POLICY GRADIENT METHODS

What are Policy Gradient Methods?

- Before: Learn the values of actions and then select actions based on their estimated action-values. The policy was generated directly from the value function
- We want to learn a parameterised policy that can select actions without consulting a value function. The parameters of the policy are called policy weights
- A value function may be used to learn the policy weights but this is not required for action selection
- Policy gradient methods are methods for learning the policy weights using the gradient of some performance measure with respect to the policy weights
- Policy gradient methods seek to maximise performance and so the policy weights are updated using gradient ascent

Policy-based Reinforcement Learning

- Search directly for the optimal policy π^*
- Can use any parametric supervised machine learning model to learn policies $\pi(a \mid s; \theta)$ where θ represents the learned parameters
- Recall that the optimal policy is the policy that achieves maximum future return

Notation

- Policy weight vector $\boldsymbol{\theta}$
- The policy is $\pi(a \mid s, \boldsymbol{\theta})$, which represents the probability that action a is taken in state s with policy weight vector $\boldsymbol{\theta}$
- If using learned value functions, the value function's weight vector is $\mathbf{w}$
- Performance measure $\eta(\boldsymbol{\theta})$
- Episodic case: $\eta(\boldsymbol{\theta}) = v_{\pi}(s_0)$
- Performance is defined as the value of the start state under the parameterised policy

Policy Approximation

- The policy can be parameterised in any way provided $\pi(a \mid s, \theta)$ is differentiable with respect to its weights
- To ensure exploration, we generally require that the policy never becomes deterministic and so $\pi(a \mid s, \theta)$ is in interval (0, 1)
- Assume discrete and finite set of actions
- Form parameterised numerical preferences $h(s, a, \theta)$ for each state-action pair
- Can use a deep neural network as the policy approximator
- Use softmax so that the most preferred actions in each state are given the highest probability of being selected

$$\pi(a|s,\boldsymbol{\theta}) \doteq \frac{\exp(h(s,a,\boldsymbol{\theta}))}{\sum_b \exp(h(s,b,\boldsymbol{\theta}))}$$

Types of Policy Gradient Method

- Finite Difference Policy Gradient
- Monte Carlo Policy Gradient
- Actor-Critic Policy Gradient

Finite Difference Policy Gradient

- Compute the partial derivative of the performance measure $\eta(\boldsymbol{\theta})$ with respect to θ_k using finite difference gradient approximation
- Compute performance measure $\eta(\boldsymbol{\theta})$
- Perturb $\boldsymbol{\theta}$ by small positive amount epsilon in the kth dimension and compute performance measure $\eta(\boldsymbol{\theta} + u_k\varepsilon)$ where u_k is unit vector that is 1 in kth component and 0 elsewhere
- The partial derivative is approximated by

  $$(\eta(\boldsymbol{\theta} + u_k\varepsilon) - \eta(\boldsymbol{\theta})) / \varepsilon$$

- $\theta_k \leftarrow \theta_k + \alpha (\eta(\boldsymbol{\theta} + u_k\varepsilon) - \eta(\boldsymbol{\theta})) / \varepsilon$
- Perform gradient ascent using the partial derivative
- Simple, noisy and inefficient procedure that is sometimes effective
- This procedure works for arbitrary policies (can be non-differentiable)

REINFORCE: Monte Carlo Policy Gradient

- We want a quantity that we can sample on each time step whose expectation is equal to the gradient
- We can then perform stochastic gradient ascent with this quantity

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \gamma^t G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})}.$$

REINFORCE Properties

- On-policy method
- Uses the complete return from time t, which includes all future rewards until the end of the episode
- REINFORCE is thus a Monte Carlo algorithm and is only well-defined for the episodic case with all updates made in retrospect after the episode is completed
- Note that the gradient of log x is the gradient of x divided by x by the chain rule

REINFORCE Algorithm

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta}), \forall a \in \mathcal{A}, s \in \mathcal{S}, \boldsymbol{\theta} \in \mathbb{R}^n$
Initialize policy weights $\boldsymbol{\theta}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G_t \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t,\boldsymbol{\theta})$

Actor-Critic Methods

- Methods that learn approximations to both policy and value functions are called actor-critic methods
- Actor refers to the learned policy
- Critic refers to the learned value function, which is usually a state-value function
- The critic is bootstrapped – the state-values are updated using the estimated state-values of subsequent states
- The number of steps in the actor-critic method controls the degree of bootstrapping

One-step Actor-Critic Update Rules

- On-policy method
- The state-value function update rule is the TD(0) update rule
- The policy function update rule is shown below.
- For n-step Actor-Critic, simply replace $G_t^{(1)}$ with $G_t^{(n)}$

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha\left(G_t^{(1)} - \hat{v}(S_t,\mathbf{w})\right)\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})} \\ &= \boldsymbol{\theta}_t + \alpha\left(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})\right)\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}.\end{aligned}$$

One-step Actor-Critic

One-step Actor-Critic (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta}), \forall a \in \mathcal{A}, s \in \mathcal{S}, \boldsymbol{\theta} \in \mathbb{R}^n$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w}), \forall s \in \mathcal{S}, \mathbf{w} \in \mathbb{R}^m$
Parameters: step sizes $\alpha > 0$, $\beta > 0$

Initialize policy weights $\boldsymbol{\theta}$ and state-value weights $\mathbf{w}$
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \beta \delta \nabla_{\mathbf{w}} \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha I \delta \nabla_{\boldsymbol{\theta}} \log \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

ASYNCHRONOUS REINFORCEMENT LEARNING

Asynchronous Methods for Deep Reinforcement Learning

What is Asynchronous Reinforcement Learning?

- Use asynchronous gradient descent to optimise controllers
- This is useful for deep reinforcement learning where the controllers are deep neural networks, which take a long time to train
- Asynchronous gradient descent speeds up the learning process
- Can use one multi-core CPU to train deep neural networks asynchronously instead of multiple GPUs

Parallelism (1)

- Asynchronously execute multiple agents in parallel on multiple instances of the environment
- This parallelism decorrelates the agents' data into a more stationary process since at any given time-step, the agents will be experiencing a variety of different states
- This approach enables a larger spectrum of fundamental on-policy and off-policy reinforcement learning algorithms to be applied robustly and effectively using deep neural networks
- Use asynchronous actor-learners (i.e. agents). Think of each actor-learner as a thread
- Run everything on a single multi-core CPU to avoid communication costs of sending gradients and parameters

Parallelism (2)

- Multiple actor-learners running in parallel are likely to be exploring different parts of the environment
- We can explicitly use different exploration policies in each actor-learner to maximise this diversity
- By running different exploration policies in different threads, the overall changes made to the parameters by multiple actor-learners applying updates in parallel are less likely to be correlated in time than a single agent applying online updates

No Experience Replay

- No need for a replay memory. We instead rely on parallel actors employing different exploration policies to perform the stabilising role undertaken by experience replay in the DQN training algorithm
- Since we no longer rely on experience replay for stabilising learning, we are able to use on-policy reinforcement learning methods to train neural networks in a stable way

Asynchronous Algorithms

- Asynchronous one-step Q-learning
- Asynchronous one-step SARSA
- Asynchronous n-step Q-learning
- Asynchronous Advantage Actor-Critic (A3C)

Asynchronous one-step Q-learning

- Each thread interacts with its own copy of the environment and at each step, computes the gradient of the Q-learning loss
- Use a globally shared and slowly changing target network with parameters θ^- when computing the Q-learning loss
- DQN uses a globally shared and slowly changing target network too – the network is updated using a mini-batch of experience tuples drawn from replay memory
- The gradients are accumulated over multiple time-steps before being applied (similar to using mini-batches), which reduces chance of multiple actor-learners overwriting each other's updates

Exploration

- Giving each thread a different exploration policy helps improve robustness and generally improves performance through better exploration
- Use epsilon-greedy exploration with epsilon periodically sampled from some distribution by each thread

Asynchronous one-step Q-learning Algorithm

Algorithm 1 Asynchronous one-step Q-learning - pseudocode for each actor-learner thread.

// Assume global shared θ, θ^-, *and counter* $T = 0$.
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state s
repeat
  Take action a with ϵ-greedy policy based on $Q(s, a; \theta)$
  Receive new state s' and reward r
  $$y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases}$$
  Accumulate gradients wrt θ: $d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$
  $s = s'$
  $T \leftarrow T + 1$ and $t \leftarrow t + 1$
  if $T \mod I_{target} == 0$ **then**
    Update the target network $\theta^- \leftarrow \theta$
  end if
  if $t \mod I_{AsyncUpdate} == 0$ or s is terminal **then**
    Perform asynchronous update of θ using $d\theta$.
    Clear gradients $d\theta \leftarrow 0$.
  end if
until $T > T_{max}$

Asynchronous one-step SARSA

- This is the same algorithm as Asynchronous one-step Q-learning except the target value used for Q(s, a) is different
- One-step SARSA uses $r + \gamma Q(s', a'; \theta^-)$ as the target value

n-step Q-learning

- So far we have been using one-step Q-learning
- One-step Q-learning updates the action value Q(s, a) towards r + $\gamma \max_{a'} Q(s', a'; \theta)$, which is the one-step return
- Obtaining a reward r only directly affects the value of the state-action pair (s, a) that led to the reward
- The values of the other state-action pairs are affected only indirectly through the updated value Q(s, a)
- This can make the learning process slow since many updates are required to propagate a reward to the relevant preceding states and actions
- Use n-step returns to propagate the rewards faster

n-step Returns

- In n-step Q-learning, Q(s, a) is updated towards the n-step return:
- $r_t + \gamma\, r_{t+1} + \ldots + \gamma^{n-1}\, r_{t+n-1} + \max_a \gamma^n\, Q(s_{t+n}, a)$
- This results in a single reward directly affecting the values of n preceding state-action pairs
- This makes the process of propagating rewards to relevant state-action pairs potentially much more efficient
- One-step update for the last state, two-step update for the second last state, and so on until n-step update for the nth last state
- Accumulated updates are applied in a single step

Asynchronous n-step Q-learning

Algorithm S2 Asynchronous n-step Q-learning - pseudocode for each actor-learner thread.

// Assume global shared parameter vector θ.
// Assume global shared target parameter vector θ^-.
// Assume global shared counter $T = 0$.
Initialize thread step counter $t \leftarrow 1$
Initialize target network parameters $\theta^- \leftarrow \theta$
Initialize thread-specific parameters $\theta' = \theta$
Initialize network gradients $d\theta \leftarrow 0$
repeat
  Clear gradients $d\theta \leftarrow 0$
  Synchronize thread-specific parameters $\theta' = \theta$
  $t_{start} = t$
  Get state s_t
  repeat
    Take action a_t according to the ϵ-greedy policy based on $Q(s_t, a; \theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ \max_a Q(s_t, a; \theta^-) & \text{for non-terminal } s_t \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \frac{\partial \left(R - Q(s_i, a_i; \theta')\right)^2}{\partial \theta'}$
  end for
  Perform asynchronous update of θ using $d\theta$.
  if $T \mod I_{target} == 0$ **then**
    $\theta^- \leftarrow \theta$
  end if
until $T > T_{max}$

A3C

- Maintains a policy $\pi(a_t | s_t; \theta)$ and an estimate of the value function $V(s_t; \theta_v)$
- n-step Actor-Critic method
- As with value-based methods, use parallel actor-learners and accumulate updates to improve training stability
- In practice, the policy approximation and the value function share some parameters
- Use a neural network that has one softmax output for the policy $\pi(a_t | s_t; \theta)$ and one linear output for the value function $V(s_t; \theta_v)$ with all non-output parameters shared

Advantage Definition

- For a one-step actor-critic method with a learned policy and a learned value function, the advantage is defined as follows:
- $A(a_t, s_t) = Q(a_t, s_t) - V(s_t)$
- The quantity $r_t - V(s_t; \theta_v)$ is an estimate of the advantage
- This estimate is used to scale the policy gradient in an actor-critic method
- The advantage estimate for a n-step actor-critic method is shown below:

$$\sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_v) - V(s_t; \theta_v)$$

A3C Algorithm

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$
// Assume thread-specific parameter vectors θ' *and* θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Summary

- Get faster training because of parallelism
- Can use on-policy reinforcement learning methods
- Diversity in exploration can lead to better performance than synchronous methods
- In practice, the on-policy A3C algorithm appears to be the best performing asynchronous reinforcement learning method in terms of performance and training speed

A SMALL RECAP

Overview

- Introduction to Reinforcement Learning
- Finite Markov Decision Processes
- Temporal-Difference Learning (SARSA, Q-learning, Deep Q-Networks)
- Policy Gradient Methods (Finite Difference Policy Gradient, REINFORCE, Actor-Critic)
- Asynchronous Reinforcement Learning

What is Reinforcement Learning?

- Learning from interaction with an environment to achieve some long-term goal that is related to the state of the environment
- The goal is defined by reward signal, which must be maximised
- Agent must be able to partially/fully sense the environment state and take actions to influence the environment state
- The state is typically described with a feature-vector

Exploration versus Exploitation

- We want a reinforcement learning agent to earn lots of reward
- The agent must prefer past actions that have been found to be effective at producing reward
- The agent must exploit what it already knows to obtain reward
- The agent must select untested actions to discover reward-producing actions
- The agent must explore actions to make better action selections in the future
- Trade-off between exploration and exploitation

Reinforcement Learning Systems

- Reinforcement learning systems have 4 main elements:
  - *Policy*
  - *Reward signal*
  - *Value function*
  - *Optional model of the environment*

Policy

- A policy is a mapping from the perceived states of the environment to actions to be taken when in those states
- A reinforcement learning agent uses a policy to select actions given the current environment state

Reward Signal

- The reward signal defines the goal
- On each time step, the environment sends a single number called the reward to the reinforcement learning agent
- The agent's objective is to maximise the total reward that it receives over the long run
- The reward signal is used to alter the policy

Value Function (1)

- The reward signal indicates what is good in the short run while the value function indicates what is good in the long run
- The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting in that state
- Compute the value using the states that are likely to follow the current state and the rewards available in those states
- Future rewards may be time-discounted with a factor in the interval [0, 1]

Value Function (2)

- Use the values to make and evaluate decisions
- Action choices are made based on value judgements
- Prefer actions that bring about states of highest value instead of highest reward
- Rewards are given directly by the environment
- Values must continually be re-estimated from the sequence of observations that an agent makes over its lifetime

Model-free versus Model-based

- A model of the environment allows inferences to be made about how the environment will behave
- Example: Given a state and an action to be taken while in that state, the model could predict the next state and the next reward
- Models are used for planning, which means deciding on a course of action by considering possible future situations before they are experienced
- Model-based methods use models and planning. Think of this as modelling the dynamics $p(s' \mid s, a)$
- Model-free methods learn exclusively from trial-and-error (i.e. no modelling of the environment)
- This presentation focuses on model-free methods

On-policy versus Off-policy

- An on-policy agent learns only about the policy that it is executing
- An off-policy agent learns about a policy or policies different from the one that it is executing

Credit Assignment Problem

- Given a sequence of states and actions, and the final sum of time-discounted future rewards, how do we infer which actions were effective at producing lots of reward and which actions were not effective?
- How do we assign credit for the observed rewards given a sequence of actions over time?
- Every reinforcement learning algorithm must address this problem

Reward Design

- We need rewards to guide the agent to achieve its goal
- Option 1: Hand-designed reward functions
- This is a black art
- Option 2: Learn rewards from demonstrations
- Instead of having a human expert tune a system to achieve the desired behaviour, the expert can demonstrate desired behaviour and the robot can tune itself to match the demonstration

What is Deep Reinforcement Learning?

- Deep reinforcement learning is standard reinforcement learning where a deep neural network is used to approximate either a policy or a value function
- Deep neural networks require lots of real/simulated interaction with the environment to learn
- Lots of trials/interactions is possible in simulated environments
- We can easily parallelise the trials/interaction in simulated environments
- We cannot do this with robotics (no simulations) because action execution takes time, accidents/failures are expensive and there are safety concerns

FINITE MARKOV DECISION PROCESSES

Chapter 3 – Reinforcement Learning: An Introduction

Markov Decision Process (MDP)

- Set of states S
- Set of actions A
- State transition probabilities p(s' | s, a). This is the probability distribution over the state space given we take action a in state s
- Discount factor γ in [0, 1]
- Reward function R: S x A -> set of real numbers
- For simplicity, assume discrete rewards
- Finite MDP if both S and A are finite

Time Discounting

- The undiscounted formulation $\gamma = 0$ across episodes is appropriate for episodic tasks in which agent-environment interaction breaks into episodes (multiple trials to perform a task).
- Example: Playing Breakout (each run of the game is an episode)
- The discounted formulation $0 < \gamma <= 1$ is appropriate for continuing tasks, in which the interaction continues without limit
- Example: Vacuum cleaner robot
- This presentation focuses on episodic tasks

Agent-Environment Interaction (1)

- The agent and environment interact at each of a sequence of discrete time steps t = {0, 1, 2, 3, …, T} where T can be infinite
- At each time step t, the agent receives some representation of the environment state S_t in S and uses this to select an action A_t in the set $A(S_t)$ of available actions given that state
- One step later, the agent receives a numerical reward R_{t+1} and finds itself in a new state S_{t+1}

Agent-Environment Interaction (2)

- At each time step, the agent implements a stochastic policy or mapping from states to probabilities of selecting each possible action
- The policy is denoted π_t where $\pi_t(a \mid s)$ is the probability of taking action a when in state s
- A policy is a stochastic rule by which the agent selects actions as a function of states
- Reinforcement learning methods specify how the agent changes its policy using its experience

Action Selection

- At time t, the agent tries to select an action to maximise the sum G_t of discounted rewards received in the future

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$$

MDP Dynamics

- Given current state s and action a in that state, the probability of the next state s' and the next reward r is given by:

$$p(s', r|s, a) = \Pr(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a)$$

State Transition Probabilities

- Suppose the reward function is discrete and maps from S x A to W
- The state transition probability or probability of transitioning to state s' given current state s and action a in that state is given by:

$$p(s'|s,a) = \sum_{r \in W} p(s', r|s, a)$$

Expected Rewards

- The expected reward for a given state-action pair is given by:

$$r(s,a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_{r \in W} r \sum_{s' \in S} p(s', r|s, a)$$

State-Value Function (1)

- Value functions are defined with respect to particular policies because future rewards depend on the agent's actions
- Value functions give the expected return of a particular policy
- The value $v_\pi(s)$ of a state s under a policy π is the expected return when starting in state s and following the policy π from that state onwards

State-Value Function (2)

- The state-value function $v_\pi(s)$ for the policy π is given below. Note that the value of the terminal state (if any) is always zero.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

$$v_\pi(s) = \mathbb{E}_\pi\left[\sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}|S_t = s\right]$$

Action-Value Function

- The value $q_\pi(s, a)$ of taking action a in state s under a policy π is defined as the expected return starting from s, taking the action a and thereafter following policy π
- q_π is the action-value function for policy π

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

$$q_\pi(s, a) = \mathbb{E}_\pi\left[\sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right]$$

Bellman Equation (1)

- The equation expresses the relationship between the value of a state s and the values of its successor states
- The value of the next state must equal the discounted value of the expected next state, plus the reward expected along the way

Bellman Equation (2)

- The value of state s is the expected value of the sum of time-discounted rewards (starting at current state) given current state s
- This is expected value of r plus the sum of time-discounted rewards (starting at successor state) over all successor states s' and all next rewards r and over all possible actions a in current state s

$$\forall s \in S : v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$$

Optimality

- A policy is defined to be better than or equal to another policy if its expected return is greater than or equal to that of the other policy for all states
- There is always at least one optimal policy π_* that is better than or equal to all other policies
- All optimal policies share the same optimal state-value function v*, which gives the maximum expected return for any state s over all possible policies
- All optimal policies share the same optimal action-value function q*, which gives the maximum expected return for any state-action pair (s, a) over all possible policies

TEMPORAL-DIFFERENCE LEARNING

Chapter 6 – Reinforcement Learning: An Introduction
Playing Atari with Deep Reinforcement Learning
Asynchronous Methods for Deep Reinforcement Learning
David Silver's Tutorial on Deep Reinforcement Learning

What is TD learning?

- Temporal-Difference learning = TD learning
- The prediction problem is that of estimating the value function for a policy π
- The control problem is the problem of finding an optimal policy π_*
- Given some experience following a policy π, update estimate v of v_π for non-terminal states occurring in that experience
- Given current step t, TD methods wait until the next time step to update $V(S_t)$
- Learn from partial returns

Value-based Reinforcement Learning

- We want to estimate the optimal value $V^*(s)$ or action-value function $Q^*(s, a)$ using a function approximator $V(s; \theta)$ or $Q(s, a; \theta)$ with parameters θ
- This function approximator can be any parametric supervised machine learning model
- Recall that the optimal value is the maximum value achievable under any policy

Update Rule for TD(0)

- At time t + 1, TD methods immediately form a target R_{t+1} + γ $V(S_{t+1})$ and make a useful update with step size α using the observed reward R_{t+1} and the estimate $V(S_{t+1})$
- The update is the step size times the difference between the target output and the actual output

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

Update Rule Intuition

- The target output is a more accurate estimate of $V(S_t)$ given the reward R_{t+1} is known
- The actual output is our current estimate of $V(S_t)$
- We simply take one step with our current value function estimate to get a more accurate estimate of $V(S_t)$ and then perform an update to move $V(S_t)$ closer towards the more accurate estimate (i.e. temporal difference)

Tabular TD(0) Algorithm

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0, \forall s \in \mathcal{S}^+$)
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        $A \leftarrow$ action given by π for S
        Take action A, observe R, S'
        $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
        $S \leftarrow S'$
    until S is terminal

SARSA – On-policy TD Control

- SARSA = State-Action-Reward-State-Action
- Learn an action-value function instead of a state-value function
- q_π is the action-value function for policy π
- Q-values are the values $q_\pi(s, a)$ for s in S, a in A
- SARSA experiences are used to update Q-values
- Use TD methods for the prediction problem

SARSA Update Rule

- We want to estimate $q_\pi(s, a)$ for the current policy π, and for all states s and action a
- The update rule is similar to that for TD(0) but we transition from state-action pair to state-action pair, and learn the values of state-action pairs
- The update is performed after every transition from a non-terminal state S_t
- If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is zero
- The update rule uses $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

SARSA Algorithm

Sarsa: An on-policy TD control algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Q-learning – Off-policy TD Control

- Similar to SARSA but off-policy updates
- The learned action-value function Q directly approximates the optimal action-value function q_* independent of the policy being followed
- In update rule, choose action a that maximises Q given S_{t+1} and use the resulting Q-value (i.e. estimated value given by optimal action-value function) plus the observed reward as the target
- This method is off-policy because we do not have a fixed policy that maps from states to actions. This is why A_{t+1} is not used in the update rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a(Q(S_{t+1}, a)) - Q(S_t, A_t)]$$

One-step Q-learning Algorithm

Q-learning: An off-policy TD control algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose A from S using policy derived from Q (e.g., ϵ-greedy)
        Take action A, observe R, S'
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
        $S \leftarrow S'$
    until S is terminal

Epsilon-greedy Policy

- At each time step, the agent selects an action
- The agent follows the greedy strategy with probability 1 – epsilon
- The agent selects a random action with probability epsilon
- With Q-learning, the greedy strategy is the action a that maximises Q given S_{t+1}

Deep Q-Networks (DQN)

- Introduced deep reinforcement learning
- It is common to use a function approximator $Q(s, a; \theta)$ to approximate the action-value function in Q-learning
- Deep Q-Networks is Q-learning with a deep neural network function approximator called the Q-network
- Discrete and finite set of actions A
- Example: Breakout has 3 actions – move left, move right, no movement
- Uses epsilon-greedy policy to select actions

Q-Networks

- Core idea: We want the neural network to learn a non-linear hierarchy of features or feature representation that gives accurate Q-value estimates
- The neural network has a separate output unit for each possible action, which gives the Q-value estimate for that action given the input state
- The neural network is trained using mini-batch stochastic gradient updates and experience replay

Experience Replay

- The state is a sequence of actions and observations $s_t = x_1, a_1, x_2, \ldots, a_{t-1}, x_t$
- Store the agent's experiences at each time step $e_t = (s_t, a_t, r_t, s_{t+1})$ in a dataset $D = e_1, \ldots, e_n$ pooled over many episodes into a replay memory
- In practice, only store the last N experience tuples in the replay memory and sample uniformly from D when performing updates

State representation

- It is difficult to give the neural network a sequence of arbitrary length as input
- Use fixed length representation of sequence/history produced by a function $\varphi(s_t)$
- Example: The last 4 image frames in the sequence of Breakout gameplay

Q-Network Training

- Sample random mini-batch of experience tuples uniformly at random from D
- Similar to Q-learning update rule but:
  - *Use mini-batch stochastic gradient updates*
  - *The gradient of the loss function for a given iteration with respect to the parameter θ_i is the difference between the target value and the actual value is multiplied by the gradient of the Q function approximator $Q(s, a; \theta)$ with respect to that specific parameter*
- Use the gradient of the loss function to update the Q function approximator

Loss Function Gradient Derivation

network. We refer to a neural network function approximator with weights θ as a Q-network. A Q-network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right], \tag{2}$$

where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a\right]$ is the target for iteration i and $\rho(s,a)$ is a probability distribution over sequences s and actions a that we refer to as the *behaviour distribution*. The parameters from the previous iteration θ_{i-1} are held fixed when optimising the loss function $L_i(\theta_i)$. Note that the targets depend on the network weights; this is in contrast with the targets used for supervised learning, which are fixed before learning begins. Differentiating the loss function with respect to the weights we arrive at the following gradient,

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]. \tag{3}$$

Rather than computing the full expectations in the above gradient, it is often computationally expedient to optimise the loss function by stochastic gradient descent. If the weights are updated after every time-step, and the expectations are replaced by single samples from the behaviour distribution ρ and the emulator $\mathcal{E}$ respectively, then we arrive at the familiar *Q-learning* algorithm [26].

DQN Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for