# Blackjack

## Problem Description:

The objective of this reinforcement learning project is to create an agent that can play Blackjack, a popular card game where the goal is to have a hand value of 21 or as close to 21 as possible without going over. The agent will learn how to play the game by receiving rewards for each action it takes and adjusting its strategy accordingly.

The project will use a dataset from Kaggle, which contains information about the game of Blackjack. The dataset includes information about the cards in the game, the values assigned to each card, and the rules of the game. It also includes sample games that can be used to train the agent.

### Dataset Link

The dataset for this project can be found on Kaggle: https://www.kaggle.com/semanticalien/blackjack-rl

### Dataset Description

The dataset consists of 100,000 games of Blackjack, each with a different set of starting hands for the player and the dealer. For each game, the dataset includes the following information:

- The player's starting hand
- The dealer's starting hand
- The player's final hand
- The dealer's final hand
- Whether the player won, lost, or tied the game
- The reward the player received for the game

The dataset also includes information about the rules of Blackjack, such as the values assigned to each card and the conditions for winning or losing the game.

## Background Information

Reinforcement learning is a type of machine learning where an agent learns to make decisions by receiving rewards for its actions. In the case of Blackjack, the agent will learn to make decisions about when to hit, stand, or double down based on the cards it has been dealt and the dealer's up card. The goal of the agent is to maximize its total reward over a series of games.

To create an agent that can play Blackjack, we will need to use reinforcement learning algorithms such as Q-learning or SARSA. These algorithms will allow the agent to learn from its experiences and improve its strategy over time.

## Deliverables

The deliverables for this project include:

- A trained reinforcement learning agent that can play Blackjack
- A report on the project, including the methodology, results, and future work
- A Python code repository with documentation on how to run the code
- A web application or API for others to use the trained agent

## Possible Framework :

The following is a detailed framework for the reinforcement learning project on Blackjack:

1. **Data Preprocessing**
- Load the dataset from Kaggle
- Preprocess the data to extract the relevant features and labels
- Split the data into training and testing sets
2. **Environment Setup**
- Define the state space, action space, and rewards for the game
- Implement the game environment using the OpenAI Gym framework
3. **Agent Design**
- Define the Q-learning or SARSA algorithm for the agent
- Implement the agent using Python and TensorFlow
4. **Training**
- Train the agent using the training data
- Evaluate the agent's performance on the testing data
- Plot the learning curve for the agent's performance over time
5. **Hyperparameter Tuning**
- Tune the hyperparameters for the Q-learning or SARSA algorithm
- Evaluate the agent's performance with different hyperparameter settings
- Choose the best hyperparameters based on the performance on the testing data
6. **Testing and Deployment**
- Test the agent's performance on new data
- Deploy the agent as a web application or API for others to use
7. **Documentation and Reporting**
- Document the code and project details
- Write a report on the project, including the methodology, results, and future work

## Code Explanation :

**Here is the simple explanation for the code you can find at code.py file.**

**Section 1: Environment**

In this section, we create the Blackjack environment using the OpenAI Gym library. The environment represents a simplified version of the popular card game in which the player tries to get a hand value as close to 21 as possible without going over. The environment provides observations of the player's current hand and the dealer's visible card, and allows the player to take actions (hit or stand) based on these observations. The environment also provides a reward signal (win, lose, or draw) at the end of each episode.

**Section 2: Agent**

In this section, we define the Q-learning agent that will learn to play blackjack using the environment defined in Section 1. The agent maintains a table of Q-values (i.e., estimates of the expected future rewards) for each possible state-action pair, and updates these values using the Q-learning update rule. The agent also uses an epsilon-greedy policy to choose actions during training, where it chooses a random action with probability epsilon and otherwise chooses the action with the highest Q-value estimate.

**Section 3: Data**

In this section, we generate training and testing data for the agent to learn from. We do this by running the environment for a fixed number of episodes and storing the observed state, action, reward, and next state at each time step. We use the first 80% of the data for training and the remaining 20% for testing.

**Section 4: Training**

In this section, we train the agent using the training data generated in Section 3. We do this by running the agent in the environment for a fixed number of episodes and updating its Q-value estimates using the Q-learning update rule. We also decay the epsilon parameter over time to encourage the agent to rely more on its learned Q-values and less on exploration. We store the history of episode rewards and Q-value updates over the training run for analysis and plotting.

**Section 5: Hyperparameter Tuning**

In this section, we perform a grid search over a set of hyperparameters (learning rate **alpha**, discount factor **gamma**, and epsilon decay rate **epsilon_decay**) using the scikit-learn **ParameterGrid** function. For each set of hyperparameters, we train the agent using the **train** function defined in Section 4, and calculate the mean episode reward over the entire training run. Finally, we print out the set of hyperparameters that yielded the best mean episode reward.

**Section 6: Testing**

In this section, we test the agent using the testing data generated in Section 3. We do this by running the agent in the environment for a fixed number of episodes without updating its Q-value estimates. We calculate the mean episode reward and win rate (i.e., the fraction of episodes won by the agent) and print out these metrics for evaluation.

**Section 7: Deployment**

In this section, we save the trained agent to a file using the **pickle** module, and then load the agent from the file. We then use the **get_action** method of the agent to play a game of blackjack interactively, printing out the agent's chosen action, the resulting reward, and whether the episode has terminated. This code can be used to deploy the trained agent in a real-world setting, such as a mobile app or web service.

## Future Work :

### Section 1: Improved Environment

One possible future work for this project is to improve the Blackjack environment to make it more realistic and challenging. This could involve adding additional rules and options to the game, such as splitting pairs, doubling down, and surrendering. It could also involve using a more advanced simulation model for the dealer's actions, such as a neural network or Monte Carlo Tree Search.

### Section 2: Advanced Agent

Another possible future work is to use a more advanced reinforcement learning algorithm for the agent, such as Deep Q-Networks (DQN) or Policy Gradient (PG) methods. These algorithms are able to learn more complex policies and can potentially achieve better performance on this task. It may also be beneficial to incorporate more sophisticated function approximation techniques, such as neural networks or kernel methods, to handle the high-dimensional state space.

### Section 3: Large-Scale Data Collection

To train more advanced agents, it may be necessary to collect large-scale data sets of Blackjack games. This could involve running thousands or millions of simulations of the game with different player strategies and dealer models, and storing the resulting state-action-reward sequences. This data can then be used to train and evaluate the agent, and potentially to generate new insights into the optimal playing strategies for Blackjack.

### Section 4: Distributed Training

To handle the large amounts of data and compute required for more advanced agents, it may be necessary to use distributed training techniques. This could involve parallelizing the Q-learning updates across multiple CPUs or GPUs, or using distributed deep learning frameworks such as TensorFlow or PyTorch. By using distributed training, we can potentially reduce the training time and achieve better performance on the task.

### Section 5: Transfer Learning

Another interesting direction for future work is to use transfer learning techniques to apply the agent to other related tasks. For example, we could train the agent on a similar card game such as Poker, and then fine-tune it on the Blackjack task using a smaller amount of data. This could potentially speed up the training process and improve the agent's performance on the task.

**Section 6: Real-World Deployment**

Finally, we could explore real-world deployment scenarios for the trained agent. This could involve integrating the agent into a mobile app or web service that allows users to play Blackjack against the agent, or using the agent to control a virtual dealer in an online casino. We would need to carefully evaluate the agent's performance and fairness in these scenarios, and ensure that it adheres to any relevant regulations and guidelines.

To implement these future work steps, we would need to invest additional time and resources into data collection, algorithm design, and software development. We may also need to collaborate with domain experts and other researchers in the field to ensure that our methods are sound and relevant. However, by pursuing these future work steps, we can potentially make significant progress towards developing more intelligent and adaptable agents for playing card games.

## Exercise Questions :

1. **Explain the difference between on-policy and off-policy reinforcement learning, and provide an example of each in the context of Blackjack.**

   In on-policy reinforcement learning, the agent learns from the actions it takes while following a specific policy, while in off-policy reinforcement learning, the agent learns from the actions it takes while following a different policy. In the context of Blackjack, an example of on-policy learning would be using Monte Carlo Exploring Starts to learn the optimal policy by exploring the state space and selecting actions based on a uniform random policy. An example of off-policy learning would be using Q-learning to learn the optimal policy while following an epsilon-greedy policy.

2. **What are some potential limitations of using Q-learning for solving the Blackjack problem?**

   One potential limitation of Q-learning for solving the Blackjack problem is that the state space is very large and high-dimensional, making it difficult to estimate accurate Q-values for all states and actions. Another limitation is that Q-learning assumes that the environment is stationary, which may not be the case in Blackjack if the dealer's strategy changes over time.

3. **Explain the concept of exploration-exploitation tradeoff in reinforcement learning, and provide an example of how it can be implemented in the Blackjack problem.**

   The exploration-exploitation tradeoff refers to the balance between exploring new actions and states in order to learn more about the environment, and exploiting the current knowledge to take actions that are known to be good. In the context of Blackjack, we can implement this tradeoff by using an epsilon-greedy policy, where with probability epsilon, we select a random action to explore the environment, and with probability 1-epsilon, we select the action with the highest Q-value based on the current knowledge.

4. **What are some potential drawbacks of using function approximation methods, such as neural networks, to estimate Q-values in reinforcement learning?**

   One potential drawback of using function approximation methods is that they may be prone to overfitting, particularly if the state space is very large or if the training data is limited. Another drawback is that the function approximation may not generalize well to unseen states, particularly if the feature representation is not well-chosen or if the function is not expressive enough.

5. **Explain the concept of policy gradient methods in reinforcement learning, and provide an example of how they could be used to solve the Blackjack problem.**

   Policy gradient methods are a class of reinforcement learning algorithms that directly optimize the policy function, rather than estimating the Q-values. They typically use gradient descent to update the policy based on the expected return under the current policy. In the context of Blackjack, an example of a policy gradient method would be REINFORCE, where we use a neural network to approximate the policy function, and use the likelihood ratio trick to compute the gradient of the expected return with respect to the network parameters. We can then update the parameters using stochastic gradient descent to maximize the expected return.

## Concept Explanation :

The algorithm we used in this project is called Q-learning, and it's a type of reinforcement learning algorithm that learns to find the optimal policy for an agent in an environment by estimating the Q-values of each state-action pair.

Now, you might be wondering, what the heck is a Q-value? Well, think of it as the expected reward you'd get if you took a specific action in a specific state, and then followed the optimal policy from there on out. So, for example, if you're playing Blackjack and you have a hand of 15, the Q-value for hitting might be higher than the Q-value for standing, since hitting might give you a chance to improve your hand and win the game.

The way Q-learning works is that the agent starts out with no knowledge of the optimal policy, and simply explores the environment by taking random actions and observing the resulting rewards. It then updates its estimates of the Q-values based on the rewards it receives and the estimates it already has, using a formula called the Bellman equation. Essentially, the Bellman equation says that the optimal Q-value for a state-action pair is equal to the immediate reward plus the discounted value of the expected future rewards.

As the agent continues to explore the environment and update its Q-value estimates, it gradually converges on the optimal policy, which maximizes the expected total reward over the long run. And the best part is, it does all of this without any explicit guidance or supervision from a human teacher!

So there you have it, a brief and hopefully entertaining explanation of Q-learning. Keep in mind that this is just one of many possible algorithms for solving reinforcement learning problems, and there's always more to learn and explore in the exciting field of machine learning.