Reinforcement Learning

Learning to Optimize Rewards

- A software agent makes observations and takes actions within an environment. In return, it receives rewards from the environment. This is reinforcement learning.
    - The agent's objective is to learn to act in a way that will maximize its expected rewards over time.
    - Some systems may not have positive rewards.

Policy Search

- The software agent uses an algorithm to determine its actions. This algorithm is called a policy.
- The policy can be a neural network taking observations as inputs and outputting the action to take.
    - Policies that involve randomness are called stochastic policies.
    - Many policies may involve parameters. We can find the best combination of parameters in a process called policy search.
        - Finding a good set of parameters is tedious and costly when the policy space is too large.
    - We can also use genetic algorithms where we iteratively create generations of a specific number of policies, kill the worst policies, and make each survivor produce offspring.
        - An offspring is a copy of its parent plus some random variation.
        - The surviving policies are passed to the next generation, akin to survival of the fittest.
    - Lastly, we can perform policy gradients where we evaluate the rewards' gradients about the policy parameters and tweak these parameters by following the gradients toward higher rewards.

Introduction to OpenAI Gym

- Training is rigid and slow in the real world, so we generally need a simulated environment to implement the training algorithms.
- OpenAI Gym provides a database of simulated environments (Atari games, board games, 2D and 3D physical simulations, etc.)
    - The database is accessible through the **gym()** class. We can create an environment by providing an environment name to the **make()** function.
    - We can use the **gym.envs.registry()** function to access the names of all available environments.

    - We initialize the environment by using the **reset()** method. This function returns a tuple containing the observations and extra information about the environment.
    - The **step()** function returns five variables:
        - Observation: a new observation
        - Rewards: the value of the rewards
        - Done: indicates if the episode is over (either due to the environment's rules or because the system won)
        - Truncated: indicates if the episode was interrupted (e.g. if a maximum value was reached)
        - Information: environment-specific dictionary providing extra information
    - We should call the **close()** method once we finish using an environment.

Neural Network Policies

- We can make a neural network that estimates a probability p for each action and select an action randomly based on the probability.
    - For example, if there are two options and p is 0.7, the net will pick one action with 70% probability and the other with 30%.
- We use randomness so that the net finds the right balance between exploring new actions and exploiting the actions it knows will work best.
    - The net may find an action that works well, but if it only picks that action, it will never be able to find an action that leads to higher rewards.

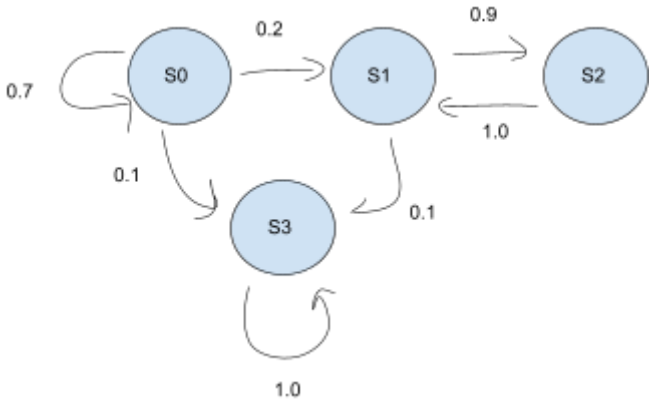
Evaluating Actions: The Credit Assignment Problem

- In reinforcement learning, the agent is guided only by the rewards it receives. However, it doesn't know which action led to a reward. This is the credit assignment problem.
- One solution is to evaluate an action based on the sum of all the rewards after it and apply a discount factor γ at each step.
    - The sum of discounted rewards is called the action's return.
    - If γ is close to zero, future rewards won't count as much as immediate rewards. On the other hand, if γ is close to one, future rewards will count almost as much as immediate rewards.
    - Typical γ values range from 0.9 to 0.99.
- Sometimes, a good action may be followed by several bad actions. In this case, a good action will get a low return.
- However, with enough iterations, on average, good actions will get a higher return than bad ones. This is called the action advantage.
    - For this advantage, we must run many episodes and normalize the action returns (i.e., subtract the mean and divide by the standard deviation).

Policy Gradients

- PG algorithms optimize the policy parameters by following the gradients toward higher rewards.
    - First, we allow the neural network policy to play the game several times. At each step, it computes the gradients that would make the chosen action more likely but doesn't apply them yet.
    - Once we have run several episodes, compute each action's advantage.
    - If the action advantage is positive, the action was probably good, and we apply the gradients to make this action more likely to be chosen in the future.
    - Conversely, if the action advantage is negative, the action was probably bad, so we apply the opposite gradients to make this action less likely in the future.
        - For this, we multiply each gradient vector by the action advantage.
    - Lastly, we compute the mean of all resulting gradient vectors and perform a gradient descent step.
        - TensorFlow's **GradientTape()** class may come in handy here.
- This PC algorithm is highly sample inefficient, needing to explore the game for a long time before making significant progress.

Markov Decision Processes

- Markov decision processes (MDPs) are based on Markov chains.
    - These processes have a fixed number of states and randomly transition from one state to another at each step.
    - The probability of the process to transition from state s to state s' is fixed and depends on the pair (s, s'), not on past states.
    - 

    - Eventually, the process will fall into state s3 and remain there forever. This is called the terminal state.
    - The agent can choose one of several possible actions, and the transition probabilities depend on the chosen action.

- The Bellman optimality equation estimates the optimal state value.
    - If the agent acts optimally, the optimal value of the current state is equal to the reward it will get on average after taking one optimal action plus the expected optimal value for all possible next states that this action can lead to.
    - The equation is $V^*(s) = max\sum_{s'} T(s, a, s')\left[R(s, a, s') + \gamma \cdot V^*(s')\right) for\ all\ s$ where
        - V*(s) is the sum of all discounted feature rewards the agent can expect on average after it reaches the state, assuming it acts optimally.
        - T(s, a, s') is the transition probability from state s to state s' given that the agent chose action a.
        - R(s, a, s') is the reward the agent receives when it goes from state s to state s', given that the agent chose action a.
        - γ is the discount factor.
- The value iteration algorithm uses the equation to optimize the optimal state value of every possible state.
    - First, we initialize the state value estimates to zero and iteratively update them using the algorithm.
    - Given enough time, the algorithm guarantees that these estimates converge to the optimal state values corresponding to the optimal policy.
    - $V_{k+1}(s) \leftarrow max\sum_{s'} T(s, a, s')\left[R(s, a, s') + \gamma \cdot V_k(s')\right) for\ all\ s$ where
        - $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.
- The Q-value iteration algorithm estimates the optimal state-action values (Q-values or quality values).
    - The optimal Q-value of the state-action pair (s, a) – noted Q*(s, a) – is the sum of discounted future rewards the agent can expect on average after it reaches state s and chooses action a but before it sees the outcome of this action.
    - $Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s')\left[R(s, a, s') + \gamma \cdot max\, Q_k(s', a')\right) for\ all\ (s, a)$
    - Once we have the optimal Q-values, we can define the optimal policy $\pi^*(s)$. When the agent is in state s, it should choose the action with the highest Q-value $\pi^*(s) = argmax\, Q^*(s, a)$.

Temporal Difference Learning

- The agent must experience each state and transition to know the rewards and experience them multiple times to have a reasonable estimate of the transition probabilities.
- The temporal difference (TD) learning algorithm is similar to the Q-value iteration algorithm but considers that the agent has only partial knowledge of the MDPs.
    - The agent uses an exploration policy to explore the MDP.
        - This policy can be as simple as a purely random policy.

- As it progresses, the TD algorithm updates the estimates of the state values based on the transitions and rewards.
- The equation is $V_{k+1}(s) \leftarrow (1 - \alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$, which is equivalent to $V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s') \; with \; \delta_k(s, r, s') = r + \gamma \cdot V_k(s') - V_k(s)$ where
  - α is the learning rate.
  - r + γ * V_k(s) is the TD target.
  - $\delta_k(s, r, s')$ is the TD error.
- For each state s, the algorithm stores a running average of the immediate rewards the agent gets upon leaving that state, plus the rewards it expects to get later, assuming it acts optimally.

Q-Learning

- Q-learning watches an agent play (e.g., randomly) and gradually improves its estimates of the Q-values. Then, it chooses the action with the highest Q-value.
  - The equation for this algorithm is $Q(s, a) \leftarrow r + \gamma \cdot max \, Q(s', a')$.
  - For each state-action pair (s, a), the algorithm stores a running average of the rewards r the agent gets upon leaving the state s with action a, plus the sum of discounted future rewards it expects to get.
  - Once we initialize the Q-values, we run the algorithm with learning rate decay.
  - This algorithm will converge to the optimal Q-values, but it will take many iterations and a lot of hyperparameter tuning.
  - The Q-learning algorithm is called an off-policy algorithm because the policy being trained is not necessarily the one used during training.
  - Conversely, the policy gradients algorithm is an on-policy algorithm: it explores the world using the policy being trained.

Exploration Policies

- The ε-greedy policy acts randomly at each step with probability ε, or greedily with probability 1 - ε (i.e., choosing the action with the highest Q-value).
  - This policy will spend more time exploring the exciting parts of the environment while still exploring unknown regions of the MDP.
  - Typically, we start with a high value for ε (e.g., 1.0) and gradually reduce it (e.g., 0.05).
- We can also encourage the exploration policy to try actions it has yet to attempt much before.
  - We can use the equation $Q(s, a) \leftarrow r + \gamma \cdot max \, f(Q(s', a'), \, N(s', \, a'))$ where
    - N(s', a') counts the number of times the action a' was chosen in state s'.
    - f(Q, N) is an exploration function, such as f(Q, N) = Q + k / (1 + N), where k is a curiosity hyperparameter.

Approximate Q-Learning and Deep Q-Learning

- Q-learning does not scale well to large MDPs with numerous states and actions.
- The solution is to find a function Qθ(s, a) that approximates the Q-value of any state-action pair (s, a) using a manageable number of parameters. This is approximate Q-learning.
    - A DNN used to estimate Q-values is called a deep Q-network (DQN), and using a DQN for approximate Q-learning is called deep Q-learning.
    - We get an approximate future Q-value for each possible action. Then, we pick and discount the highest.
    - The equation is $y(s,a) = r + \gamma \cdot max\, Q_\theta(s', a')$ where y(s, a) is the target Q-value and Q_θ(s, a) is the estimated Q-value.
    - Then, we run a training step using any gradient descent algorithm. We try to minimize the squared error between the estimated and target Q-values.
    - The neural network inputs a state and outputs an approximate Q-value for each possible action.
- We can also store all experiences in a replay buffer (or replay memory) and sample a random training batch at each training iteration.
- DQNs can suffer from catastrophic forgetting. As the agent explores the environment, it updates its policy, but what it learns in one part of the environment may break what it learned earlier in other parts of the environment.

Deep Q-Learning Variants

- The fixed Q-value target algorithm uses two DQNs.
    - One DQN is an online model that learns at each step and moves the agent.
    - The other is the target model that defines the targets.
    - The Q-value targets are more stable since the target model is updated much less often than the online model.
- The double DQN algorithm fixes issues of the above DQN.
    - This algorithm uses the online model instead of the target model when selecting the best actions for the next states and uses the target model only to estimate the Q-values for these best actions.
    - This fixes the issue of the target network being prone to overestimating Q-values.
- Prioritized experience replay (PER) or importance sampling (IS) samples important instances from the replay buffer more frequently.
    - Experiences are considered “important” if they will likely lead to fast learning progress.
    - We use the TD error to identify important experiences. A large TD error indicates that a transition (s, a, s′) is surprising and, thus, probably worth learning from.
    - However, since the samples will be biased toward important experiences, we must compensate for this bias during training by down-weighting the experiences according to their importance.

        - β is the hyperparameter that controls how much we want to compensate for the importance sampling bias (0 means not at all, 1 means entirely).
- Dueling DQN (DDQN) estimates the value of the state and the advantage of each possible action.
    - Since the best action should have an advantage of 0, the model subtracts the maximum predicted advantage from all predicted advantages.
    - DDQNs can be combined with other algorithms. In general, many RL techniques can be combined.

Overview of Some Popular RL Algorithms

- AlphaGo
    - AlphaGo uses a variant of Monte Carlo tree search (MCTS).
    - It selects the best move after running many simulations, repeatedly exploring the search tree starting from the current position, and spending more time on the most promising branches.
    - AlphaGo uses a policy network to select moves rather than playing randomly.
- Actor-critic algorithms
    - An actor-critic agent contains two neural networks: a policy net and a DQN.
    - The DQN is trained normally, but in the policy net, the agent (actor) relies on the action values estimated by the DQN (critic).
- Asynchronous advantage actor-critic (A3C)
    - Multiple agents learn in parallel, exploring different copies of the environment.
    - Each agent pushes some weight updates to a master network, pulling the latest weights from that network.
    - Instead of estimating the Q-values, the DQN estimates the advantage of each action, stabilizing training.
- Soft actor-critic
    - This algorithm tries to be as unpredictable as possible while getting as many rewards as possible.
    - It learns rewards and how to maximize the entropy of its actions.
- Curiosity-based exploration
    - This algorithm ignores rewards and makes the agent curious to explore the environment.
    - The agent continuously tries to predict the outcome of its actions and seeks situations where the outcome does not match its predictions.
    - If the outcome is predictable (boring), it goes elsewhere.
        - If the outcome is unpredictable, but the agent notices it has no control, it gets bored.
        - If it loses several times, it gets bored, so it learns to avoid actions leading to losing, improving its performance.
- Open-ended learning (OEL)
    - OEL aims to train agents capable of endlessly learning new and interesting tasks, typically generated procedurally.

- The agent aims to walk as fast as possible while avoiding obstacles.
- The algorithm starts with simple environments but gradually gets harder over time; this is called curriculum learning.
- Moreover, although each agent is only trained within one environment, it must regularly compete against other agents across all environments.