**TF – RL Cookbook**

**Chapter 1**

1. Marco decision process

* a mathematical framework used to model decision making in situations where outcomes are partially random and depend on previous choices.
* The key assumption of an MDP is the probability of transitioning to a new state and receiving a reward depends only on the current state and action not on the history of previous States and actions. This is called “Markov property” and it allows the agent to make decisions based solely on the current state, without needing to keep track of the entire history.

1. Implementing A neural network-based policy (discrete actions)

* Policy function: maps between observation and actions. Formally a policy is a distribution over actions that prescribes A choosing of action given an observation.
* A discrete probability distribution can be used to represent a reinforcement learning agents’ policy when the agent can take one of the possible actions in an environment.

1. Implementing A neural network-based policy (continuous actions)

* A continuous probability distribution can be used to represent an reinforcement learning agent’s policy when the action is space of the environment contains real numbers.

1. Building a neural agent

* policy is defined to be a custom multilayer perceptron (MLP) policy based on the brain's neural network architecture.
* Categorical method: samples are valid action from the given unnormalized probabilities.
* MLP based policy: instead of a distribution policy here we use a neural network as the policy function.

1. Building a neural evolutionary agent

* Evolutionary methods are based on black-box optimization also known as “gradient free methods” since no gradient computation is used.
* Or new iteration, evolutionary process collects experienced data, using the current weights in agent’s brain.
* Elitism criterion is used to next piece data to gather the top experiences (with high rewards). This selected top experience was then used to update the weights in the agent’s brain.
* Elitism criteria: identify the best experience /individuals and carry them over to the next generation without any modifications.

**Chapter 2**

1. Building value-based RL agents

* Works by learning the state value function of the action value function in a given environment.
* State - value function:
  + It is a function that estimates the expected cumulative reward and agent can receive from a given state in the environment.
  + State value function estimates the expected sum of discounted rewards an agent can obtain by starting in a state “s” and following a policy “π”. It is a prediction of how valuable a state is for achieving the agents’ long-term goal.
  + Monte – Carlo and Temporal – Difference are various methods used to estimate state value functions.
* Action value functions:
  + actions value function is a function that estimates the expected cumulative reward an agent can receive by taking a particular action in a given state, and then following a certain policy.
  + Action value function estimates the expected sum of discounted rewards and agent can obtain by starting in a state “s”, taking an action “a” and then following a policy “π”. It is a prediction of how valuable a particular action is for achieving the agent’s long – term goal in a given state.
  + Q-learning or SARSA methods are used to estimate action value functions.
* Model-free RL problems:
  + Is a type of a problem where the agent learns to make decisions based on trial-and-error experiences, without having access to a model of the environment.
  + The agent learns a policy or an action value function directly from experience without explicitly estimating the dynamics of the reward function of the environment.
  + Often used in problems where the environment is complex or unknown or we are building an accurate model of the environment is computationally expensive or infeasible. They are also useful when the agent needs to adapt to changes in the environment or to new tasks, without having to re-learn the model.
  + Model free algorithms: Q-learning, SARSA, Monte – Carlo, Temporal – Difference

1. SARSA (State Action Reward State Action)

* The SARSA algorithm can be applied to model free control problems and allows us to optimize the value function of an unknown MDP.
* Silent greedy exploration policy:
  + Encourages the agent to explore new actions in the beginning of the learning process, when its knowledge of the environment is limited, and gradually decreases the exploration as the agent becomes more certain of the best actions. This way, the agent can discover new and potentially better actions while still exploiting the actions it has already learned to be good.

1. Implementing policy gradients

* These algorithms directly optimize for the best policy, which can lead to faster learning compared to value-based algorithms policy gradient algorithms are effective for problems and applications with high dimensional or continuous action spaces.
* Policy gradient is an on – policy algorithm that can only use experiences/trajectories or episode transitions from the same policy that is being optimized. Policy gradients have a problem of not converging or being stuck in a local minimum.
* Pseudocode for the policy gradient algorithm
  + Sample N trajectories by following the policy
  + Evaluate the gradient using the below expression:
  + Update the policy parameters.
  + repeat 123 until we find the optimal policy

1. Implementing actor critic RL algorithms

* Allow us to combine both value-based and policy-based reinforcement learning all in one agent.
* Two main components in the model; actor and critic.
  + Actor – Takes actions based on a policy and the goal of the actor is to maximize the expected reward.
  + Critic – Evaluates the quality of the actors’ actions and provides feedback to the actor. The goal is to accurately estimate the value of the state or state action pairs.
* Policy Gradient Method tends to get stuck at the local maxima.
* Value based methods like convergence guarantees, may have high variance (such as Q-learning) and are not very sample – efficient.
* The actor-critic model updates the policy and value functions based on the temporal difference error, which is the difference between the predicted value and the actual reward received. The algorithm used in the actor-critic model is based on stochastic gradient descent, where the parameters of both the actor and the critic are updated after each iteration.
* There are several variations of the actor-critic algorithm, including the A2C (Advantage Actor-Critic), A3C (Asynchronous Advantage Actor-Critic), and DDPG (Deep Deterministic Policy Gradient) algorithms, which have been shown to be effective in various applications, such as robotics, game playing, and natural language processing.
* **Stochastic Gradient Descent**
  + Stochastic gradient descent (SGD) is an optimization algorithm used in machine learning and deep learning to find the optimal values of the parameters of a model. It is a variant of gradient descent that randomly selects a small subset of the training data (known as a mini batch) to compute the gradient of the cost function, instead of using the entire training set at once.
  + Here is the general algorithm for SGD:
    1. Initialize the model parameters randomly.
    2. Choose a mini batch of size "m" from the training data.
    3. Compute the gradients of the cost function with respect to the model parameters using the mini batch.
    4. Update the model parameters using the gradients and a learning rate "alpha".
    5. Repeat steps 2-4 until convergence or for a fixed number of iterations.
  + In step 3, the gradients are computed using backpropagation, which is an efficient algorithm for computing the derivatives of a function with respect to its inputs. The learning rate "alpha" controls the step size of the parameter updates, and it is usually set to a small value to avoid overshooting the minimum of the cost function.
  + The use of mini batches in SGD provides several benefits over batch gradient descent. First, it reduces the computational cost of computing the gradients since we only need to compute them for a small subset of the data. Second, it introduces randomness into the parameter updates, which can help the algorithm escape from local minima and avoid overfitting. Finally, it allows for online learning, where the model can be updated continuously as new data arrives.
* **Backpropagation Algorithm**
  + Backpropagation is an algorithm used in neural networks to train the model by computing the gradients of the loss function with respect to the model's parameters. The gradient information is then used to update the parameters in a way that minimizes the loss function.
  + The backpropagation algorithm is based on the chain rule of calculus, which allows us to compute the gradient of a composite function by recursively applying the derivatives of its individual components. In the context of neural networks, the composite function is the output of the network, which is a function of the input data and the model's parameters.
  + Here is the general algorithm for backpropagation:
    1. Forward pass: compute the output of the network for a given input.
    2. Compute the loss function based on the output and the desired output (ground truth).
    3. Backward pass: compute the gradient of the loss function with respect to the output of each layer in the network.
    4. Use the chain rule to compute the gradient of the loss function with respect to each parameter in the network.
    5. Update the parameters using the gradients and a learning rate.
  + Repeat steps 1-5 for a fixed number of iterations or until convergence.
  + In step 3, the gradient is computed using the partial derivatives of the output of each layer with respect to its inputs and the gradient of the loss function with respect to the output of the layer. This is done recursively for each layer in the network, starting from the output layer and working backwards towards the input layer.
  + Backpropagation is a powerful algorithm that has enabled the training of deep neural networks with many layers. However, it can suffer from the vanishing gradient problem, where the gradient becomes very small for layers that are far from the output layer. Several techniques, such as weight initialization, activation functions, and batch normalization, have been developed to mitigate this issue.

**Chapter 3**

* + - 1. Implementing the Deep Q-Learning algorithm, DQN, and Double-DQN agent

**DQN (Deep Q-Learning Network)**

* DQN specifically is a deep neural network that is used to approximate the Q-function, which is a measure of the expected future reward of taking a particular action in a particular state. By training the DQN on a set of experiences, it can learn to estimate the Q-values for all possible actions in a given state and use that information to select the best action to take in each state.
* Here are some key differences and advantages of DQN over Actor-Critic:

1. *Model-free vs. Model-based*: DQN is a model-free algorithm, which means it does not require knowledge of the underlying dynamics of the environment. Actor-Critic, on the other hand, is a model-based algorithm that uses a learned model of the environment to make decisions. DQN's model-free approach can be advantageous in situations where the dynamics of the environment are complex and difficult to model.
2. *Single Network vs. Two Networks*: DQN uses a single neural network to approximate the Q-function, while Actor-Critic uses two separate networks: one to learn the policy (Actor) and another to learn the value function (Critic). The use of a single network in DQN simplifies the architecture and reduces the number of hyperparameters that need to be tuned.
3. *Sample Efficiency*: DQN can be more sample-efficient than Actor-Critic because it stores experiences in a replay buffer and samples from it randomly during training, allowing for better use of past experiences. Actor-Critic, on the other hand, typically updates the policy based on the most recent experience.
4. *Stability*: DQN is known to be more stable than Actor-Critic because it uses a technique called target network, which uses a separate network with frozen parameters to compute target Q-values. This stabilizes the learning process and prevents the network from overfitting to the current set of experiences.

* Overall, DQN is a powerful and flexible algorithm that has proven to be effective in a wide range of applications, while Actor-Critic can be advantageous in situations where a learned model of the environment is available, and the dynamics of the environment are relatively simple.

**Replay Buffer**

* A replay buffer is a memory structure used in the training of deep reinforcement learning algorithms, such as DQN. It is essentially a circular buffer that stores a collection of experiences, consisting of the agent's observations, actions, rewards, and resulting next observations.
* During the training process, the DQN network is updated by sampling batches of experiences from the replay buffer, rather than using the most recent experience. This has several benefits:
  + ***Reducing sample correlation:*** By sampling experiences randomly from the replay buffer, the network is less likely to be biased towards the most recent experiences and can avoid overfitting to the current distribution of data.
  + ***Improving sample efficiency:*** By reusing past experiences, the agent can learn more efficiently from a smaller number of experiences, which is especially important in applications where collecting new data is expensive or time-consuming.
  + ***Breaking temporal correlation:*** The experiences stored in the replay buffer can be randomly sampled, which reduces the correlation between consecutive samples and provides a more diverse and representative set of experiences for training.
* Overall, the replay buffer is a crucial component of the DQN algorithm, as it helps to improve sample efficiency, reduce sample correlation, and provide a more stable and efficient learning process.

**The Double DQN Agent**

* The Double DQN (Double Deep Q-Network) agent is an extension of the original DQN algorithm introduced by DeepMind in 2015. The Double DQN agent is designed to address a problem with the original DQN algorithm, where the overestimation of Q-values can lead to suboptimal policies.
* In the original DQN algorithm, the Q-values *used to select actions* and *update the network* are estimated using the same network that is being updated. This can lead to a *positive feedback loop*, where the network's estimates of Q-values become increasingly optimistic over time, resulting in overestimation of the true Q-values.
* The Double DQN agent solves this problem by *using two separate neural networks***: one to estimate the Q-values**, and **another to select the actions**. *Specifically*, the Double DQN agent uses the first network to select the action with the highest Q-value, but then uses the second network to estimate the Q-value of that action. This helps to reduce overestimation of Q-values, resulting in a more accurate and stable learning process.
* In summary, the Double DQN agent is an extension of the DQN algorithm that improves the accuracy of Q-value estimates by using two separate networks to select actions and estimate Q-values. By reducing overestimation of Q-values, the Double DQN agent can learn more accurate and robust policies in a variety of reinforcement learning environments.

Updates to the weights in the DQN are performed as per the following Q learning equation.

*Gradient of Q value*

*Predicted Q value.*

*Max Q value for s’*

Here, is the change in the parameters (weights) of the DQN, s is the current state, a is the current action, *s'* is the next state, *w* represents the weights of the DQN, is the discount factor, is the learning rate, and represents the Q-value for the given state *(s)* and action *(a)* predicted by the DQN with a weight *w*.

1. The Dueling DQN agent

* The Dueling-DQN architecture aims to improve the efficiency and stability of the traditional DQN algorithm by separating the Q-function into two streams: one for estimating the state-value function and one for estimating the action advantage function. The state-value function estimates the value of being in a particular state, regardless of the action taken, while the action advantage function estimates the relative advantage of each action in that state.
* By decoupling the estimation of state values and action advantages, the Dueling-DQN network can learn which states are valuable and which actions are advantageous independently. This can help the network generalize better across different states and actions, and ultimately lead to improved performance in reinforcement learning tasks.



* The DQN (top half of the diagram) has a linear architecture and predicts a single quantity , whereas the Dueling-DQN has a bifurcation in the last layer and predicts multiple quantities.

1. The Dueling Double DQN algorithm and DDDQN agent

* A DDDQN (Double Dueling Deep Q-Network) agent is a variant of the traditional DQN (Deep Q-Network) algorithm used in reinforcement learning. It combines two key improvements to the DQN algorithm: double Q-learning and dueling network architecture.
* Double Q-learning is a technique that addresses the overestimation of Q-values that can occur in traditional Q-learning algorithms. By using two separate networks to independently estimate the Q-values for each action, DDDQN agents can avoid the overestimation problem and converge to more accurate Q-values.
* The dueling network architecture used in DDDQN agents is similar to that of Dueling-DQN networks. It separates the estimation of state values and action advantages, which can improve the efficiency and stability of the network in reinforcement learning tasks.
* Double Q-learning corrects DQN from overestimating the action values. The Dueling architecture uses a modified architecture to separately learn the state value function (V) and the advantage function (A). This explicit separation allows the algorithm to learn faster, especially when there are many actions to choose from and when the actions are very similar to each other.
* The dueling architecture enables the agent to learn even when only one action in a state has been taken, as it can update and estimate the state value function, unlike the DQN agent, which cannot learn from actions that were not taken yet.

1. Deep Recurrent Q- Learning Agent (DQRN)

* DQRN stands for Deep Q-Network with Recurrent Neural Networks. It is a type of reinforcement learning agent that uses a combination of deep Q-learning and recurrent neural networks (RNNs) to make decisions in sequential decision-making tasks.
* In DQRN, the deep Q-learning algorithm is used to estimate the optimal action-value function, which maps a state-action pair to its expected future reward. The recurrent neural network is used to capture the sequential structure of the input data, allowing the agent to make decisions based on the current state and the history of past states and actions.
* By using an RNN, the DQRN agent can handle non-Markovian environments where the current state may not provide all the necessary information for making optimal decisions. This makes DQRN a suitable choice for tasks such as video games or robotics, where the agent needs to consider the previous states and actions to make informed decisions.
* ***Recurrent Learning***:
  + Is used to process sequential data, where the output of each step depends not only on the current input but also on the previous inputs and outputs. In other words, the model maintains a memory of the past inputs and uses this information to make decisions about the current input.
  + Is commonly used in natural language processing, speech recognition, and time-series analysis, among other applications. In these tasks, the input data is often sequential and has a temporal dependency. For example, in natural language processing, the meaning of a word may depend on the words that came before it in the sentence.
  + Is implemented using recurrent neural networks (RNNs), which are neural networks with feedback connections that allow the model to maintain a memory of past inputs. The most common type of RNN is the Long Short-Term Memory (LSTM) network, which is designed to address the problem of vanishing gradients that can occur in traditional RNNs.
* Long Short-Term Memory (LSTM) Network
  + A Long Short-Term Memory (LSTM) network is a type of recurrent neural network (RNN) that is designed to handle the vanishing gradient problem that can occur in traditional RNNs. The vanishing gradient problem refers to the issue where the gradients used to update the model parameters become very small as they are backpropagated through many time steps, which can make learning difficult.
  + The key feature of an LSTM network is its ability to maintain a cell state that can store information over many time steps. This is accomplished using three gating mechanisms: the input gate, forget gate, and output gate. These gates regulate the flow of information into and out of the cell state, allowing the network to selectively remember or forget information as needed.
  + The input gate determines how much new information should be added to the cell state, while the forget gate decides which information should be discarded. The output gate determines how much of the cell state should be passed on to the next time step.
  + The LSTM network is trained using backpropagation through time, which involves propagating the errors back through each time step and adjusting the model parameters accordingly.

1. Asynchronous Advantage Actor-Critic algorithm (A3C agent)

* In simple terms, the crux of the A3C algorithm can be summarized in the following sequence of steps for each iteration:



* The steps repeat again from top to bottom for the next iteration and so on until convergence.
* The Asynchronous Advantage Actor-Critic (A3C) algorithm is a deep reinforcement learning algorithm that is designed to learn policies for sequential decision-making problems.
* The A3C algorithm employs a deep neural network to represent the policy and value function, which are used to select actions and estimate the expected return, respectively. The algorithm uses multiple actor-learner threads that run asynchronously, each with its own copy of the neural network. This allows the algorithm to explore the state space more efficiently and accelerate learning.
* At each time step, each actor interacts with the environment and collects experience data, which is used to update the corresponding learner's copy of the neural network. The learners use this data to compute the policy gradient and the value gradient, which are used to update the parameters of the neural network.
* The advantage of the A3C algorithm is that it is both computationally efficient and effective at learning policies for sequential decision-making problems. It has been successfully applied to a wide range of applications, including video game playing, robotics, and natural language processing.
* Here are the main steps of the A3C algorithm:
  1. Initialize the policy network and value network with random weights.
  2. Initialize a set of worker agents that operate in parallel to explore the state space.
  3. For each worker agent, initialize its local copy of the policy and value networks with the global weights.
  4. For each time step, the worker agent interacts with the environment and collects a sequence of observations, actions, and rewards.
  5. The worker agent uses its local copy of the policy network to compute the probabilities of each action given the current observation.
  6. The worker agent selects an action using the computed probabilities and executes it in the environment.
  7. The worker agent computes the advantage function, which is an estimate of how much better the selected action was compared to the average action value for the current state.
  8. The worker agent updates its local copy of the value network using the advantage function and the observed rewards.
  9. The worker agent updates its local copy of the policy network using the advantage function and the gradients of the log-probabilities of the selected actions.
  10. Every N time steps, the worker agent updates the global weights of the policy and value networks by averaging its local copies.
  11. Repeat steps 4-10 for a fixed number of episodes or until convergence.
* The A3C algorithm combines the benefits of actor-critic methods, which learn both the policy and value function, with the advantages of asynchronous methods, which can explore the state space more efficiently. The algorithm has been shown to be effective in a wide range of reinforcement learning tasks, including video games, robotics, and natural language processing.

1. Proximal Policy Optimization Algorithm (PPO)

* The Proximal Policy Optimization (PPO) algorithm is a reinforcement learning algorithm used to train agents in environments where an agent interacts with an environment and learns to make decisions based on rewards. PPO is designed to optimize policy functions, which determine the agent's behavior in a given state.
* PPO belongs to the family of policy gradient algorithms and is specifically developed to address some limitations of earlier algorithms like TRPO (Trust Region Policy Optimization). It aims to strike a balance between sample efficiency and policy stability during training.
* The PPO algorithm works by iteratively collecting experience data from the environment and updating the policy based on that data. Here's a simplified overview of the key steps in the PPO algorithm:

1. ***Data Collection***: The agent interacts with the environment by taking actions according to its current policy. During this process, the agent collects a batch of trajectories, which consist of state-action pairs, rewards, and other relevant information.
2. ***Policy Evaluation***: The collected trajectories are then used to estimate the advantages of different actions. Advantages represent how much better or worse an action is compared to other actions in a given state.
3. ***Policy Update***: The policy is updated based on the collected experience data. PPO employs a surrogate objective function that constrains the policy update to prevent significant deviations from the original policy. This constraint helps to maintain stability during training.
4. ***Optimization:*** PPO maximizes the surrogate objective function using optimization techniques such as stochastic gradient descent (SGD) or Adam. The objective is to find the policy parameters that maximize the expected cumulative rewards.
5. ***Repeat:*** Steps 1 to 4 are repeated for multiple iterations, allowing the agent to gradually improve its policy over time.

* The key idea behind PPO is to balance exploration (trying new actions) and exploitation (taking advantage of known good actions). It achieves this by optimizing the policy in small steps, which prevents drastic changes and allows for more stable learning.
* PPO has gained popularity due to its effectiveness, ease of implementation, and ability to handle both continuous and discrete action spaces. It has been successfully applied to various tasks, including robotic control, game playing, and autonomous driving, among others.

7. Deep Deterministic Policy Gradient

* DDPG (Deep Deterministic Policy Gradient) is an algorithm that combines deep learning with reinforcement learning to train agents in continuous action spaces. It is designed to solve tasks where the agent needs to choose actions from a continuous action space, such as controlling a robotic arm or steering a car.
* DDPG is an extension of the DQN (Deep Q-Network) algorithm, which is commonly used for discrete action spaces. DDPG overcomes the limitations of DQN and extends it to continuous action spaces by using an actor-critic architecture.
* Here's a breakdown of the key components and steps involved in the DDPG algorithm:

1. ***Actor-Critic Architecture***: DDPG uses two neural networks, an actor network, and a critic network. The actor network takes the current state as input and outputs the best action to take in that state. The critic network evaluates the quality of the action taken by the actor by estimating the expected cumulative reward.
2. ***Experience Replay:*** DDPG utilizes experience replay, a technique also used in DQN. The agent collects experience by interacting with the environment, storing transitions consisting of the state, action, reward, and next state in a replay buffer. During training, a mini batch of experiences is randomly sampled from the replay buffer, which helps break the correlation between consecutive experiences and provides more stable training.
3. ***Exploration and Exploitation:*** To balance exploration and exploitation, DDPG introduces an exploration policy called "noise addition." Gaussian noise or other types of exploration noise are added to the action outputs of the actor network. This encourages the agent to explore different actions while still exploiting the learned policy.
4. ***Target Networks:*** To stabilize the learning process, DDPG employs target networks. There are two sets of target networks: a target actor network and a target critic network. These networks are periodically updated by slowly blending their weights with the weights of the corresponding actor and critic networks. The target networks provide a fixed target for the actor and critic networks, reducing the potential for feedback loops during learning.
5. ***Loss and Optimization:*** The critic network is trained using the mean-squared error loss between the predicted Q-value and the target Q-value. The actor network is trained using the gradients provided by the critic network. The optimization is typically performed using techniques such as stochastic gradient descent (SGD) or Adam.

* By combining these elements, DDPG learns a policy that can handle continuous action spaces. It iteratively improves the actor and critic networks by learning from the experiences stored in the replay buffer. DDPG has been successfully applied to various tasks, including robotic control, locomotion, and continuous control in simulated environments.
* Overall, DDPG is an effective algorithm for training agents in tasks with continuous action spaces, leveraging the power of deep neural networks and reinforcement learning techniques.