# Blackjack-GPT: An Actor-Critic Reinforcement Learning via LLM System

Bryan Ambriz, Sri Vinay Appari, Suresh Ravuri
Computer Engineering Department
San José State University (SJSU)
San José, CA, USA
Email: {bryan.ambriz, srivinay.appari, suresh.ravuri}@sjsu.edu

**Abstract**—The goal of this project is to extend traditional reinforcement learning agents to utilize deep reinforcement learning by enhancing the agent with a neural network model using the Gymnasium library in the Python programming language. Traditional preference algorithms such as Monte Carlo, Temporal Difference, SARSA or Q-Learning learn the optimal preference strategy without requiring a neural network. Deep reinforcement learning strategies involving a neural network agent exist that can learn to predict the optimal state-value and action-value functions used within the Q preference learning algorithm. For this purpose we will use the reinforcement learning library Gymnasium, the neural network library Pytorch, and the deep reinforcement learning library agileRL in Python within the Google colaboratory computational environment. Thus, we aim to train a deep neural network to learn the optimal reinforcement learning strategy within Gymnasium's Blackjack or complex Atari game environments.

**Index Terms**—Actor, Critic, Reinforcement, Learning, Network, Large-Language-Model, LLM, Blackjack, GPT

## 1 Introduction

Reinforcement learning's foundational principles via classical conditioning emerged from behavioral psychology research in the early 20th century, particularly through the work of researchers like Thorndike (1911) and Skinner (1938). Classical conditioning was first systematically studied by Ivan Pavlov in his experiments with dogs (1927), where he discovered that dogs would salivate not only at the sight of food but also at the sound of a bell that had been repeatedly paired with food presentation. While classical conditioning, demonstrated by Pavlov's experiments, showed how organisms learn associations between stimuli, operant conditioning—more closely related to modern reinforcement learning—demonstrated how behaviors are modified through consequences. In reinforcement learning, an agent learns optimal behavior through interactions with an environment, receiving rewards or penalties that shape its decision-making process.

The mathematical framework often used to formalize reinforcement learning is the Markov Decision Process (MDP), a term first introduced by Richard Bellman in 1957 when he built upon Andrey Markov's earlier work on state transition probabilities to develop a framework for sequential decision-making under uncertainty. Traditional reinforcement learning algorithms such as the Monte Carlo, Temporal Difference, SARSA,

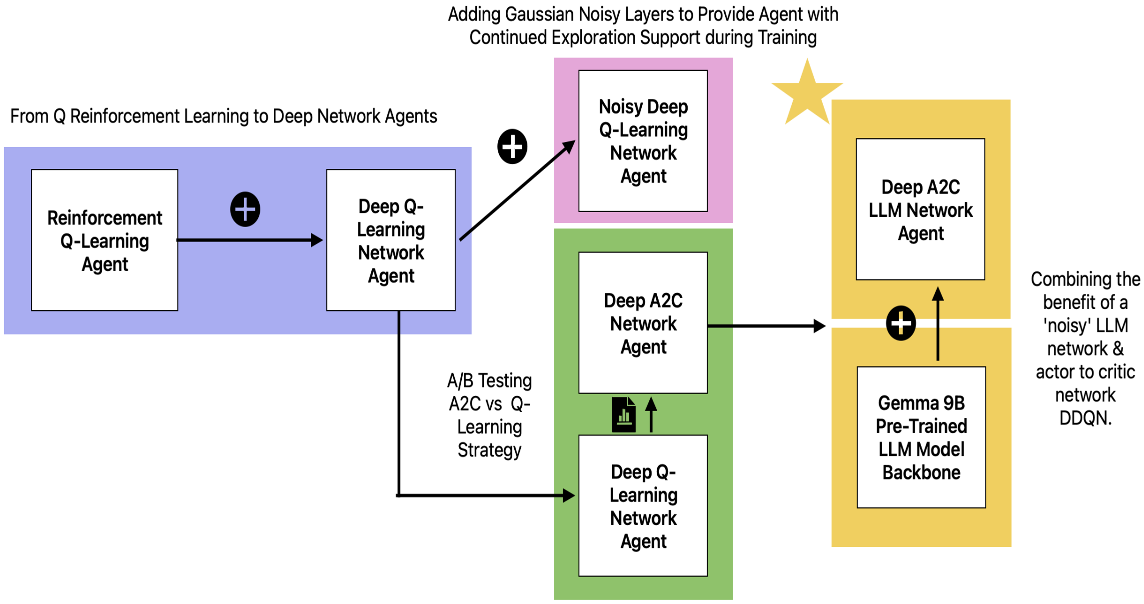



Fig. 1. Evolution of the blackjack learning network agent development from left-to right: Q-learning to double deep q learning (DDQN) in blue, experimental noisy DDQN in pink, A/B testing of actor-to-critic (A2C) DDQN and previous DDQN in green, and finally the combined LLM based A2C agent in yellow.

and Q-Learning algorithms make use of the mathematical definition of an MDP. These algorithms allow an agent in an environment to learn a decision following strategy over time by applying linear state-value and action-value or Q-Learning functions to estimate the goodness of a state or action.

## 2 Problem Statement / Project Architecture

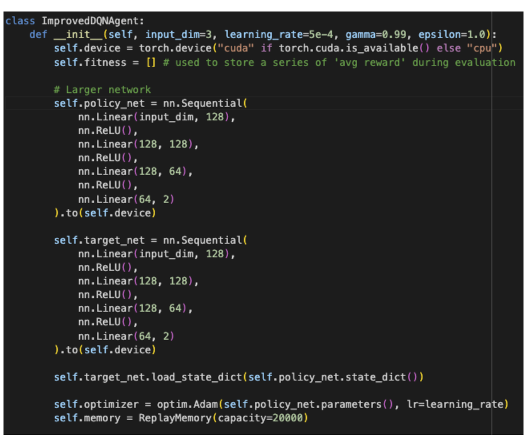
Traditional reinforcement algorithms and agents are inefficient and lack the ability to pick up complex patterns from an MDP environment, such as in Gymnasium's Blackjack and Atari environments. These learning algorithms (i.e SARSA or the Q-Learning strategies) use a strategy that utilizes state and action value expectation functions to influence the Q-learning algorithm. A limitation of these approaches is that these traditional algorithms use simple linear functions to modify the Q strategy matrix.

In deep reinforcement learning, neural networks emulate the strategy for the agent within the MDP framework to approximate the value functions via non-linear regression, allowing the agent to learn and make decisions in complex state-action mappings or high-dimensional state spaces. The Gymnasium and the agileRL libraries in Python are popularly used in both reinforcement learning and deep reinforcement learning. Gymnasium provides a variety of simple and complex action and state spaces or environments that can be used to contextualize an MDP, while agileRL provides a variety of neural network architectures and training strategies for training the deep reinforcement learning agent. In Gymnasium's CartPole environment using the Q-Learning strategy, for example, the actions are discrete while the states are continuous and require discretization and binning to be able to map the states to actions in the Q-Learning matrix.

Deep reinforcement learning provides more flexible, accurate, and efficient learning within Gymnasium's more complex environments, or within custom environments. Traditional reinforcement learning algorithms that require converting the continuous states or actions into discrete values will inherently lose valuable information in the process. Deep learning via neural networks solves this problem as it is able to understand the more complex, continuous input. Our project aims to solve the learning problems posed in Gymnasium's Blackjack and potentially more complex Atari environments to leverage the benefit of deep reinforcement learning.

## 3 Method(s) / System Design

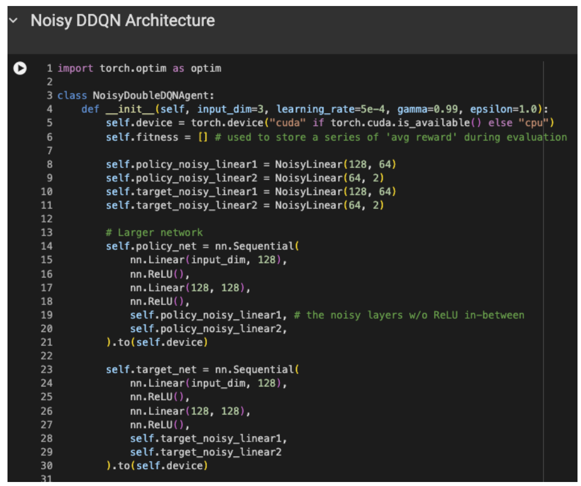
The first step of our architectural building process was to setup a base reinforcement learning algorithm that utilizes a neural network. For this purpose, we built a Deep Q-Learning Network (DQN) agent. A deep Q-Learning network is similar to a basic neural network and has traditional elements such as multiple dense layers, an activation function, an optimizer, etc. A key difference is that the DQN we implemented adds specific elements related to the reinforcement learning literature; it implements an off policy learning strategy. As such, it contains two networks within it - one that is used to learn the target Q-Learning strategy during training, while the other is used to enact the policy during prediction or inference.



```python
class ImprovedDQNAgent:
    def __init__(self, input_dim=3, learning_rate=5e-4, gamma=0.99, epsilon=1.0):
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.fitness = [] # used to store a series of 'avg reward' during evaluation

        # Larger network
        self.policy_net = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 2)
        ).to(self.device)

        self.target_net = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 2)
        ).to(self.device)

        self.target_net.load_state_dict(self.policy_net.state_dict())

        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=learning_rate)
        self.memory = ReplayMemory(capacity=20000)
```

Fig. 2. Example of the Deep Q-Learning Network Agent.

An additional exercise we managed to include was to add noise layers to add noise to the output predictions of the target policy networks, based on a learned distribution of gaussian noise. Including these layers has the potential to increase the "exploration" capability of a reinforcement learning model. By including noise to add randomization to the models outputs, the model is able to make "exploratory" decisions during training and after the typical exploration phase (when the epsilon value is close to 0) as well as post training. The next and culminating step of our project was to use a different learning algorithm, and compare their performance at learning the blackjack environment in Gymnasium. As previously mentioned, the DQN implemented a Q-Learning strategy that is an off-policy strategy to ensure the agent is taking reasonable risks during training that would allow them to explore the environment without strictly adhering to the policy. The A2C learning strategy, on the other hand, is an on-policy & risk-averse learning strategy that ensures the model follows it's policy during training & during prediction or inference. We built and trained a deep A2C network agent and then compared it to the DQN, and found promising results.



Noisy DDQN Architecture

```python
import torch.optim as optim

class NoisyDoubleDQNAgent:
    def __init__(self, input_dim=3, learning_rate=5e-4, gamma=0.99, epsilon=1.0):
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.fitness = [] # used to store a series of 'avg reward' during evaluation

        self.policy_noisy_linear1 = NoisyLinear(128, 64)
        self.policy_noisy_linear2 = NoisyLinear(64, 2)
        self.target_noisy_linear1 = NoisyLinear(128, 64)
        self.target_noisy_linear2 = NoisyLinear(64, 2)

        # Larger network
        self.policy_net = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            self.policy_noisy_linear1, # the noisy layers w/o ReLU in-between
            self.policy_noisy_linear2,
        ).to(self.device)

        self.target_net = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            self.target_noisy_linear1,
            self.target_noisy_linear2
        ).to(self.device)
```

Fig. 3. Example of the Noisy Deep Q-Learning Network Agent.

Actor-Critic Network Architecture

```python
class ActorCritic(nn.Module):
    def __init__(self, input_dim=3):
        super(ActorCritic, self).__init__()

        # Shared features extractor
        self.features = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU()
        )

        # Actor head (policy network)
        self.actor = nn.Sequential(
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 2),  # 2 actions: hit or stand
            nn.Softmax(dim=-1)  # Output probabilities for each action
        )

        # Critic head (value network)
        self.critic = nn.Sequential(
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 1)  # Single value output
        )

    def forward(self, x):
        features = self.features(x)
        action_probs = self.actor(features)
        state_value = self.critic(features)
        return action_probs, state_value

# Initialize the actor-critic network
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
ac_model = ActorCritic().to(device)
optimizer = optim.Adam(ac_model.parameters(), lr=1e-3)
```

Fig. 4. Example of the Deep Actor-to-Critic Learning Network Agent.

## 4 Results

Although we did not evaluate the noisy DQN as compared to the DQN, we noticed the noisy DQN performed slightly worse than the original DQN in terms of the training loss. This may be expected however, as the noisy DQN is including noise into the learned strategy and output predictions. In essence, it seems that the noisy layers may be used when training the model for more iterations where it might be necessary to extend the length of the exploration phase to avoid overfitting. As we did not train the agents for many iterations, we may not have had enough time to take full advantage of the noise layers.

We conducted an ablation experiment by evaluating the output of our previously trained blackjack playing DQN as compared to the trained A2C model. We found that the A2C model outperformed the DQN model at blackjack. After 200 simulated rounds of the game, the DQN model only won 18% of the games, whereas the A2C model won 35%. Similarly, the A2C outperformed the DQN by achieving a higher average reward across the 200 rounds. The A2C model averaged a reward of -0.250 while the DQN averaged a reward of -0.591. Finally, we compared them by measuring whether they agreed or differed in choices during their matches. We found that they had a decision disagreement rate of around 48% - implying that the models learned a very different strategy due to their difference in learning algorithm.

## 5 Conclusion

## References