SCIENTIA PLENA VOL. 1, NUM. 1 2025
www.scientiaplena.org.br doi: 10.14808/sci.plena.2025.XXXXXX

Space Invaders with Reinforcement Learning - DQN and CNN

B. F. Vieira[1]; D. S. F. Santos[2]; H. C. Júnior[3]

N. S. S. Júnior[4]; R. N. Andrade[5]

Departamento de Computação, Universidade Federal de Sergipe, 49107-230, Aracaju-SE, Brasil

hendrik@dcomp.ufs.br
(Recebido em 24 de fevereiro de 2025)

Abstract:
This study explored the application of deep reinforcement learning using Deep Q-Networks (DQN) and Convolutional Neural Networks (CNN) to train an agent in the game *Space Invaders* within the Arcade Learning Environment (ALE). The model was evaluated over 2,440 iterations, demonstrating a significant increase in the reward rate and the quality of the agent's decision-making. The results indicate that combining DQN with CNN enables a more accurate perception of the environment, leading to improved gameplay strategy. However, challenges such as training time and model convergence remain areas for improvement in future research.

Keywords: Reinforcement Learning, Deep Q-Network, Convolutional Neural Networks, Arcade Learning Environment.

Resumo:
Este estudo explorou a aplicação do aprendizado por reforço profundo, utilizando Deep Q-Networks (DQN) e Redes Neurais Convolucionais (CNN), para treinar um agente no jogo Space Invaders dentro do Arcade Learning Environment (ALE). O modelo foi avaliado ao longo de 2440 iterações, demonstrando um aumento significativo na taxa de recompensa e na qualidade das decisões tomadas pelo agente. Os resultados indicam que a combinação de DQN com CNN permite uma percepção mais precisa do ambiente, resultando em uma melhor estratégia de jogo. No entanto, desafios como o tempo de treinamento e a convergência do modelo são pontos que devem ser aprimorados em pesquisas futuras.

Palavras-chave: Aprendizado por Reforço, Deep Q-Network, Redes Neurais Convolucionais, Arcade Learning Environment.

1. INTRODUCTION

The development of independent agents based on data from high-dimensional sensors, such as speech and vision, has been a major challenge in the field of reinforcement learning. However, with the advancement of convolutional neural networks, which enable the efficient extraction of useful information from images, it has become possible to make significant progress in this area. The advancements in deep learning technologies have driven remarkable improvements in perception-related problems within artificial intelligence. When combined with an efficient implementation of reinforcement learning—which traditionally required manually labeled data—this progress allows for a more automated agent development and training process, reducing the need for human intervention.

The Arcade Learning Environment (ALE) is a widely used platform for testing reinforcement learning agents, both in terms of perception and policy selection. This tool includes a library with various Atari 2600 games and stands out for its practicality, as it runs within Python and facilitates reward definition, making it an ideal environment for agent training.

In this study, we developed a reinforcement learning model that utilizes Deep Q-Networks (DQN) for policy selection and convolutional networks for agent perception. The developed agent will be applied to the game *Space Invaders* within ALE, allowing for a detailed analysis of the training process and evaluation of results.

This paper is structured as follows: Section 2 describes the materials and methods used; Section 3 presents the results and discussions; Section 4 provides the conclusions; and Section 5 includes the references.

2. MATERIAL E MÉTODOS

This study was conducted using a computer with a CUDA-compatible GPU to accelerate training. The software employed included Python 3.11 and various specialized libraries, such as PyTorch, Gymnasium for environment simulation, Torchvision, NumPy, Scikit-Image, TorchSummary, and ALE-Py for interaction with the Arcade Learning Environment.

The experiment consisted of training a deep reinforcement learning agent using a convolutional neural network based on the Deep Q-Network (DQN) algorithm. Initially, the game environment frames were converted to grayscale and resized to a standardized dimension, in addition to applying the frame stacking technique to provide temporal context to the model.

The neural network was designed with a convolutional architecture composed of three convolutional layers followed by fully connected layers, using the ReLU activation function. The model training involved the DQN algorithm to learn the optimal actions for the agent, implementing the Experience Replay mechanism with a transition buffer to enhance training stability. An epsilon-decay exploration rate was employed to balance exploration and exploitation, along with the Adam optimizer with an adjustable learning rate.

The model evaluation was conducted through periodic tests to assess the agent's win rate and score. The experiments were performed in Gymnasium environments, including tests in different Atari games to evaluate the model's generalization capability.

2.1 Main Functions

Below are some of the main functions applied in the model:

```python
class DQN(nn.Module):
    def __init__(self, input_shape, num_actions):
        super(DQN, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )
        self.fc = nn.Sequential(
            nn.Linear(self.feature_size(input_shape), 512),
            nn.ReLU(),
            nn.Linear(512, num_actions)
        )

    def feature_size(self, input_shape):
        return self.conv(torch.zeros(1, *input_shape)).view(1, -1).size(1)

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)
        return self.fc(x)
```

Source: Own work

The code defines a deep neural network called DQN (Deep Q-Network), used for reinforcement learning in environments that utilize visual representations, such as image-based games.

The **DQN class** inherits from nn.Module and consists of two main layers:

1. **Convolutional layer (self.conv):**
    a. Three convolutional layers process the image input, extracting relevant features.
    b. Each layer is followed by a ReLU activation function, which introduces non-linearity to improve the network's learning capacity.
2. **Fully connected layer (self.fc)**:
    a. Transforms the features extracted by the convolutions into vector representations.
    b. Uses a dense layer with 512 neurons and ReLU activation.
    c. The final layer has a number of neurons equal to the number of possible actions (num_actions), producing the Q-values for each action.

The feature_size() method automatically calculates the output size of the convolutional layer, ensuring compatibility with the transition to the fully connected layer.

The forward(x) method performs the forward pass of the data through the network, first extracting features with the convolutional layers and then classifying the possible actions with the fully connected layers.

.

```python
class DQNAgent:
    def __init__(self, state_shape, num_actions, learning_rate=0.00025, gamma=0.99, epsilon=1.0, epsilon_min=0.1, epsilon_d
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.model = DQN(state_shape, num_actions).to(self.device)
        self.target_model = DQN(state_shape, num_actions).to(self.device)
        self.target_model.load_state_dict(self.model.state_dict())  # Inicializa o target model com os mesmos pesos
        self.optimizer = optim.Adam(self.model.parameters(), lr=learning_rate)
        self.memory = ReplayBuffer(memory_size)
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay
        self.batch_size = batch_size
        self.update_target_freq = update_target_freq
        self.train_step = 0
        self.num_actions = num_actions
        self.loss_fn = nn.MSELoss() # Salva o Loss
        self.episode_q_values = []  # Para a média dos Q-values


    def choose_action(self, state):
        if random.random() < self.epsilon:
            return random.randrange(self.num_actions)
        else:
            state = torch.FloatTensor(state).unsqueeze(0).to(self.device)
            q_values = self.model(state)
            self.episode_q_values.append(q_values.mean().item())  #  Salva o Q-value
            return q_values.argmax().item()

    def learn(self):
        if len(self.memory) < self.batch_size:
            return None, None

        states, actions, rewards, next_states, dones = self.memory.sample(self.batch_size)
        states = states.to(self.device)
        actions = actions.unsqueeze(1).to(self.device)
        rewards = rewards.to(self.device)
        next_states = next_states.to(self.device)
        dones = dones.to(self.device)

        q_values = self.model(states).gather(1, actions)
        next_q_values = self.target_model(next_states).max(1)[0].unsqueeze(1)
        target_q_values = rewards.unsqueeze(1) + self.gamma * next_q_values * (1 - dones.unsqueeze(1))

        loss = self.loss_fn(q_values, target_q_values)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        self.train_step += 1
        if self.train_step % self.update_target_freq == 0:
            self.target_model.load_state_dict(self.model.state_dict())

        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

        return loss.item(), q_values.mean().item()

    def save(self, filepath):
        torch.save(self.model.state_dict(), filepath)
```

Source: Own work

The DQNAgent class represents a reinforcement learning agent based on Deep Q-Networks (DQN). This agent is designed to interact with an environment, learn optimal policies, and improve its decision-making over time.

2.2 Main Components

1. **Initialization (__init__)**
    a. Creates two DQN models: one for learning (self.model) and another for predicting future values (self.target_model), which is periodically updated.
    b. Uses ReplayBuffer to store past experiences and enable efficient sampling.
    c. Defines hyperparameters such as learning rate (learning_rate), discount factor (gamma), exploration rate (epsilon and its decay), and target network update frequency (update_target_freq).
    d. Uses nn.MSELoss() to compute the difference between predicted Q-values and target values.

2. **Action Selection (choose_action)**
    a. Implements an **ε**-greedy strategy to balance exploration and exploitation:
        i. With probability epsilon, selects a random action.
        ii. Otherwise, selects the action with the highest predicted Q-value from the model.
    b. Stores the average Q-values for learning monitoring.
3. **Learning (learn)**
    a. Retrieves a batch of experiences from the ReplayBuffer**.**
    b. Computes the current Q-values and target Q-values using the target network.
    c. Minimizes the difference between predicted and target Q-values using MSELoss.
    d. Updates network weights using the Adam optimizer.
    e. Periodically copies the weights from the main network to the target network (update_target_freq).
    f. Gradually reduces epsilon to decrease exploration over time.
4. **Model Saving (save)**
    a. Allows saving the model weights for future use or continued training.

This agent is well-suited for reinforcement learning tasks, such as games and robotic control, where the agent must learn efficient strategies through trial and error.

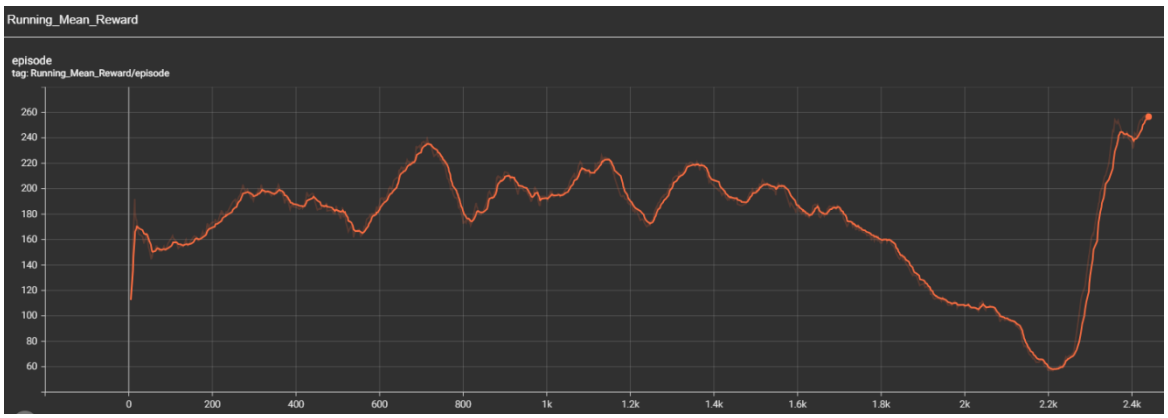
3. RESULTS AND DISCUSSION

This section presents the results obtained during the experiments conducted with the reinforcement learning model based on DQN and CNN. The tests were performed using the game *Space Invaders* in the Arcade Learning Environment (ALE). The model was trained over 2,440 iterations.

3.1 Graphical Visualization

After completing the training, several graphs were generated to provide a better analysis of the results. Below are some examples:

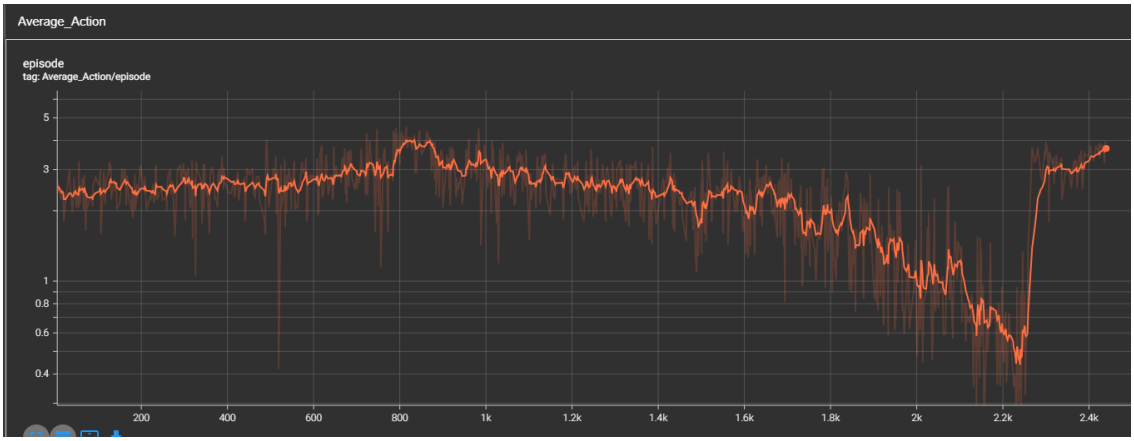
3.1.1 Average reward per episode:



Source: Own work

The results indicate a progressive improvement in the agent's performance throughout training. Initially, the average reward per episode was approximately 160 points, while by the end of the training, this average increased to 260 points. This growth highlights the agent's effective learning of the optimal policy.

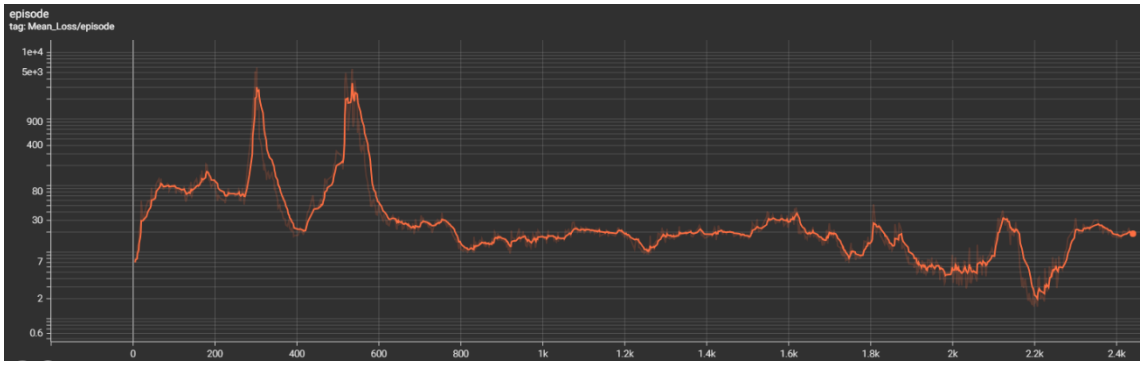
3.1.2 Average Actions per Episode



Source: Own work

From the graph, it is evident that as training progressed, the agent began to execute more actions per game session. This fact alone does not necessarily indicate the agent's quality. However, at the beginning, the agent achieved an average of fewer than 50 points per game session. By the end of the training, after 2,440 iterations, the average score increased significantly. This demonstrates that, in addition to taking more actions, the agent aims to maximize the score obtained during the game by selecting the best possible actions.

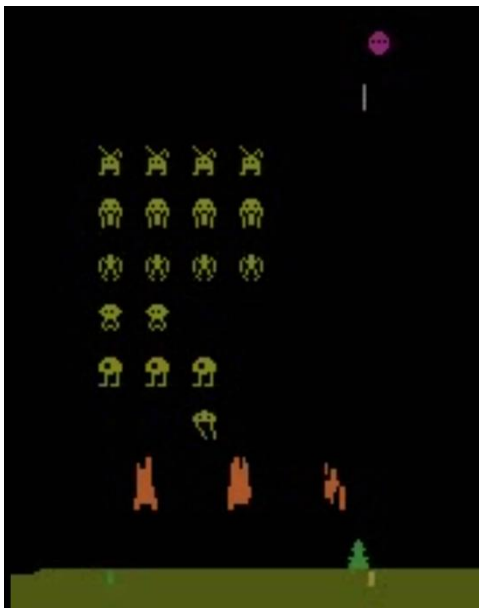
3.1.3 Average Loss per Episode



Source: Own work

The image above shows a decrease in the **loss** parameter, which measures how much the network's predictions deviate from the actual values. This indicates that the agent is learning to perform the task more effectively at each stage of the training process.

Image 1: Model 2,440 Episodes



Source: Own work

Image 2: Model - 2,440 Episodes

Source: Own work

At the end of the model's training, after a total of 2,440 episodes, it became evident that the trained model developed a tendency to remain on the right side of the screen, consistently targeting the farthest enemy spaceship due to the high reward it provided. As a result, most episodes executed with this trained network followed this pattern, which, despite occasionally leading to the loss of lives, was still beneficial in terms of the overall score achieved.

In this context, it is possible to observe overfitting in the model, as it adopted a predictable and specific behavior that maximized rewards rather than learning general gameplay patterns, such as dodging enemy shots and surviving longer. In future work, a possible approach to mitigate this issue would be to implement dropout on certain neurons to reduce overly specific learning and promote a more generalized model. Additionally, increasing the amount of training data could further enhance generalization.

3.2 Discussion

Ultimately, it is clear that the hybrid approach using DQN and CNN yields promising results within the presented tests. However, the relatively low number of training iterations negatively impacts the agent’s performance. During simulations with the most trained model, it became evident that the agent had not yet reached its full potential, meaning it remains far from the state-of-the-art in this field.

4. CONCLUSION

The experiments conducted in this study demonstrated that integrating Deep Q-Networks (DQN) with Convolutional Neural Networks (CNN) significantly enhances the performance of reinforcement learning agents in high-dimensional visual environments, such as *Space Invaders*. The agent successfully improved its reward rate throughout the iterations, highlighting the positive impact of the chosen approach.

Despite these advancements, several challenges remain, such as the need for longer training times to achieve performance closer to state-of-the-art levels and the potential exploration of more

sophisticated architectures like Dueling DQN and Double DQN. For future work, we suggest implementing distributed learning techniques and testing the model in different environments and games to assess its generalization capabilities.

Thus, this study contributes to the advancement of deep reinforcement learning, providing valuable insights into the combination of DQN with CNN and highlighting paths for future improvements and applications.

5. REFERENCES

1. Mnih, Volodymyr. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).
2. Guo, Xiaoxiao, et al. "Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning." Advances in neural information processing systems 27 (2014).
3. Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533.