EEE 448/548 - Project Report

Playing Atari with Deep Reinforcement Learning

Kutay Şenyiğit, Utku Kurtulmuş

Abstract

In this paper, the authors suggest that they successfully implemented the first deep learning model to learn control policies through high-dimensional sensory input using reinforcement learning. They claimed that they used a convolutional neural network, and while they were training, they used a variant of Q-learning. In the paper, it can be understood that raw pixels are used as input, while as output, they use a value function for foreseeing future rewards. All methods are applied to seven Atari 2600 games from the Arcade Learning Environment. They claim that made no changes to the architecture of the game or the learning algorithm. As a result of the study, they concluded that the resulting model of the study is better than the previous studies. Additionally, the human experts have been defeated by the model in three of the games.

1 Introduction

The authors claim that learning by high-dimensional inputs such as vision and speech is one of the difficult challenges in reinforcement learning. They suggest that the most successful reinforcement algorithms rely on hand-crafted features combined with linear value functions or policy representations. The paper suggests that recent advances in computer vision and speech recognition allow the extraction of high-level features from the raw sensory data. This is achieved by various neural network architectures that include convolutional networks, multilayer perceptrons, restricted Boltzmann machines, and recurrent neural networks and have exploited both supervised and unsupervised learning. The authors claim that these techniques can be used together with reinforcement learning. However, they emphasize that this choice comes with several challenges. They suggest that most successful deep learning models up to date require supervised learning, or in other words, large amounts of hand-labeled training data. On the other hand, reinforcement learning algorithms must be able to learn from a scalar reward. This reward is generally delayed, noisy, and sparse. The second issue is the dependence and distribution of the data samples. Authors suggest that most deep learning algorithms assume that data samples are independent of each other, that there is no strong correlation between them, and that there is a fixed underlying data distribution. Nonetheless, we encounter sequences of highly correlated states in reinforcement learning, and the data distribution changes while the algorithm learns new behaviors. The authors suggest that their published paper shows that a convolutional neural network can handle these problems.

2 Background

The authors describe the problem as follows: an agent interacts with its environment, which, in this case, is the Atari emulator. The agent is allowed to select from a set of game actions determined by the game itself. At each step, the agent makes an action, and then the Atari

emulator processes this action and changes its internal state and the game score. The agent can take the raw input pixels representing the current state and the score. The agent takes the current score as its reward corresponding to its action. The authors represent the states as follows: $s_t = x_1, a_1, x_2, a_2, \ldots, a_{t-1}, x_t$. where: x_t represents the current screen frame at time t and a_{t-1} represents the action taken at time t-1. The authors claim that the original problem transforms into a Markov Decision Process (MDP) by defining the states in this manner. We can show that the state representation satisfies the Markov property:

$$s_t = x_1, a_1, x_2, a_2, \ldots, a_{t-1}, x_t, \text{ and } s_{t-1} = x_1, a_1, x_2, a_2, \ldots, a_{t-2}, x_{t-1} \Rightarrow s_t = s_{t-1}, a_{t-1}, x_t.$$

The authors define their reward function as a sum of discounted rewards where the γ represents the discount factor:

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

$$R_t = \gamma^{t-t} r_t + \gamma^{t+1-t} r_{t+1} + \ldots + \gamma^{T-1-t} r_{T-1} + \gamma^{T-t} r_T$$

$$R_t = r_t + \gamma r_{t+1} + \ldots + \gamma^{T-1-t} r_{T-1} + \gamma^{T-t} r_T$$

This is a valid and commonly used approach since it emphasizes the importance of immediate rewards. It discounts the future rewards with the discount factor. The authors also use a Q function to find the optimal policy to maximize the expected cumulative discounted rewards.

$$Q^*(s, a) = \max_{\pi} \left(\mathbb{E}(R_t \,|\, s_t = s, a_t = a, \pi) \right)$$

The authors also show that the Q function used to define the optimal policy fits well into the well-known Bellman equation.

$$Q^*(s, a) = \mathbb{E}_{s' \sim \varepsilon} \left[r + \gamma \max_{a'} \left(Q^*(s', a') \right) \,|\, s, a \right]$$

We can show this equation holds using the previous definition of the Q function and the reward function R:

$$Q^*(s, a) = \max_{\pi} \left(\mathbb{E}(R_t \,|\, s_t = s, a_t = a, \pi) \right)$$

$$Q^*(s, a) = \max_{\pi} \left(\mathbb{E}(\sum_{t'=t}^{T} (\gamma^{t'-t} r_{t'}) \,|\, s_t = s, a_t = a, \pi) \right)$$

$$Q^*(s, a) = \max_{\pi} \left(\mathbb{E}(r_t + \gamma r_{t+1} + \ldots + \gamma^{T-1-t} r_{T-1} + \gamma^{T-t} r_T | \, s_t = s, a_t = a, \pi) \right)$$

$$Q^*(s, a) = \max_{\pi} \left(\mathbb{E}(r_t + \gamma \left(r_{t+1} + \ldots + \gamma^{T-2-t} r_{T-1} + \gamma^{T-1-t} r_T \right) | \, s_t = s, a_t = a, \pi) \right)$$

$$Q^*(s, a) = \max_{\pi} \left(\mathbb{E}(r_t + \gamma \left(\sum_{t'=t+1}^{T} (\gamma^{t'-t} r_{t'}) \right) | \, s_t = s, a_t = a, \pi) \right)$$

$$Q^*(s, a) = \max_{\pi} \left(\mathbb{E}(r_t + \gamma \left(R_{t+1} \right) | \, s_t = s, a_t = a, \pi) \right)$$

$$Q^*(s, a) = \mathbb{E}_{s' \sim \varepsilon} \left[r + \gamma \max_{a'} \left(Q^*(s', a') \right) \,|\, s, a \right]$$

As the authors suggest, the value of a Q function can be calculated using value iteration techniques, and it will converge to the optimal Q* as iteration counts go to infinity:

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'}\left(Q_i(s',a')\right) \mid s,a\right] \quad \text{and} \quad \lim_{i\to\infty} Q_i \to Q^*$$

However, the authors suggest that it is impractical to try to calculate the value of Q functions directly. Instead, the authors suggest that it is common to use approximation functions to estimate the value of Q functions. $Q(s,a;\theta) \approx Q^*(s,a)$ The authors suggest that, in general, this estimator function is chosen to be linear, but they claim that a non-linear neural network estimator can be used. They call this approximator Q-network with weights θ. Then, they define a mean squared error function to train this neural network:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q_i(s,a;\theta_i))^2\right] \quad \text{and} \quad y_i = \mathbb{E}_{s'\sim\varepsilon}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \mid s,a\right]$$

$\rho(s,a)$ represents the probability distribution of the sequence of states s and actions a, also known as the behavior distribution. Differentiating the loss function with respect to the weights, we arrive at the following equation:

$$\nabla_{\theta_i} L_i(\theta_i) = \nabla_{\theta_i}\left(\mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q_i(s,a;\theta_i))^2\right]\right)$$

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[\nabla_{\theta_i}(y_i - Q_i(s,a;\theta_i))^2\right]$$

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\varepsilon}\left[2\left(y_i - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\varepsilon}\left[2\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\varepsilon}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

The author claims that rather than computing the full expectations in the above gradient, using a stochastic gradient descent can help us reach our results faster. Using gradient descent and replacing expectations from single samples coming from the behavior distribution, we can arrive at the following Q-learning algorithm:

$$\nabla_{\theta_i} L_i(\theta_i) \approx \left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)$$

$$\theta_i \leftarrow \theta_i + \alpha \nabla_{\theta_i} L_i(\theta_i)$$

$$\theta_i \leftarrow \theta_i + \alpha\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)$$

3 Implementation

3.1 Related Works

The authors mention that the best success story of an AI playing a game is probably TD Gammon. That plays a backgammon game by using reinforcement learning. They mention that the program plays the game as superhuman. In this model, a model-free reinforcement learning algorithm that is similar to Q-learning is used. Therefore, it estimated the value function by using a multi-layer perceptron with one hidden layer. However, when we dive into the referenced paper "Temporal Difference Learning and TD-Gammon," we see that the best performance of TD-Gammon is achieved with the network that has 40 hidden units that were trained for a total of 200,000 games. This achieved a strong intermediate level of play that is equal to NeroGammon's. The paper mentions that the other applications of TD-gammon were less successful such as chess and go. This leads people to think that the better result was exceptional for backgammon. That is because perhaps the stochasticity in the dice rolls contributes to exploring the other states in the space state. This situation makes the value function smooth. However, chess and go are deterministic games, and there are not any chance factors. Maybe it is the reason that they were not successful. Moreover, the combination of non-linear function estimations along with the Q-learning may happen to diverge. Most of the study about reinforcement learning is made on linear function estimations since it is more guaranteed to be converged. The authors indicate that the use of reinforcement learning with the combination of deep learning to estimate the environment can be used to estimate nonlinear functions using gradient-temporal differences. This addresses the diverge problem in the estimation of non-linear function when evaluating a fixed policy. However, the paper suggests that these methods have not been extended to non-linear control.

3.2 Preprocessing

The authors suggest that working with 210x160 images with 128 color palettes can increase the computational complexity. The model probably won't need this color information, and a lower resolution might work. Therefore, RGB representation was converted to gray-scale representation, and the image was down-sampled to 110x84. After that, the authors claim that they cropped the playing area to be 84x84, and this was needed due to the GPU implementation of 2D convolutions expecting square inputs. Then, they took the last frames as an input to the neural network. Therefore, the input size is 84x84x4. The algorithm suggests that instead of giving the state action pairs to the neural network and getting one final output, giving only the state as the input and getting output Q values for each action pair is a better approach; by doing this, we avoid doing forward passes for each individual input. Their Deep-Q network architecture consists of a hidden layer that convolves 16 8x8 filters with stride 4 with the input image and applies a rectifier nonlinearity. The second hidden layer convolves 32 4x4 filters with stride 2, again followed by a rectifier nonlinearity. The final hidden layer is fully connected and consists of 256 rectifier units. The final output layer is a fully connected linear layer with output neurons for each valid action.

3.3 Algorithm

Their motivation is to take Tesauro's TD-Gammon architecture as a starting point. That was using on-policy samples of experience to evaluate the value function. The self-play mechanism is used to train the model. However, they implemented the technique of experience replay. This technique includes the following. Experiences of the agent at each time-step $e_t = (s_t, a_t, r_t, s_{t+1})$ are stored in a data set D such that $D = e_1, e_2, \ldots, e_N$. Each time, a random experience is drawn from the data set, and the Q-learning is updated. After that, the

agent selects and performs an action using an ϵ-greedy policy.

In games, agents usually need to take into account the past events to make decisions. For example, if we consider a moving ball, the model should select an action according to the ball's direction. In order to understand what the direction of the ball is, a series of past events are needed. Different sizes of experience may be problematic for the neural networks. This is because the neural networks expect inputs of the same size. However, the past events have not fixed size length. The function ϕ helps to transform the historical sequences into a consistent format that the network can get as an input. Therefore, this allows the network to learn the best actions according to the history of events. Moreover, it enables updating weights every time a new experience is added, allowing data efficiency for big data.

Algorithm 1 Deep Q-learning with Experience Replay

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

Interpretation of the Algorithm

The given algorithm outlines the training process for an agent in a reinforcement learning setting, particularly within the context of deep Q-learning. Initially, the agent initializes a replay memory of size N and the action-value function Q with random weights. During each round or episode of the game, the agent captures the current state by preprocessing the first screen frame using the function ϕ. Subsequently, for each step within the episode, the agent employs an ϵ-greedy strategy, balancing exploration and exploitation. It either selects a random action with probability ϵ or follows the greedy policy based on the Q-function's current understanding of the best action. After executing an action, the agent observes the emulator's response to determine the obtained reward, recording this experience in its replay memory. Periodically, the agent randomly samples experiences from the replay memory, using them to calculate the loss function, and updates the weights of the Q-function through stochastic gradient descent to improve its predictive accuracy. This process is iteratively repeated, allowing the agent to refine its strategy over time and enhance its overall performance in the given environment.

In the beginning, the function of the ϕ function within the Deep Q-Learning framework was difficult to understand for us. This is because the paper indicates that the ϕ function is a function for preprocessing observations. It transforms the inputs into a more suitable form for the neural network to process. However, the information about what ϕ does with the actions was not explicitly stated. Therefore, we are confused. Our first intuition was to think

that the ϕ function took both the state and action as inputs and transformed them to give a neural network as an input. This misunderstanding is happened because of the notation $s_{t+1} = s_t, a_t, x_{t+1}$ in Algorithm 1 and the state definition given in the background. That is, it seemed that actions are a part of the input of the function ϕ along with the raw pixels. However, we later realized that ϕ only preprocesses the observations (images from the game frames) and does not include the actions. We consensus that the authors could have explained the ϕ function more clearly. Their notation suggests that actions are included in the sequence that ϕ processes, which is misleading. However, the actions are, in fact, part of the experience tuple stored in the replay memory and are important for learning. Thus, they are not part of the preprocessed state the neural network uses.

4 Experiment and Results

They tried their model with the seven popular Atari Games. These games are Beam Rider, Breakout, Enduro, Pong, Q*bert, Seaquest, and Space Invaders. They used constant hyperparamers while training. We think they might prefer to make the hyperparameters constant because they got great results. However, maybe they can try other combinations of the parameters to check if they can suppress the result they got. However, they made changes to the reward structure of the games in the training phase. It is because games have different scoring scales, and they handle this by changing all positive rewards to 1 and all negative rewards to -1 while keeping the 0 rewards the same. They used the RMSPropt algorithm with a minibatch size of 32. They used a frame-skipping technique to speed up the learning process. The agent selected its action only on every kth frame, and this action is repeated on the skipped frames. They have chosen the k=4 in all games except Space Invaders. In that game, due to the period of the blinking laser beams, they have chosen k to be 3. They determined the behavior policy as ϵ-greedy, with ϵ being annealed linearly from 1 to 0.1 over the first million frames. Finally, they fixed the ϵ to 0.1. They have trained 10 million frames and they used replay memory of one million most recent frames. They mentioned that one epoch included 50000 weight updates with roughly 30 minutes of training time. They observed that the average reward of the agent collected was very noisy; on the other hand, the value of the Q function steadily increased.

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	-20.4	157	110	179
Sarsa [3]	996	5.2	129	-19	614	665	271
Contingency [4]	1743	6	159	-17	960	723	268
DQN	4092	168	470	20	1952	1705	581
Human	7456	31	368	-3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	1720
HNeat Pixel [8]	1332	4	91	-16	1325	800	1145
DQN Best	5184	225	661	21	4500	1740	1075

Table 1: Performance scores across different agents and games.

As a final step, they compared the performance of their algorithm with the best-performing RL algorithms and a real human. The algorithm seems to be outperforming other RL algorithms, but in general, it loses to a real human in 4/7 games. The model was more successful than humans at the games Breakout, Enduro, and Pong with the DQN algorithm. They used DQN with ϵ-greedy policy where ϵ = 0.05. The performance of each algorithm and the human can be compared in Table 1.