffer, the process of sampling imposes one important, yet obvious restriction. In essence, we have to ensure that all four sampled transitions that are used for building one training datum come from the same instance of the game. This is achieved by inspecting the terminal flag *t* for each sampled transition, except for the last one. If any of the first three transitions is terminal, all four transitions are returned to the *replay memory* and the sampling process for this training datum is repeated. Furthermore, due to the circular nature of the *replay memory*, we also need to check that none of the first three frames that constitute our training datum are part of the newest transition in the *replay memory*.

ALGORITHM 2: MINIBATCH SAMPLING FROM REPLAY MEMORY.



```
Algorithm Minibatch sampling
1: procedure SAMPLEMINIBATCH
2:     M ← current size of replay memory
3:     N ← size of minibatch
4:     for i ← 0 to N do
5:         while True do
6:             index ← a random number between 3 and M
7:             Continue if any of frames from index - 3 to index is terminal
8:             Continue if frame at index - 3 is newer then frame at index
9:             Construct training datum from frames from index - 3 to index + 1
10:            Break
```

Note that my model was trained with a minibatch of size 32, whereas the maximum size of the replay memory was 1 million frames.

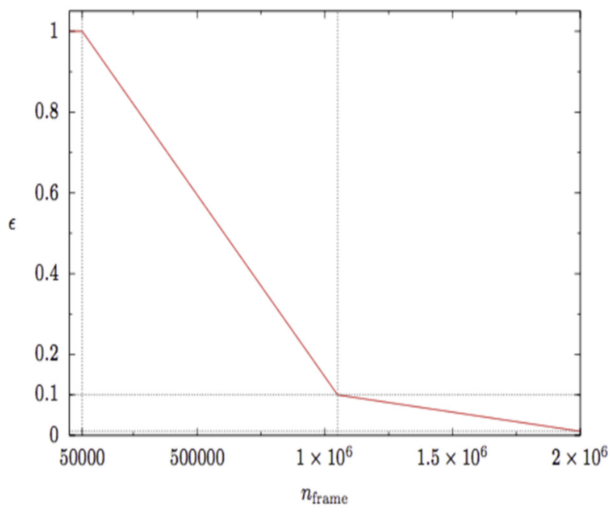
One might argue that the aforementioned Algorithm 2 might take an unpredictable amount of time, especially in situations where terminal transitions are more prevalent. In extreme cases, it may not even be possible to construct *N* different training samples with a larger frame stack size (12, 16 or even more). However, that does not seem to be the case for Atari Breakout. In Atari Breakout, an agent playing randomly over 1000 episodes acquires an average transition number of around 180 per episode. Given that our frame stack size is 4, and that we do not start training until we gather at least 50000 transitions, we can conclude that the overhead imposed in Algorithm 2 is insignificant. Moreover, as the agent learns during training, the episodes become longer and the size of the *replay memory* increases, which decreases the proportion of terminal transitions.

E. Action Selection

An inevitable part of every RL application is determining the right balance between exploration and exploitation. "Exploitation is the concept of repeatedly performing the same actions in the same situations because it results in the current maximum reward. On the other hand, exploration can be described as attempting to discover new features about the environment by performing sub-optimal actions [5]." Since the overall goal of a RL agent is to maximize its cumulative, expected reward, it is natural to think that the agent should only care about exploitation. However, resorting to full exploitation only makes sense if we are confident that we have explored the environment to a sufficient extent. Therefore, it is obvious that a trade-off between exploration and exploitation is present.

In DQN, exploitation is achieved by selecting an action whose corresponding Q value in the current state is maximized. Remember that the Q value for each action is obtained as an output of our model (the Q-function), whereas the state is represented as a stack of last four frames received from the environment. Exploration, on the other hand, we achieve by using the ε-greedy strategy for action selection. At each time step, we select the best possible action with the probability 1- ε and a random action with the probability ε. Note that the parameter ε is not a constant, but is subject to change as the agent receives more observations from the environment (Figure 1).

FIGURE 1: THE VALUE OF ε OVER TRAINING



Unlike in Mnih et al. (2015 [6]*), where ε is kept at 0.1, I decide to reduce ε to 0.01 in order to exploit more. This is because my model was trained for a smaller amount of time.*

The algorithm initially start with ε=1 and follow a full exploration policy until the number of observations reaches 50000. Afterwards, ε is decreased in order to benefit from learned weights of the model and put more emphasis on exploitation. After 2 million observations, ε remains a constant with value 0.01. The main benefit of ε-greedy strategy compared to other exploration methods lies in its computational simplicity. This is due to the fact that we do not require a forward pass through our network when choosing a random action.

F. Target Network

Previously in the paper, we already saw how we can achieve greater training stability by sampling transitions randomly from the *replay memory*. However, strong correlations between training samples are not the only problem that contribute to training instability. "Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q) function [6]." The problem that arises with the use of neural networks in RL is known as the "moving target" problem. The "moving target" problem refers to the fact that the target values are not fixed, but are actually calculated using the model that is currently being trained. Consequently, the target values are actually dependent on the weights θ of the model. The fact that these weights are constantly changed during training can give raise to great instability as we regress the predictions (3) towards the targets (4). Therefore, the goal is to introduce a "fixed target", and that can be achieved by using a second neural network (the target network) with a different set of weights θ' to calculate the target values. We update the target network by occasionally copying weights θ of our main network to weights θ' of our target network.

TABLE 2: THE LOSS FUNCTION FOR DQN AND DQN WITH TARGET NETWORK

DQN	$L = \frac{1}{2}(r + \gamma \cdot max_{a'}Q(s',a',\theta) - Q(s,a,\theta))^2$	5
DQN with target network	$L = \frac{1}{2}(r + \gamma \cdot max_{a'}Q(s',a',\theta') - Q(s,a,\theta))^2$	6

The changes to the loss function are shown in Table 2. Note that in the second case (6), the argument of the max operator is calculated using the weights θ' that correspond to the target network. Since gradient descent is performed with respect to the weights θ of the main network, the weights θ' of the target network do not change with every minibatch update. On the contrary, they remain fixed for a fixed number (*C*, which was 1000 in my model*)* of observations received from the environment, after which we copy θ to θ'.

G. Double DQN

The idea behind the introduction of the target network to the DQN algorithm was to use a different function for the estimation of the target Q-values. Although the target network is not trainable like the main network, it still yields Q-values that are noisy. Consider a situation in which the target network outputs all zeros for a given landing state s' (*Q (*s', a', θ'*)*). Since the network is noisy, some of the Q-values will be slightly above zero, while the others will be slightly below zero. However, since we are following a greedy policy for Q-value estimation (due to the max operator in the Bellman equation), our network will always favor those Q-values that have a positive noise. Consequently, we conclude that the current Q-value estimation is biased and over-optimistic. It was shown in Hasselt et. al. (2016, [7]) that this over-optimism impairs the performance of the model and leads to poorer policies. The suggested improvement was to split the responsibility of action selection and action evaluation inside the max operator to two different Q-functions. Since the algorithm already contains such two functions (main and target network), I decide to use the main network to select the "estimated" best action *a* in landing state *s'* and the target network to evaluate it (Table 3). This solves the problem of overestimated Q-values because the two networks have different noise and the bias towards slightly larger noisy Q-values cancels.

TABLE 3: THE TARGET CALCULATION FOR DQN WITH TARGET NETWORK AND DOUBLE DQN

DQN with target network	$Q_{target} = r + \gamma \cdot max_a Q(s',a,\theta')$	7
Double DQN	$Q_{target} = r + \gamma \cdot Q(s', argmax_a Q(s,a,\theta), \theta')$	8

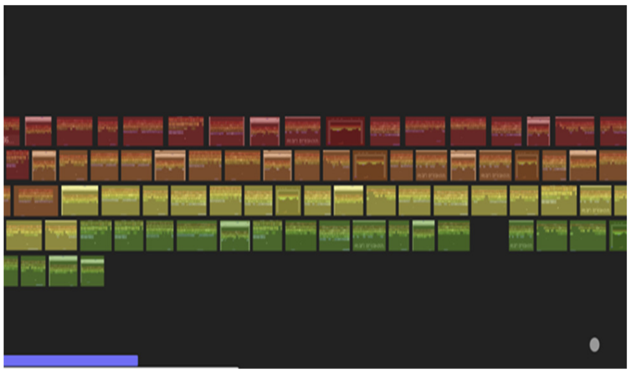
Double DQN (8) requires one additional forward pass per training step. This cost is greatly compensated by the fact that we learn better policies.

Note that the problem of Q-value overestimation arises from Bellman updates. Therefore, it is an issue present in *online Q-learning* as well as in DQN.

H. Dueling Architectures

The current model architecture used in DQN uses a final hidden layer of 512 neurons followed by an output layer with 4 separate neurons for each action. Such a neural network has a single stream whose only goal is to calculate the Q-value of each action for the given state. However, it becomes obvious that calculating the Q-value for each action of a given state is redundant if performing any action in that state does not make a significant impact on the environment. For instance, consider a game situation in Figure 2. It is obvious that the agent will inevitably lose in this game, so learning the Q-values of going right and staying in current place becomes irrelevant. It would be much more efficient only to learn the overall value *V(s)* of being in the given state *s*.

FIGURE 2: ATARI BREAKOUT GAMEPLAY



Learning a Q-value of each action in the given state is redundant as the agent will inevitably lose in this game.

Consequently, I improve the model by utilizing the idea of *dueling architectures* described in Wang et al. (2016, [8]). Instead of using a single stream to compute the Q values, the new model uses two streams – the first stream is responsible for computing the associated $V(s;\ \theta,\ \alpha)$ value for the given state s while the second stream computes the advantage value $A(s,\ a;\ \theta,\ \beta)$ of each action a in given state s. Note that $V(s;\ \theta,\ \alpha)$ is a measure of the overall value of being in state s while $A(s,\ a;\ \theta,\ \beta)$ is the advantage of taking action a at state s (how much better is to take this action versus all other possible actions at that state). The parameters α and β correspond to the weights of the *value* and *advantage* stream respectively. The changes to our neural network model in order to obtain the two streams are in the following paragraph.

Instead of the fully-connected final layer of 512 neurons, we use an additional convolutional layer of 1024 filters with kernel size 7 and stride 1. This layer is then split into two streams of equal dimensions (512 neurons each). The final step is to aggregate the two streams into the output layer that represents the Q-values. There are several ways to perform the aggregation step. The trained model uses equation 9 from Wang et al. (2016 [8]):

$$Q(s,a;\ \theta,\alpha,\beta) = V(s;\ \theta,\alpha) + \left(A(s,a;\ \theta,\beta) - \frac{1}{|A|} \cdot \sum A(s,a;\ \theta,\beta)\right) \quad 9$$

IV. EXPERIMENTS

Three different models were trained to play Atari Breakout: DQN, double DQN and dueling architectures. Due to the fact that I did not have access to a GPU, each model was trained for only 2 million frames received from the environment, with the maximum size of replay memory equal to 1 million transitions. Since I perform a training step after 4 actions, the total number of training steps performed on each model was (2000000 – 50000) / 4 = 487500. Note that this is a considerably smaller amount of training time then shown in Mnih et al. (2013, [3]), Hasselt et al. (2016, [7]) and Wang et al. (2016. [8]), however, it is still enough to notice the differences related to the performance of these models on Atari Breakout. The results are summarized in Table 4.

TABLE 4: RESULTS AFTER 2 MILLION FRAMES IN ATARI BREAKOUT

Algorithm	Average score over last 100 evaluation episodes	Best score
DQN with target network	27.93	45
Double DQN	29.76	49
Dueling architectures(with Double DQN)	15.11	32

Note that the results from the second column are obtained after 2 million training frames. They represent the average reward over last 100 evaluation games. The last column represents the overall best performance of the agent during both training and evaluation phase. All three algorithms used the same hyperparameters.

We can conclude from Table 4 that the dueling architectures perform considerably worse than single stream models on Atari Breakout environment. This performance gap was also reported in Wang et al. (2016, [8]) where using the dueling architectures on Atari Breakout decreased the performance by more than 2% compared to a variant of double DQN (in our case it was much more than 2%, but the magnitude of the gap is not comparable because of different training length). This is most likely because the Breakout environment consists of only 4 actions in each state (which is considerably less than in other Atari games), so we do not gain much by generalizing between these actions. However, by swapping a fully connected layer of 512 neurons with a convolutional layer of 1024 filters and by subsequently performing a split to two separate streams, the neural network used in dueling architecture consists of a greater number of trainable weights than the single stream models. Thus, the dueling architectures network may be more prone to overfitting. This could be one explanation of the performance gap.

V. CONCLUSION

This paper demonstrates how a deep neural network can be used as an end-to-end RL solution for teaching an agent to play Atari Breakout. The proposed solution manages to extract features from a high-dimensional environment by employing state-of-the-art computer vision techniques, without hardcoding any specifics about the game to the model. I also manage to adjust the Bellman equations used in *online Q-learning* so that they can be used for training a deep learning model. By introducing *experience replay* and *target network*, I manage to stabilize and accelerate training. As an additional enhancement, I also discuss double DQN in order to tackle over-optimistic Q-value estimations. Finally, I review an alternative, double stream neural network architecture that addresses the problem of redundant Q-value calculations and conclude that it is not suitable for environments that have a small number of valid actions in each space, such as Atari Breakout.

There are plenty of ways in which the three observed models could be improved. The first step would be to train them on a decent GPU for a greater number of training frames (such as 10 or 15 million) and afterwards perform the same evaluation metric as in this paper. If the same performance gap between single and double stream models persists, it would be interesting to see whether employing the dropout technique to the dueling architectures model tackles the potential occurrence of overfitting.

VI. References

[1] Watkins, C. J., & Dayan, P. (1992). Q-learning. Machine learning, 8(3-4), 279-292.

[2] Riedmiller, M. (2005, October). Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method. In European Conference on Machine Learning (pp. 317-328). Springer, Berlin, Heidelberg.

[3] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

[4] Lin, L. J. (1993). Reinforcement learning for robots using neural networks (No. CMU-CS-93-103). Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

[5] Louis, R., & Yu, D. (2019). A study of the exploration/exploitation trade-off in reinforcement learning: Applied to autonomous driving.

[6] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.

[7] Van Hasselt, H., Guez, A., & Silver, D. (2016, March). Deep reinforcement learning with double q-learning. In Thirtieth AAAI conference on artificial intelligence.

[8] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., & De Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. arXiv preprint arXiv:1511.06581.