Reinforcement Learning with Neural Networks: Playing Atari Games using Deep Q-Networks

Cătălin-Aurelian CIOCÎRLAN

Intelligent Systems and Computer Vision
University "Politehnica" of Bucharest, Bucharest, Romania
catalin.ciocirlan@stud.etti.upb.ro

***Abstract*—In this study, we explore the application of Deep Q-Network (DQN) models to three classic Atari games: Ms. Pacman, Breakout, and Space Invaders. The objective is to train agents capable of achieving high scores in these games through reinforcement learning techniques. Our approach involves the implementation of five Fully Connected (FC) models for each game using PyTorch, alongside a Convolutional Neural Network (CNN) model utilizing the StableBaselines3 library. Through extensive experimentation and evaluation, we investigate the effectiveness of different neural network architectures in learning optimal strategies for gameplay. Results indicate significant variations in performance across the games and architectures, shedding light on the challenges and opportunities in applying DQN to diverse gaming environments. This work contributes to the exploration of reinforcement learning methods in video game domains, providing insights into the behavior of various neural network architectures and their performance in diverse gaming environments.**

***Index Terms*—RL, DQN, Atari, FC, CNN, PyTorch, StableBaselines3**

I. Introduction

The field of artificial intelligence has witnessed remarkable advancements in recent years, particularly in the domain of Reinforcement Learning (RL), which represents a machine learning paradigm where an agent learns to make sequences of decisions by interacting with an environment to achieve a certain goal. In RL, the agent learns through trial and error, receiving feedback from the environment in the form of rewards or penalties based on its actions. The goal of the agent is typically to maximize the total cumulative reward it receives over time.

Among the myriad applications of RL, training agents to play video games has emerged as a compelling testbed for evaluating learning algorithms due to the complex and dynamic nature of game environments. For example, Mnih *et al.* in [1] and [2] introduced the deep Q-network (DQN), a reinforcement learning algorithm that combines deep neural networks with Q-learning [3], a classic RL technique. DQN is notable for its ability to learn directly from high-dimensional sensory input, such as raw pixel data from video games, without requiring manual feature extraction. It employs a deep neural network to approximate the Q-function, which estimates the expected future reward for taking a particular action in a given state. By iteratively updating its Q-values through experience replay — a technique that randomly samples past experiences stored in a replay buffer — DQN improves its policy over time, enabling it to learn effective strategies for complex tasks. It has demonstrated remarkable success in mastering various Atari games and has since been extended and adapted for use in diverse RL applications.

Lillicrap *et al.* [4] extended on the work of Mnih and [5] in order to account for environments with continuous action spaces, in various gaming environments as well as in the field of robotic manipulation and locomotion. Their algorithm, Deep Deterministic Policy Gradient (DDPG), is an actor-critic reinforcement learning algorithm that combines elements of deep Q-learning and policy gradient methods to learn a deterministic policy directly from observations. DDPG maintains two neural networks — an actor network that learns the policy function, mapping states to actions, and a critic network that evaluates the quality of actions taken by the actor. By employing the actor-critic architecture and utilizing the deterministic policy gradient theorem, DDPG learns both the optimal policy and its corresponding action-value function, demonstrating its effectiveness in learning complex, high-dimensional policies.

Double Deep Q-Network (Double DQN) [6] is an enhancement of the Deep Q-Network (DQN) algorithm that addresses overestimation bias in Q-value estimation. In traditional DQN, the same set of parameters is used to both select and evaluate actions, leading to potentially overestimated Q-values. Double DQN mitigates this issue by decoupling action selection and evaluation, using one set of parameters to select actions and another to evaluate their Q-values. By doing so, Double DQN provides more accurate estimates of Q-values, resulting in improved policy learning and better performance on reinforcement learning tasks, especially in environments with large action spaces or high variance.

Another extension of DQN is the N-step DQN [7], that addresses the issue of sample efficiency and stability in training by incorporating multiple future rewards into the Q-learning update. In traditional DQN, updates are based on the immediate reward and the estimated future reward from the next state, leading to high variance and slow learning. N-step DQN mitigates this by considering rewards obtained over multiple time steps into the update, effectively bootstrapping the Q-value estimation over a longer horizon. By incorporating multiple rewards, typically spanning N time steps, the

algorithm achieves smoother updates and more accurate value estimates, leading to faster learning and improved performance in reinforcement learning tasks.

Dueling DQN [8] is another variation of DQN that enhances learning efficiency and stability by separating the estimation of state values from action advantages. Unlike traditional DQN, Dueling DQN divides the network into two streams: one for estimating state values and another for estimating action advantages. This separation enables more efficient learning by independently determining valuable states and advantageous actions, leading to improved performance in reinforcement learning tasks.

Besides gaming and robotics, reinforcement learning has also been successfully in the field of autonomous vehicles [9], [10], finance [11], [12], healthcare [13], [14], and more recently in the context of Large Language Models (LLMs) [15], [16], [17].

In this paper we delve into the application of reinforcement learning techniques, specifically a slightly modified Deep Q-Network algorithm, across three iconic Atari games: Ms. Pacman, Breakout, and Space Invaders. By leveraging various neural network architectures, including Fully Connected models implemented using PyTorch and a Convolutional Neural Network (CNN) model integrated with the StableBaselines3 library, the study aims to assess the effectiveness of these algorithms in mastering complex gaming environments.

This article is organized as follows: Section II provides a theoretical overview of Deep Q-Networks and their application to video games. Section III details the methodology employed in our study, including the design of DQN models and experimental setup. Section IV presents the results of our experiments and discusses key findings. Finally, Section V concludes the paper with a summary of contributions and avenues for future research.

II. Problem formulation

For the deep Q-network algorithm we consider tasks in which an agent interacts with an environment $\mathcal{E}$, in this case the Atari emulator, in a sequence of actions, observations and rewards. At each time-step the agent selects an action at from the set of legal game actions, $\mathcal{A} = \{1, \ldots, \mathcal{K}\}$. The action is passed to the emulator and modifies its internal state and the game score, after which it receives a reward r_t representing the change in game score, that may depend whole prior sequence of actions and observations; feedback about an action may only be received after many thousands of time-steps have elapsed.

Therefore, we operate on sequences of actions and observations, $s_t = x_1, a_1, x_2, \ldots, a_{t-1}, x_t$, and learn game strategies that depend upon these sequences. Assuming that all sequences terminate in a finite number of steps gives rise to a large Markov decision process, where every sequence represents a distinct state.

The goal of the agent is to interact with the emulator by selecting actions in a way that maximises future rewards. We make the standard assumption that future rewards are discounted by a factor of γ per time-step, and define the future discounted *return* at time t as:

$$R_t = \sum_{i=t}^{T} \gamma^{i-t} r_t, \tag{1}$$

where T is the final step of the game. We define the optimal action-value function $Q^*(s; a)$ as the maximum expected return achievable by following any strategy, after seeing some sequences and then taking some action a:

$$Q^*(s; a) = \max_{\pi} \mathbb{E}\big[R_t | s_t = s, a_t = a, \pi\big], \tag{2}$$

where π is a policy mapping sequences to actions (or distributions over actions).

The optimal action-value function obeys an important identity known as the Bellman equation. This is based on the following intuition: if the optimal value $Q^*(s', a')$ of the sequence $'$ at the next time-step was known for all possible actions a', then the optimal strategy is to select the action a' maximising the expected value of $r + \gamma Q^*(s', a')$.

$$Q^*(s; a) = \mathbb{E}_{s' \sim \mathcal{E}}\big[r + \gamma Q^*(s', a') | s, a\big] \tag{3}$$

In theory, one could estimate the action value function by using the Bellman equation as an iterative update, $Q_{i+1}(s; a) = \mathbb{E}\big[r + \gamma Q_i(s', a') | s, a\big]$, which would converge to $Q_i \to Q^*$ as $i \to \infty$. In practice, this approach is impractical. Instead, it is common to use a function approximator to estimate the action-value function, $Q(s, a; \theta) \approx Q^*(s, a)$, such as a neural network with the weights θ called Q-network. The network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

$$L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)}\big[(y_i - Q(s, a; \theta_i)\big], \tag{4}$$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}}\big[r + \gamma Q(s', a'; \theta_{i-1}) | s, a\big]$ is the target for iteration i and $\rho(s, a)$ is a probability distribution over sequences s and actions a named *behaviour distribution*. The parameters from the previous iteration θ_{i-1} are held fixed when optimising the loss function. The targets depend on the network weights, which is in contrast with the targets used for supervised learning, which are fixed before learning begins.

This algorithm is *model-free*, as it solves the reinforcement learning task directly using samples from the emulator $\mathcal{E}$, without explicitly constructing an estimate of $\mathcal{E}$. It is also *off-policy*: it learns about the greedy strategy $a = \max_a Q(s, a; \theta)$, while following a behaviour distribution that ensures adequate exploration of the state space. In practice, the behaviour distribution is often selected by an ϵ-greedy strategy that follows the greedy strategy with probability $1 - \epsilon$ and selects a random action with probability ϵ. This process is also called exploration vs. exploitation, as in the early stages of learning the agent is encouraged to explore the environment, taking primarily random action, in order to accumulate experience, which it will later use to refine its movements and exploit known strategies in order to maximize its immediate rewards.

III. Experimental Setup

A. Atari environments

In this study, we focus on the application of Deep Q-Network (DQN) models to classic Atari games, a popular benchmark suite for RL research introduced by Mnih et al. in 2013. Specifically, we target three iconic Atari games: Ms. Pacman, Breakout, and Space Invaders.

Ms. Pacman is a classic arcade game where players control Ms. Pacman, guiding her through mazes to eat pellets while avoiding ghosts. Eating power pellets lets Ms. Pacman turn the tables on ghosts briefly. The game features multiple levels with increasing difficulty. The action space of the game is discrete, having eight possible directions to move in (up, right, down, left, up right, up left, down write, down left) plus an extra *noop* action.

Breakout is another iconic arcade game where players control a paddle to bounce a ball and break a wall of bricks at the top of the screen. The goal is to clear all the bricks while preventing the ball from falling past the paddle. The action space is comprised of four possible moves: *noop*, fire, move left and move right.

Space Invaders tasks players with controlling a spaceship to shoot down rows of descending alien invaders. Dodging enemy fire and barriers, players must eliminate the aliens before they reach the bottom of the screen. The action space has 6 possible actions, namely *noop*, fire, left, right, left fire, right fire.

A screen capture of all the described games is presented in Figure 1. The games were integrated with our algorithm using the Gymnasium[1] library.

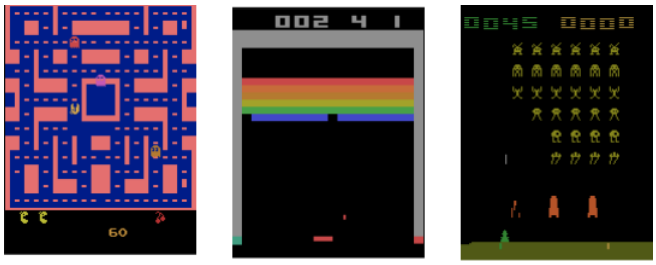

Fig. 1: Studied Atari Games. From left to right: Ms. Pacman, Breakout, Space Invaders

B. DQN

This study follows closely the method proposed by Mnih *et al.* 2015 [2], with a few notable differences.

In order to maximize the model's performance we utilize a technique known as experience replay, where we store the agent's experiences at each time-step, $e_t = (s_t, a_t, s_{t+1}, r_t)$ in a dataset $\mathcal{D} = \{e_1, \ldots, e_N\}$, pooled over many episodes into a replay memory. During the inner loop of the algorithm, we apply minibatch updates to samples of experience, $e \sim \mathcal{D}$, drawn at random from the pool of stored samples. This approach has several advantages over standard online Q-learning, where the agent learns from a set of sequential data:

- each step of experience is potentially used in many weight updates, which allows for greater data efficiency;
- learning directly from consecutive samples is inefficient, due to the strong correlations between the samples, randomizing the samples reduces the variance of the updates;
- when learning on-policy the current parameters determine the next data sample that the parameters are trained on; by using experience the learning is smoothed out and oscillations or divergence in the parameters are avoided.

Firstly, while the original authors used a total buffer size of 1M samples, we chose a total size of 10K samples for our dataset, to minimize the computational costs.

Secondly, for each game we employ two types of models:

- five fully connected (FC) models, using PyTorch[2], that take as input "RAM" observations from the Atari emulator, including information such as the positions of game objects, the player's score, and other relevant game state variables. These observations come in the form of 1D vectors of size 128;
- one convolutional neural network (CNN), using the StableBaselines3[3] library, where the inputs represent raw RGB images directly from the game environment, having a resolution of 210 x 160 px. Contrary to Mnih *et al.*, we did not apply any form of preprocessing to the images, whereas the original study downsampled the images to 84 x 84 px.

Table I presents the architecture of all the implemented models. Each layer is followed by ReLU activation. Additionally, we used an ϵ-greedy policy to obtain an action at every time-step, instead of every 4$^{\text{th}}$ step as presented in [2].

Lastly, inspired by [4], we chose to utilize a separate target network to compute the targets y_i instead of using the previous model weights. This network is updated at every time-step using a soft update, controlled by the hyperparameter τ, which was proven to greatly improve the stability of learning when $\tau \ll 1$. Algorithm 1 presents our entire learning algorithm.

For the five FC models training was realised for 1K games, with an additional training of the first model for 10K games. The CNN models were trained for 1M and 10M time-steps. Adam [18] was used as optimizer, with a learning rate of $1e{-}4$ and a batch size of 128. The loss function used to minimize the error was the Huber loss, given by the following equation:

$$\mathcal{L} = \frac{1}{|B|} \sum_{(s,a,s',r) \in B} \mathcal{L}(\delta), \tag{5}$$

where B represents a batch from the replay buffer and

$$\delta = Q(s,a) - (r + \gamma \max_{a'} Q(s',a')), \tag{6}$$

$$\mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \|\delta\| \leq 1, \\ \|\delta\| - \frac{1}{2} & \text{otherwise} \end{cases} \tag{7}$$

[1] https://gymnasium.farama.org/

[2] https://pytorch.org/

[3] urlhttps://stable-baselines3.readthedocs.io/en/master/index.html

This loss acts like the mean squared error when the error is small, but like the mean absolute error when the error is large, which makes it more robust to outliers when the estimates of Q are very noisy.

We set the discount factor γ was to 0.99 and the soft update factor $\tau = 0.005$. The ϵ-greedy policy starts with an exploration probability of 0.9 and exponentially decays to its final value of 0.05 over roughly the first 10% of the training process. The source code for this paper can be found on our Github repo[4].

Algorithm 1 DQN training

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function network Q with random weights θ
Initialize target network Q' with weights $\theta' \leftarrow \theta$
for episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t, otherwise select $a_t = \max_a Q^*(s_t, a; \theta)$
    Execute action a_t in emulator and observe reward r_t and next state x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$
    Store transition (s_t, a_t, s_{t+1}, r_t) in $\mathcal{D}$
    Sample random minibatch of transitions (s_j, a_j, s_{j+1}, r_j) from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{final } s_{j+1} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta) & \text{otherwise} \end{cases}$
    Perform an optimization step on $Q(s_j, a_j; \theta) - y_j$
    Update the target network $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$
  end for
end for

IV. Performance Evaluation

During training, for each model we logged the current episode reward, as well as the moving average of the scores from the past 100 games, in order to asses the quality and stability of the training process. Figure 2 depicts an example of training curves.

For evaluation, we validated our trained algorithms on separate test environments for 1000 episodes, for each model reporting the best and average scores. The results can be seen in Table II.

Random Play serves as a baseline, where actions are chosen randomly without any learning or strategy. Consequently, its performance is consistently poor across all games because it lacks any form of intelligent decision-making.

The performance of FC models can vary based on their architecture and training length. Models with more learnable parameters (e.g., FC2, FC4) tend to have higher capacity to represent complex relationships in the data, potentially leading to better performance.

[4]https://github.com/Cata400/reinforcement-learning/tree/main/Atari

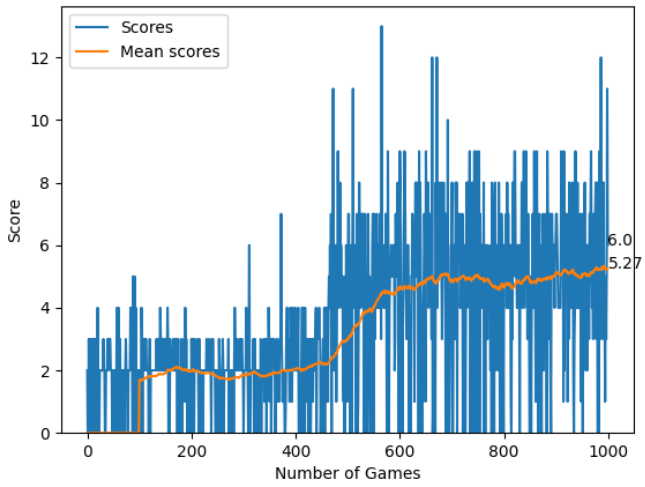

Fig. 2: Training curves for the first fully connected model, in the case of Breakout

For the CNN models, their hierarchical nature allows them to capture features at multiple levels of abstraction, enabling them to learn rich representations of the game states. However, in the case of Breakout and Space Invaders it seems that, at least in the early stages of training, the internal state of the emulator presents itself to be more relevant than raw pixel data for the agents to learn to adapt the environment, leading to higher scores on average. In the case of Ms. Pacman, our best results (from CNN-10M) come close to what Mnih *et al.* reported in [2].

Longer training durations (e.g., FC1-10K, CNN-10M) allow the models to explore the environment more thoroughly and refine their policies, resulting in improved performance compared to shorter training lengths. Regardless of input data or model architecture, we can conclude that DQN requires extensive training time in order to obtain high performing agents.

V. Conclusion

In conclusion, our study delved into the application of Deep Q-Network (DQN) models to classic Atari games, aiming to train agents capable of achieving high scores through reinforcement learning. We explored various neural network architectures, including Fully Connected models and Convolutional Neural Networks, and evaluated their performance across Ms. Pacman, Breakout, and Space Invaders. While longer training durations led to improved performance, we observed that the relevance of input data, especially in Breakout and Space Invaders, influenced the agents' ability to adapt the environment. Our study contributes to the exploration of reinforcement learning methods in video game domains, providing insights into the behavior of different neural network architectures and their performance across varied gaming environments. Ultimately, we highlight the necessity of extensive training time for DQN to yield high-performing agents, emphasizing the importance of future research in refining training methodologies and exploring novel architectures.

Model \Layers	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5
FC1	FC 64	FC 32	FC k	-	-
FC2	FC 256	FC 64	FC k	-	-
FC3	FC 128	FC 64	FC 32	FC k	-
FC4	FC 512	FC 128	FC 32	FC k	-
FC5	FC 512	FC 128	FC 64	FC 16	FC k
CNN	Conv 32x8x8, stride 4	Conv 64x4x4, stide 2	Conv 64x3x3, stride 1	FC 512	FC k

TABLE I: Architecture of all trained models. The size of the action space is denoted by k. Fully connected layers are expressed by the number of neurons they contain, while the convolutional layers by the shape and stride of the kernel

Agent	Score	Ms. Pacman	Breakout	Space Invaders
Random Play [2]	avg	307.3	1.7	148
FC1	best	2730	15	945
	avg (± std)	573.5 (± 323.5)	4.3 (± 2.3)	272.5 (± 133.1)
FC1-10K	best	2500	**24**	1070
	avg (± std)	772.9 (± 437.7)	**10.3 (± 3.7)**	**379.6 (± 175.1)**
FC2	best	4000	19	935
	avg (± std)	1267.5 (± 714.9)	3.8 (± 2.0)	278.7 (± 140.0)
FC3	best	3600	18	980
	avg (± std)	497.2 (± 353.0)	6.2 (± 2.6)	253.0 (± 126.7)
FC4	best	4180	21	**1150**
	avg (± std)	1401.0 (± 508.6)	6.7 (± 2.5)	357 (± 138.3)
FC5	best	2270	17	845
	avg (± std)	885.3 (± 270.5)	7.3 (± 2.4)	269.6 (± 114.1)
CNN	best	4100	10	660
	avg (± std)	1252.6 (± 514.7)	1.5 (± 1.4)	138.7 (± 110.2)
CNN-10M	best	**6620**	16	760
	avg (± std)	**2086.3 (± 739.4)**	5.5 (± 2.2)	188.0 (± 117.3)
Mnih *et al.* 2015 [2]	avg (± std)	2311 (± 525)	401.2 (± 26.9)	1976 (± 893)

TABLE II: Results for all the trained models

REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[4] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[5] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *International conference on machine learning*. Pmlr, 2014, pp. 387–395.

[6] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[7] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.

[8] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1995–2003.

[9] M. Kuderer, S. Gulati, and W. Burgard, "Learning driving styles for autonomous vehicles from demonstration," in *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2015, pp. 2641–2646.

[10] Y. Wu, S. Liao, X. Liu, Z. Li, and R. Lu, "Deep reinforcement learning on autonomous driving policy with auxiliary critic network," *IEEE transactions on neural networks and learning systems*, 2021.

[11] T. L. Meng and M. Khushi, "Reinforcement learning in financial markets," *Data*, vol. 4, no. 3, p. 110, 2019.

[12] B. Hambly, R. Xu, and H. Yang, "Recent advances in reinforcement learning in finance," *Mathematical Finance*, vol. 33, no. 3, pp. 437–503, 2023.

[13] S. H. Oh, S. J. Lee, and J. Park, "Effective data-driven precision medicine by cluster-applied deep reinforcement learning," *Knowledge-Based Systems*, vol. 256, p. 109877, 2022.

[14] K. Khezeli, S. Siegel, B. Shickel, T. Ozrazgat-Baslanti, A. Bihorac, and P. Rashidi, "Reinforcement learning for clinical applications," *Clinical Journal of the American Society of Nephrology*, pp. 10–2215, 2023.

[15] J. D. Chang, K. Brantley, R. Ramamurthy, D. Misra, and W. Sun, "Learning to generate better than your llm," *arXiv preprint arXiv:2306.11816*, 2023.

[16] D. Zhang, L. Chen, S. Zhang, H. Xu, Z. Zhao, and K. Yu, "Large language model is semi-parametric reinforcement learning agent," *arXiv preprint arXiv:2306.07929*, 2023.

[17] B. Hu, C. Zhao, P. Zhang, Z. Zhou, Y. Yang, Z. Xu, and B. Liu, "Enabling intelligent interactions between an agent and an llm: A reinforcement learning approach."

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.