Introduction to Reinforcement Learning

Zaiwen Wen

http://bicmr.pku.edu.cn/~wenzw/bigdata2019.html

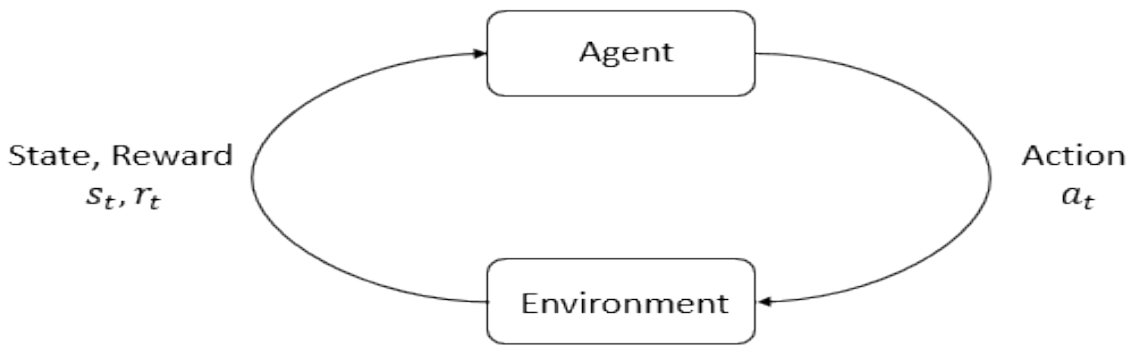
Acknowledgement: this slides is based on OpenAI Spinning Up

https://spinningup.openai.com/

Outline

Main characters of RL: agent and environment

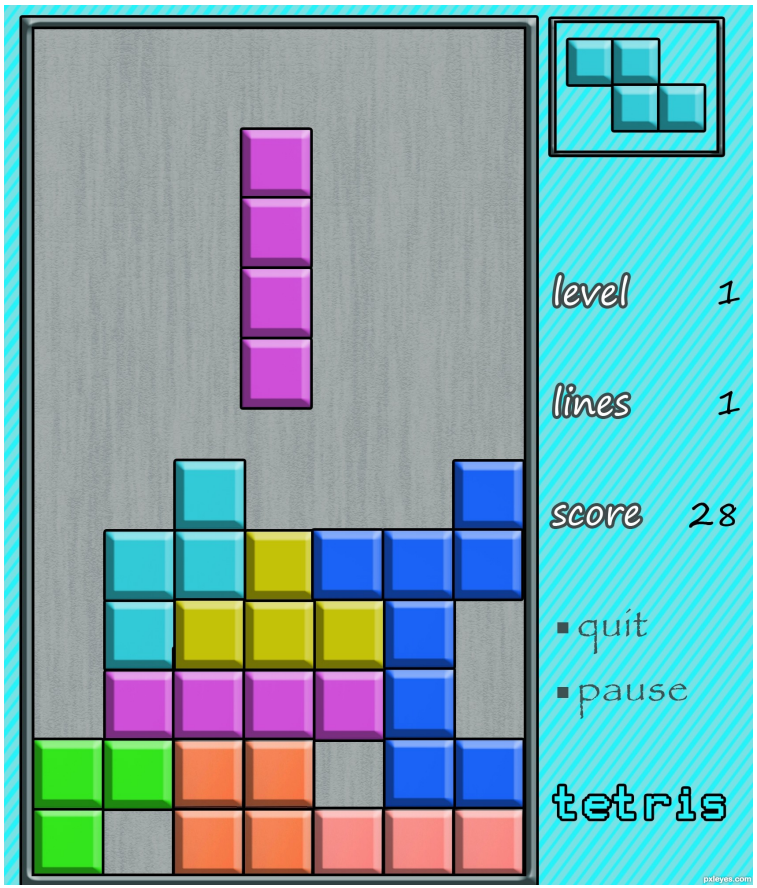
- **environment**: the world that the agent lives in and interacts with. At every step of interaction, the agent sees a (possibly partial) observation of the state of the world, and then decides on an action to take. The environment changes when the agent acts on it, but may also change on its own.
- **agent**: perceives a **reward** signal from the environment, a number that tells it how good or bad the current world state is. The goal of the agent is to maximize its cumulative reward, called **return**. Reinforcement learning methods are ways that the agent can learn behaviors to achieve its goal.



Terminology

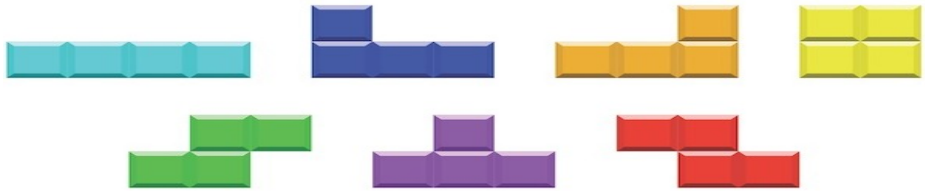
- states and observations,
- action spaces,
- policies,
- trajectories,
- different formulations of return,
- the RL optimization problem,
- and value functions.

Tetris



- Height: 12
- Width: 7
- Rotate and move the falling shape
- Gravity related to current height
- Score when eliminating an entire level
- Game over when reaching the ceiling

DP Model of Tetris



- State: The current board, the current falling tile, predictions of future tiles
- Termination state: when the tiles reach the ceiling, the game is over with no more future reward
- Action: Rotation and shift
- System Dynamics: The next board is deterministically determined by the current board and the player's placement of the current tile. The future tiles are generated randomly.
- Uncertainty: Randomness in future tiles
- Transitional cost g: If a level is cleared by the current action, score 1; otherwise score 0.
- Objective: Expectation of total score.

Interesting facts about Tetris

- First released in 1984 by Alexey Pajitnov from the Soviet Union
- Has been proved to be NP-complete.
- Game will be over with probability 1.
- For a 12×7 board, the number of possible states $\approx 2^{12 \times 7} \approx 10^{25}$
- Highest score achieved by human $\approx$ 1 million
- Highest score achieved by algorithm $\approx$ 35 million (average performance)

States and Observations

- **state**: s is a complete description of the state of the world. There is no information about the world which is hidden from the state. An **observation** o is a partial description of a state, which may omit information.
- In deep RL, we almost always represent states and observations by a "real-valued vector, matrix, or higher-order tensor". For instance, a visual observation could be represented by the RGB matrix of its pixel values; the state of a robot might be represented by its joint angles and velocities.
- When the agent is able to observe the complete state of the environment, we say that the environment is **fully observed**. When the agent can only see a partial observation, we say that the environment is **partially observed**.
- We often write that the action is conditioned on the state, when in practice, the action is conditioned on the observation because the agent does not have access to the state.

Action Spaces

- Different environments allow different kinds of actions. The set of all valid actions in a given environment is often called the **action space**. Some environments, like Atari and Go, have **discrete action spaces**, where only a finite number of moves are available to the agent. Other environments, like where the agent controls a robot in a physical world, have **continuous action spaces**. In continuous spaces, actions are real-valued vectors.
- This distinction has some quite-profound consequences for methods in deep RL. Some families of algorithms can only be directly applied in one case, and would have to be substantially reworked for the other.

Policies

- A **policy** is a rule used by an agent to decide what actions to take. It can be deterministic denoted by μ:

$$a_t = \mu(s_t),$$

  or it may be stochastic denoted by π:

$$a_t \sim \pi(\cdot|s_t).$$

- The policy is essentially the agent's brain. Often say "The policy is trying to maximize reward."

- **parameterized policies**: policies whose outputs are computable functions that depend on a set of parameters (eg the weights and biases of a neural network)

- We often denote the parameters of such a policy by θ or ϕ, and then write this as a subscript on the policy symbol to highlight the connection:

$$a_t = \mu_\theta(s_t), \quad a_t \sim \pi_\theta(\cdot|s_t).$$

Example: deterministic Policies

a code snippet for building a simple deterministic policy for a continuous action space in Tensorflow:

```
obs = tf.placeholder(shape=(None, obs_dim), dtype=tf.float32)
net = mlp(obs, hidden_dims=(64,64), activation=tf.tanh)
actions = tf.layers.dense(net, units=act_dim, activation=None)
```

where “mlp” is a function that stacks multiple “dense” layers on top of each other with the given sizes and activation.

Stochastic Policies

The two most common kinds of stochastic policies in deep RL are **categorical policies** and **diagonal Gaussian policies**:

- **Categorical** policies can be used in discrete action spaces
- **diagonal Gaussian** policies are used in continuous action spaces.

Two key computations are centrally important for using and training stochastic policies:

- sampling actions from the policy,
- and computing log likelihoods of particular actions, $\log \pi_\theta(a|s)$.

Categorical Policies

- A categorical policy is like a classifier over discrete actions. You build the neural network for a categorical policy the same way you would for a classifier: the input is the observation, followed by some number of layers (possibly convolutional or densely-connected, depending on the kind of input), and then you have one final linear layer that gives you logits for each action, followed by a 'softmax' to convert the logits into probabilities.
- **Sampling**: Given the probabilities for each action, frameworks like Tensorflow have built-in tools for sampling. For example, see the 'tf.distributions.Categorical' documentation, or 'tf.multinomial'.
- **Log-Likelihood**: Denote the last layer of probabilities as $P_\theta(s)$. It is a vector with how many entries as there are actions, so we can treat the actions as indices for the vector. The log likelihood for an action a can then be obtained by indexing into the vector: $\log \pi_\theta(a|s) = \log \left[P_\theta(s)\right]_a$.

Diagonal Gaussian Policies

- A multivariate Gaussian distribution of a mean vector μ, and a covariance matrix Σ. A diagonal Gaussian distribution: the covariance matrix only has entries on the diagonal.
- A diagonal Gaussian policy always has a neural network that maps from observations to mean actions $\mu_\theta(s)$. There are two different ways that the covariance matrix is typically represented.
- **The first way**: There is a single vector of log standard deviations $\log \sigma$, which is **not** a function of state: the $\log \sigma$ are standalone parameters.
- **the second way**: There is a neural network that maps from states to log standard deviations, $\log \sigma_\theta(s)$. It may optionally share some layers with the mean network.
- Note that in both cases we output log standard deviations instead of standard deviations directly.

Diagonal Gaussian Policies

- **Sampling**: Given the mean action $\mu_\theta(s)$ and standard deviation $\sigma_\theta(s)$, and a vector z of noise from a spherical Gaussian ($z \sim \mathcal{N}(0, I)$), an action sample can be computed with

$$a = \mu_\theta(s) + \sigma_\theta(s) \odot z,$$

  where $\odot$ denotes the elementwise product of two vectors. Standard frameworks have built-in ways to compute the noise vectors, such as 'tf.random_normal'. Alternatively, you can just provide the mean and standard deviation directly to a 'tf.distributions.Normal' object and use that to sample.

- **Log-Likelihood**: The log-likelihood of a k-dimensional action a, for a diagonal Gaussian with mean $\mu = \mu_\theta(s)$ and standard deviation $\sigma = \sigma_\theta(s)$, is given by

$$\log \pi_\theta(a|s) = -\frac{1}{2}\left(\sum_{i=1}^{k}\left(\frac{(a_i - \mu_i)^2}{\sigma_i^2} + 2\log\sigma_i\right) + k\log 2\pi\right)$$

Trajectories

- A trajectory τ is a sequence of states and actions in the world,

$$\tau = (s_0, a_0, s_1, a_1, ...).$$

- The very first state s_0, is randomly sampled from the **start-state distribution** ρ_0:

$$s_0 \sim \rho_0(\cdot).$$

- State transitions (what happens between the state at time t, s_t, and the state at $t+1$, s_{t+1}), are governed by the natural laws of the environment, and depend on only the most recent action a_t. They can be either deterministic,

$$s_{t+1} = f(s_t, a_t)$$

  or stochastic,

$$s_{t+1} \sim P(\cdot|s_t, a_t).$$

- Actions come from an agent according to its policy.
- Trajectories are also frequently called **episodes** or **rollouts**.

Reward and Return

- The reward function R depends on the current state of the world, the action just taken, and the next state of the world:

$$r_t = R(s_t, a_t, s_{t+1}).$$

  Simplification: $r_t = R(s_t)$, or state-action pair $r_t = R(s_t, a_t)$.

- Goal: maximize some "cumulative" reward over a trajectory
- **finite-horizon undiscounted return**: the sum of rewards obtained in a fixed window of steps:

$$R(\tau) = \sum_{t=0}^{T} r_t$$

- **infinite-horizon discounted return**: the sum of all rewards ever obtained by the agent, but discounted by how far off in the future they're obtained with a discount factor $\gamma \in (0, 1)$:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t.$$

Reward and Return

- Why would we ever want a discount factor, though? Don't we just want to get all rewards? We do, but the discount factor is both intuitively appealing and mathematically convenient. On an intuitive level: cash now is better than cash later. Mathematically: an infinite-horizon sum of rewards 'may not converge' to a finite value, and is hard to deal with in equations. But with a discount factor and under reasonable conditions, the infinite sum converges.
- While the line between these two formulations of return are quite stark in RL formalism, deep RL practice tends to blur the line a fair bit—for instance, we frequently set up algorithms to optimize the undiscounted return, but use discount factors in estimating **value functions**.

The RL Problem

- The goal in RL is to select a policy which maximizes **expected return** when the agent acts according to it.
- Suppose that both the environment transitions and the policy are stochastic. The probability of a T-step trajectory is:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t).$$

- The expected return (for whichever measure), denoted by $J(\pi)$, is then:

$$J(\pi) = \int_{\tau} P(\tau|\pi)R(\tau) = \underset{\tau\sim\pi}{\mathbf{E}}\left[R(\tau)\right].$$

- The central optimization problem in RL can then be expressed by

$$\pi^* = \arg\max_{\pi} J(\pi),$$

with π^* being the **optimal policy**.

Value Functions

It's often useful to know the **value** of a state, or state-action pair. By value, we mean the expected return if you start in that state or state-action pair, and then act according to a particular policy forever after. **Value functions** are used, one way or another, in almost every RL algorithm.

- The **On-Policy Value Function** $V^{\pi}(s)$, which gives the expected return if you start in state s and always act according to policy π:

$$V^{\pi}(s) = \underset{\tau \sim \pi}{\mathbf{E}} \left[R(\tau) \,|s_0 = s \right]$$

- The **On-Policy Action-Value Function** $Q^{\pi}(s,a)$, which gives the expected return if you start in state s, take an arbitrary action a (which may not have come from the policy), and then forever after act according to policy π:

$$Q^{\pi}(s,a) = \underset{\tau \sim \pi}{\mathbf{E}} \left[R(\tau) \,|s_0 = s, a_0 = a \right]$$

Value Functions

- The **Optimal Value Function** $V^*(s)$, which gives the expected return if you start in state s and always act according to the *optimal* policy in the environment:

$$V^*(s) = \max_{\pi} \underset{\tau \sim \pi}{\mathbf{E}} \left[R(\tau) \,|s_0 = s \right]$$

- The **Optimal Action-Value Function**, $Q^*(s,a)$, which gives the expected return if you start in state s, take an arbitrary action a, and then forever after act according to the *optimal* policy in the environment:

$$Q^*(s,a) = \max_{\pi} \underset{\tau \sim \pi}{\mathbf{E}} \left[R(\tau) \,|s_0 = s, a_0 = a \right]$$

Value Functions

- When we talk about value functions, if we do not make reference to time-dependence, we only mean expected **infinite-horizon discounted return**. Value functions for finite-horizon undiscounted return would need to accept time as an argument. Can you think about why? Hint: what happens when time's up?
- There are two key connections between the value function and the action-value function that come up pretty often:

$$V^{\pi}(s) = \mathop{\mathbf{E}}_{a\sim\pi}\left[Q^{\pi}(s,a)\right],$$

and

$$V^{*}(s) = \max_{a} Q^{*}(s,a).$$

These relations follow pretty directly from the definitions just given: can you prove them?

The Optimal Q-Function and the Optimal Action

- There is an important connection between the optimal action-value function $Q^*(s, a)$ and the action selected by the optimal policy. By definition, $Q^*(s, a)$ gives the expected return for starting in state s, taking (arbitrary) action a, and then acting according to the optimal policy forever after.
- The optimal policy in s will select whichever action maximizes the expected return from starting in s. As a result, if we have Q^*, we can directly obtain the optimal action, $a^*(s)$, via

$$a^*(s) = \arg\max_a Q^*(s, a).$$

- Note: there may be multiple actions which maximize $Q^*(s, a)$, in which case, all of them are optimal, and the optimal policy may randomly select any of them. But there is always an optimal policy which deterministically selects an action.

Bellman Equations

- All four of the value functions obey special self-consistency equations called **Bellman equations**. The basic idea is: *The value of your starting point is the reward you expect to get from being there, plus the value of wherever you land next.*
- The Bellman equations for the on-policy value functions are

$$V^\pi(s) = \mathop{\mathbf{E}}_{\substack{a\sim\pi \\ s'\sim P}} \left[r(s,a) + \gamma V^\pi(s') \right],$$

$$Q^\pi(s,a) = \mathop{\mathbf{E}}_{s'\sim P} \left[r(s,a) + \gamma \mathop{\mathbf{E}}_{a'\sim\pi} \left[Q^\pi(s',a') \right] \right],$$

where $s' \sim P$ is shorthand for $s' \sim P(\cdot|s,a)$, indicating that the next state s' is sampled from the environment's transition rules; $a \sim \pi$ is shorthand for $a \sim \pi(\cdot|s)$; and $a' \sim \pi$ is shorthand for $a' \sim \pi(\cdot|s')$.

Bellman Equations

- The Bellman equations for the optimal value functions are

$$V^*(s) = \max_a \mathop{\mathbf{E}}_{s' \sim P} \left[r(s,a) + \gamma V^*(s') \right],$$

$$Q^*(s,a) = \mathop{\mathbf{E}}_{s' \sim P} \left[r(s,a) + \gamma \max_{a'} Q^*(s',a') \right].$$

  The crucial difference between the Bellman equations for the on-policy value functions and the optimal value functions, is the absence or presence of the $\max$ over actions. Its inclusion reflects the fact that whenever the agent gets to choose its action, in order to act optimally, it has to pick whichever action leads to the highest value.

- The term "Bellman backup" comes up quite frequently in the RL literature. The Bellman backup for a state, or state-action pair, is the right-hand side of the Bellman equation: the reward-plus-next-value.

Advantage Functions

- Sometimes in RL, we don't need to describe how good an action is in an absolute sense, but only how much better it is than others on average. That is to say, we want to know the relative **advantage** of that action. We make this concept precise with the **advantage function**.
- The advantage function $A^\pi(s,a)$ corresponding to a policy π describes how much better it is to take a specific action a in state s, over randomly selecting an action according to $\pi(\cdot|s)$, assuming you act according to π forever after. Mathematically, the advantage function is defined by

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s).$$

- We'll discuss this more later, but the advantage function is crucially important to policy gradient methods.

Outline

Platform

- Gym: a toolkit for developing and comparing reinforcement learning algorithms. Support Atari and Mujoco.
- Universe: measuring and training an AI across the world's supply of games, websites and other applications
- Deepmind Lab: a fully 3D game-like platform tailored for agent-based AI research
- ViZDoom: allows developing AI bots that play Doom using only the visual information

Platform

- Rllab: mainly supports for TRPO, VPG, CEM, NPG
- Baselines: supports for TRPO, PPO, DQN, A2C...
- Github
- Implement your algorithms through these packages

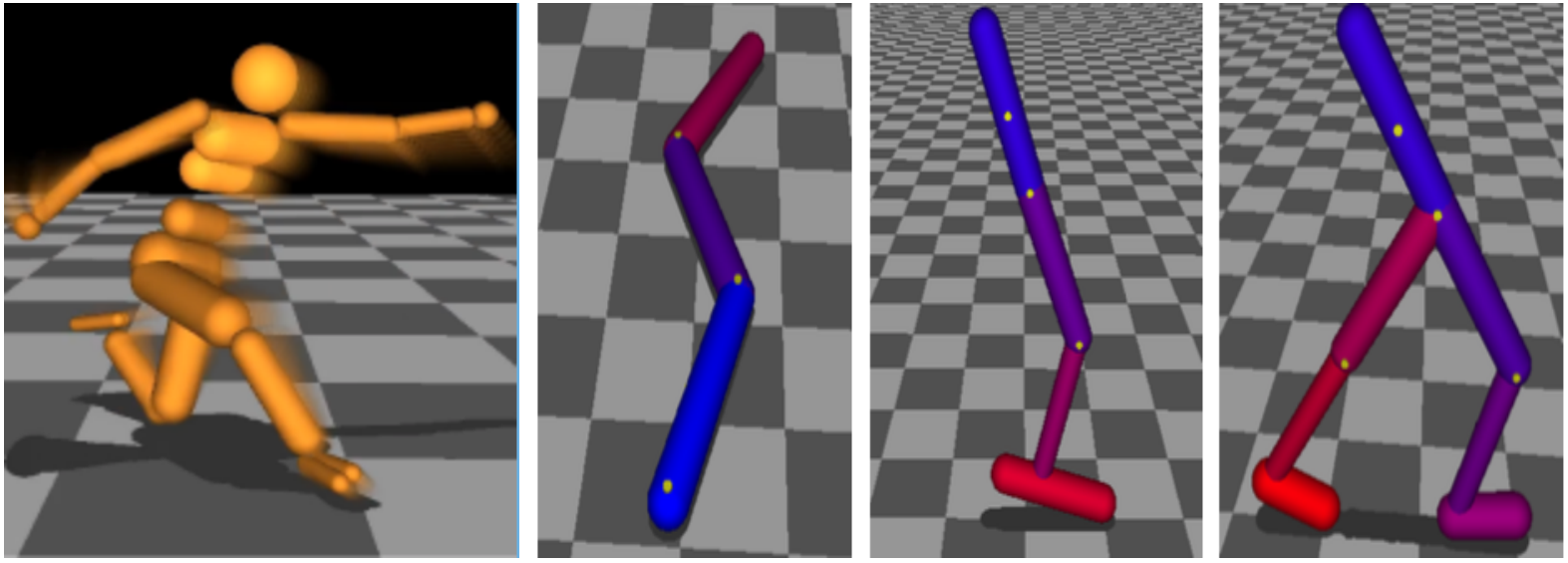
Gym

- A simple implementation of sampling a single path with gym

```
1  import gym, numpy
2  env = gym.make("MsPacman-v0")
3  ob = env.reset()
4  env.render()
5  rew = []
6  while True:
7    action = pi.act(ob)
8    ob, reward, done, info = env.step(action)
9    rew.append(reward)
10   if done:
11       ret = numpy.sum(rew)
12       break
```

Environments

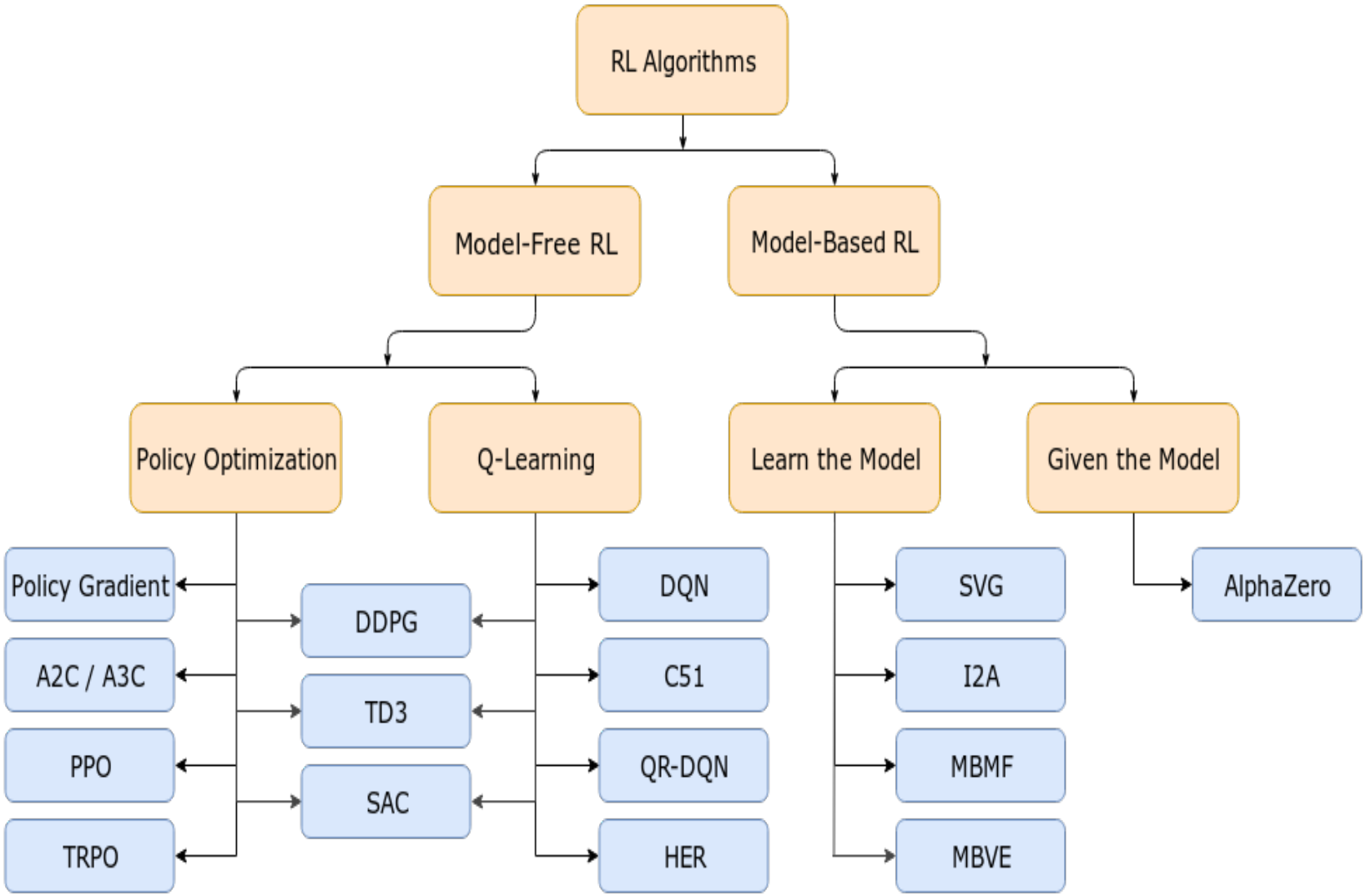
- Mujoco, continuous tasks
- A physics engine for detailed, efficient rigid body simulations with contacts
- Swimmer, Hopper, Walker, Reacher,...
- Gaussian distribution



Environments

- Atari 2600, discrete action space
- Categorial distribution

A Taxonomy of RL Algorithms



A Taxonomy of RL Algorithms

We'll start this section with a disclaimer: it's really quite hard to draw an accurate, all-encompassing taxonomy of algorithms in the modern RL space, because the modularity of algorithms is not well-represented by a tree structure. Also, to make something that fits on a page and is reasonably digestible in an introduction essay, we have to omit quite a bit of more advanced material (exploration, transfer learning, meta learning, etc). That said, our goals here are

- to highlight the most foundational design choices in deep RL algorithms about what to learn and how to learn it,
- to expose the trade-offs in those choices,
- and to place a few prominent modern algorithms into context with respect to those choices.

Model-Free vs Model-Based RL

- One of the most important branching points in an RL algorithm is the question of **whether the agent has access to (or learns) a model of the environment**. By a model of the environment, we mean a function which predicts state transitions and rewards.
- upside: **it allows the agent to plan** by thinking ahead, seeing what would happen for a range of possible choices, and explicitly deciding between its options. Agents can then distill the results from planning ahead into a learned policy. A particularly famous example of this approach is 'AlphaZero'. When this works, it can result in a substantial improvement in sample efficiency over methods that don't have a model.

Model-Free vs Model-Based RL

- downside: **a ground-truth model of the environment is usually not available to the agent.** If an agent wants to use a model in this case, it has to learn the model purely from experience, which creates several challenges. The biggest challenge is that bias in the model can be exploited by the agent, resulting in an agent which performs well with respect to the learned model, but behaves sub-optimally (or super terribly) in the real environment. Model-learning is fundamentally hard, so even intense effort—being willing to throw lots of time and compute at it—can fail to pay off.
- Algorithms which use a model are called **model-based** methods, and those that don't are called **model-free**. While model-free methods forego the potential gains in sample efficiency from using a model, they tend to be easier to implement and tune. As of the time of this writing, model-free methods are more popular and have been more extensively developed and tested than model-based methods.

What to Learn

Another critical branching point in an RL algorithm is the question of **what to learn**. The list of usual suspects includes

- policies, either stochastic or deterministic,
- action-value functions (Q-functions),
- value functions,
- and/or environment models.
-

Model-Free RL: Policy Optimization

- Represent a policy explicitly as $\pi_\theta(a|s)$. They optimize the parameters θ either directly by gradient ascent on the performance objective $J(\pi_\theta)$, or indirectly, by maximizing local approximations of $J(\pi_\theta)$. This optimization is almost always performed **on-policy**, which means that each update only uses data collected while acting according to the most recent version of the policy. Policy optimization also usually involves learning an approximator $V_\phi(s)$ for the on-policy value function $V^\pi(s)$, which gets used in figuring out how to update the policy.
- A2C / A3C performs gradient ascent to directly maximize performance,
- PPO updates indirectly maximize performance, by instead maximizing a *surrogate objective* function which gives a conservative estimate for how much $J(\pi_\theta)$ will change as a result of the update.

Model-Free RL: Q-Learning

- Methods in this family learn an approximator $Q_\theta(s, a)$ for the optimal action-value function, $Q^*(s, a)$. Typically they use an objective function based on the 'Bellman equation'. This optimization is almost always performed off-policy, which means that each update can use data collected at any point during training, regardless of how the agent was choosing to explore the environment when the data was obtained. The corresponding policy is obtained via the connection between Q^* and π^*: the actions taken by the Q-learning agent are given by

$$a(s) = \arg\max_a Q_\theta(s, a).$$

- DQN: a classic which substantially launched the field of deep RL,
- C51: a variant that learns a distribution over return whose expectation is Q^*.

Trade-offs Between Policy Optimization and Q-Learning

- The primary strength of policy optimization methods is that they are principled, in the sense that *you directly optimize for the thing you want*. This tends to make them stable and reliable. By contrast, Q-learning methods only indirectly optimize for agent performance, by training Q_θ to satisfy a self-consistency equation. There are many failure modes for this kind of learning, so it tends to be less stable. But, Q-learning methods gain the advantage of being substantially more sample efficient when they do work, because they can reuse data more effectively than policy optimization techniques.

Interpolating Between Policy Optimization and Q-Learning

- Serendipitously, policy optimization and Q-learning are not incompatible (and under some circumstances, it turns out, 'equivalent'), and there exist a range of algorithms that live in between the two extremes. Algorithms that live on this spectrum are able to carefully trade-off between the strengths and weaknesses of either side. Examples include
- DDPG: an algorithm which concurrently learns a deterministic policy and a Q-function by using each to improve the other,
- SAC: a variant which uses stochastic policies, entropy regularization, and a few other tricks to stabilize learning and score higher than DDPG on standard benchmarks.

Model-Based RL: Background: Pure Planning

- The most basic approach never explicitly represents the policy, and instead, uses pure planning techniques like 'model-predictive control' (MPC) to select actions. In MPC, each time the agent observes the environment, it computes a plan which is optimal with respect to the model, where the plan describes all actions to take over some fixed window of time after the present. (Future rewards beyond the horizon may be considered by the planning algorithm through the use of a learned value function.) The agent then executes the first action of the plan, and immediately discards the rest of it. It computes a new plan each time it prepares to interact with the environment, to avoid using an action from a plan with a shorter-than-desired planning horizon.
- The 'MBMF' work explores MPC with learned environment models on some standard benchmark tasks for deep RL.

Model-Based RL: Expert Iteration

- A straightforward follow-on to pure planning involves using and learning an explicit representation of the policy, $\pi_\theta(a|s)$. The agent uses a planning algorithm (like Monte Carlo Tree Search) in the model, generating candidate actions for the plan by sampling from its current policy. The planning algorithm produces an action which is better than what the policy alone would have produced, hence it is an "expert" relative to the policy. The policy is afterwards updated to produce an action more like the planning algorithm's output.
- The 'ExIt' algorithm uses this approach to train deep neural networks to play Hex.
- 'AlphaZero' is another example of this approach.

Model-Based RL: Data Augmentation for Model-Free Methods

- Use a model-free RL algorithm to train a policy or Q-function, but either 1) augment real experiences with fictitious ones in updating the agent, or 2) use only fictitous experience for updating the agent.
- See ‘MBVE’ for an example of augmenting real experiences with fictitious ones.
- See ‘World Models’ for an example of using purely fictitious experience to train the agent, which they call "training in the dream."

Model-Based RL: Embedding Planning Loops into Policies

- Another approach embeds the planning procedure directly into a policy as a subroutine—so that complete plans become side information for the policy—while training the output of the policy with any standard model-free algorithm. The key concept is that in this framework, the policy can learn to choose how and when to use the plans. This makes model bias less of a problem, because if the model is bad for planning in some states, the policy can simply learn to ignore it.
- See ‘I2A’ for an example of agents being endowed with this style of imagination.

Outline

Introduction to Policy Optimization

- In this section, we'll discuss the mathematical foundations of policy optimization algorithms, and connect the material to sample code. We will cover three key results in the theory of **policy gradients**:
- 'the simplest equation' describing the gradient of policy performance with respect to policy parameters,
- a rule which allows us to 'drop useless terms' from that expression,
- and a rule which allows us to 'add useful terms' to that expression.
- In the end, we'll tie those results together and describe the advantage-based expression for the policy gradient—the version we use in our 'Vanilla Policy Gradient' implementation.

Deriving the Simplest Policy Gradient

- Consider the case of a stochastic, parameterized policy, π_θ. We aim to maximize the expected return $J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathbf{E}}[R(\tau)]$. For the purposes of this derivation, we'll take $R(\tau)$ to give the *finite-horizon undiscounted return*, but the derivation for the infinite-horizon discounted return setting is almost identical.
- We would like to optimize the policy by gradient ascent, eg

$$\theta_{k+1} = \theta_k + \alpha \left. \nabla_\theta J(\pi_\theta) \right|_{\theta_k}.$$

- The gradient of policy performance, $\nabla_\theta J(\pi_\theta)$, is called the **policy gradient**, and algorithms that optimize the policy this way are called **policy gradient algorithms**. (Examples include Vanilla Policy Gradient and TRPO. PPO is often referred to as a policy gradient algorithm, though this is slightly inaccurate.)

Deriving the Simplest Policy Gradient

- To actually use this algorithm, we need an expression for the policy gradient which we can numerically compute. This involves two steps: 1) deriving the analytical gradient of policy performance, which turns out to have the form of an expected value, and then 2) forming a sample estimate of that expected value, which can be computed with data from a finite number of agent-environment interaction steps.
- we'll find the simplest form of that expression. In later subsections, we'll show how to improve on the simplest form to get the version we actually use in standard policy gradient implementations.

Analytical Gradient

- **Probability of a Trajectory**: The probability of a trajectory $\tau = (s_0, a_0, ..., s_{T+1})$ given that actions come from π_θ is

$$P(\tau|\theta) = \rho_0(s_0) \prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t).$$

- **The Log-Derivative Trick**: The log-derivative trick is based on a simple rule from calculus: the derivative of $\log x$ with respect to x is $1/x$. When rearranged and combined with chain rule, we get:

$$\nabla_\theta P(\tau|\theta) = P(\tau|\theta)\nabla_\theta \log P(\tau|\theta).$$

- **Log-Probability of a Trajectory**: The log-prob of a trajectory is just

$$\log P(\tau|\theta) = \log \rho_0(s_0) + \sum_{t=0}^{T} \Big(\log P(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t) \Big).$$

Analytical Gradient

- **Gradients of Environment Functions**: The environment has no dependence on θ, so gradients of $\rho_0(s_0)$, $P(s_{t+1}|s_t, a_t)$, and $R(\tau)$ are zero.
- **Grad-Log-Prob of a Trajectory**: The gradient of the log-prob of a trajectory is thus

$$\begin{aligned}\nabla_\theta \log P(\tau|\theta) &= \cancel{\nabla_\theta \log \rho_0(s_0)} + \sum_{t=0}^{T} \left(\cancel{\nabla_\theta \log P(s_{t+1}|s_t, a_t)} + \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \\ &= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t).\end{aligned}$$

Derivation for Basic Policy Gradient

$$
\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathop{\mathbf{E}}_{\tau\sim\pi_\theta}[R(\tau)] \\
&= \nabla_\theta \int_\tau P(\tau|\theta)R(\tau) && \text{Expand expectation} \\
&= \int_\tau \nabla_\theta P(\tau|\theta)R(\tau) && \text{Bring gradient under integral} \\
&= \int_\tau P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)R(\tau) && \text{Log-derivative trick} \\
&= \mathop{\mathbf{E}}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau|\theta)R(\tau)] && \text{Return to expectation form} \\
\therefore \nabla_\theta J(\pi_\theta) &= \mathop{\mathbf{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)\right] && \text{Expression for grad-log-prob}
\end{aligned}
$$

Derivation for Basic Policy Gradient

- This is an expectation, which means that we can estimate it with a sample mean. If we collect a set of trajectories $\mathcal{D} = \{\tau_i\}_{i=1,\dots,N}$ where each trajectory is obtained by letting the agent act in the environment using the policy π_θ, the policy gradient can be estimated with

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau),$$

  where $|\mathcal{D}|$ is the number of trajectories in $\mathcal{D}$ (here, N).

- This last expression is the simplest version of the computable expression we desired. Assuming that we have represented our policy in a way which allows us to calculate $\nabla_\theta \log \pi_\theta(a|s)$, and if we are able to run the policy in the environment to collect the trajectory dataset, we can compute the policy gradient and take an update step.

Expected Grad-Log-Prob Lemma

EGLP Lemma: Suppose that P_θ is a parameterized probability distribution over a random variable, x. Then:

$$\mathop{\mathbf{E}}_{x \sim P_\theta} \left[\nabla_\theta \log P_\theta(x) \right] = 0.$$

Proof: Recall that all probability distributions are normalized:

$$\int_x P_\theta(x) = 1.$$

Take the gradient of both sides of the normalization condition:

$$\nabla_\theta \int_x P_\theta(x) = \nabla_\theta 1 = 0.$$

Use the log derivative trick to get:

$$0 = \nabla_\theta \int_x P_\theta(x) = \int_x \nabla_\theta P_\theta(x) = \int_x P_\theta(x) \nabla_\theta \log P_\theta(x)$$

$$\therefore 0 = \mathop{\mathbf{E}}_{x \sim P_\theta} \left[\nabla_\theta \log P_\theta(x) \right].$$

Don't Let the Past Distract You

- Examine our most recent expression for the policy gradient:

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathbf{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right].$$

  Taking a step with this gradient pushes up the log-probabilities of each action in proportion to $R(\tau)$, the sum of all rewards ever obtained. But this doesn't make much sense.

- Agents should really only reinforce actions on the basis of their consequences. Rewards obtained before taking an action have no bearing on how good that action was: only rewards that come after.

Don't Let the Past Distract You

- It turns out that this intuition shows up in the math, and we can show that the policy gradient can also be expressed by

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbf{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t'=t}^{T}R(s_{t'},a_{t'},s_{t'+1})\right].$$

  In this form, actions are only reinforced based on rewards obtained after they are taken.

- We'll call this form the "reward-to-go policy gradient", because the sum of rewards after a point in a trajectory,

$$\hat{R}_t \doteq \sum_{t'=t}^{T} R(s_{t'},a_{t'},s_{t'+1}),$$

  is called the **reward-to-go** from that point, and this policy gradient expression depends on the reward-to-go from state-action pairs.

Baselines in Policy Gradients

- An immediate consequence of the EGLP lemma is that for any function b which only depends on state,

$$\mathop{\mathbf{E}}_{a_t \sim \pi_\theta} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t) \right] = 0.$$

- This allows us to add or subtract any number of terms like this from our expression for the policy gradient, without changing it in expectation:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbf{E}}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t) \right) \right].$$

- Any function b used in this way is called a **baseline**.

Baselines in Policy Gradients

- The most common choice of baseline is the 'on-policy value function' $V^{\pi}(s_t)$. Recall that this is the average return an agent gets if it starts in state s_t and then acts according to policy π for the rest of its life.
- Empirically, the choice $b(s_t) = V^{\pi}(s_t)$ has the desirable effect of reducing variance in the sample estimate for the policy gradient. This results in faster and more stable policy learning. It is also appealing from a conceptual angle: it encodes the intuition that if an agent gets what it expected, it should "feel" neutral about it.

Baselines in Policy Gradients

- In practice, $V^{\pi}(s_t)$ cannot be computed exactly, so it has to be approximated. This is usually done with a neural network, $V_{\phi}(s_t)$, which is updated concurrently with the policy (so that the value network always approximates the value function of the most recent policy).
- The simplest method for learning V_{ϕ}, used in most implementations of policy optimization algorithms (including VPG, TRPO, PPO, and A2C), is to minimize a mean-squared-error objective:

$$\phi_k = \arg\min_{\phi} \underset{s_t, \hat{R}_t \sim \pi_k}{\mathbf{E}} \left[\left(V_{\phi}(s_t) - \hat{R}_t \right)^2 \right],$$

where π_k is the policy at epoch k. This is done with one or more steps of gradient descent, starting from the previous value parameters ϕ_{k-1}.

Other Forms of the Policy Gradient

What we have seen is that the policy gradient has the general form

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbf{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\Phi_t\right],$$

where Φ_t could be any of

$$\Phi_t = R(\tau),$$

or

$$\Phi_t = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}),$$

or

$$\Phi_t = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t).$$

All of these choices lead to the same expected value for the policy gradient, despite having different variances.

Other Forms of the Policy Gradient

- **On-Policy Action-Value Function**

$$\Phi_t = Q^{\pi_\theta}(s_t, a_t)$$

- **The Advantage Function** $A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$ describes how much better or worse it is than other actions on average (relative to the current policy). This choice,

$$\Phi_t = A^{\pi_\theta}(s_t, a_t)$$

  is also valid. The proof is that it's equivalent to using $\Phi_t = Q^{\pi_\theta}(s_t, a_t)$ and then using a value function baseline, which we are always free to do.

- The formulation of policy gradients with advantage functions is extremely common, and there are many different ways of estimating the advantage function used by different algorithms.

Vanilla Policy Gradient (VPG)

Key idea: push up the probabilities of actions that lead to higher return, and push down the probabilities of actions that lead to lower return, until you arrive at the optimal policy.

- VPG is an on-policy algorithm.
- VPG can be used for environments with either discrete or continuous action spaces.
- **Exploration vs. Exploitation**: VPG trains a stochastic policy in an on-policy way. This means that it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. This may cause the policy to get trapped in local optima.

Key Equations

- Let π_θ denote a policy with parameters θ, and $J(\pi_\theta)$ denote the expected finite-horizon undiscounted return of the policy. The gradient of $J(\pi_\theta)$ is

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathbf{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t) \right],$$

  where τ is a trajectory and A^{π_θ} is the advantage function for the current policy.

- The policy gradient algorithm works by updating policy parameters via stochastic gradient ascent on policy performance:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_{\theta_k})$$

- Policy gradient implementations typically compute advantage function estimates based on the infinite-horizon discounted return, despite otherwise use the finite-horizon undiscounted policy gradient formula.

Pseudocode: Vanilla Policy Gradient Algorithm

Algorithm 1 Vanilla Policy Gradient Algorithm

Input: initial policy parameters θ_0, initial value function parameters ϕ_0

for $k = 0, 1, 2, \ldots$ **do**

Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

Compute rewards-to-go $\hat{R}_t$.

Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

or via another gradient ascent algorithm like Adam.

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2,$$

typically via some gradient descent algorithm.

end for

Outline

Trust Region Policy Optimization (John Schulman, etc)

- Assume start-state distribution ρ_0 is independent with policy
- Total expected discounted reward with policy π

$$\eta(\pi) = E_\pi[\sum_{t=0}^{\infty} \gamma^t r(s_t)]$$

- Between any two different policy $\widetilde{\pi}$ and π

$$\begin{aligned}
\eta(\widetilde{\pi}) &= \eta(\pi) + E_{\widetilde{\pi}}[\sum_{t=0}^{\infty} \gamma^t A_\pi(s_t, a_t)] \\
&= \eta(\pi) + \sum_{t=0}^{\infty} \sum_s P(s_t = s|\widetilde{\pi}) \sum_a \widetilde{\pi}(a|s) \gamma^t A_\pi(s, a) \\
&= \eta(\pi) + \sum_s \sum_{t=0}^{\infty} \gamma^t P(s_t = s|\widetilde{\pi}) \sum_a \widetilde{\pi}(a|s) A_\pi(s, a) \\
&= \eta(\pi) + \sum_s \rho_{\widetilde{\pi}}(s) \sum_a \widetilde{\pi}(a|s) A_\pi(s, a).
\end{aligned}$$

Trust Region Policy Optimization

- Find new policy $\widetilde{\pi}$ to maximize $\eta(\widetilde{\pi}) - \eta(\pi)$ for given π, that is

$$\max_{\widetilde{\pi}} \eta(\pi) + \sum_s \rho_{\widetilde{\pi}}(s) \sum_a \widetilde{\pi}(a|s) A_\pi(s,a)$$

- For simpleness, maximize the approximator

$$L_\pi(\widetilde{\pi}) = \eta(\pi) + \sum_s \rho_\pi(s) \sum_a \widetilde{\pi}(a|s) A_\pi(s,a)$$

- Parameterize the policy $\pi(a|s) := \pi_\theta(a|s)$

$$L_{\pi_{\theta_{old}}}(\pi_\theta) = \eta(\pi_{\theta_{old}}) + \sum_s \rho_{\pi_{\theta_{old}}}(s) \sum_a \pi_\theta(a|s) A_{\pi_{\theta_{old}}}(s,a)$$

Why $L_{\pi_{\theta_{old}}}(\pi_\theta)$?

- A sufficiently small step $\theta_{old} \rightarrow \theta$ improves $L_{\pi_{\theta_{old}}}(\pi_\theta)$ also improves η

$$L_{\pi_{\theta_{old}}}(\pi_{\theta_{old}}) = \eta(\pi_{\theta_{old}}),$$
$$\nabla_\theta L_{\pi_{\theta_{old}}}(\pi_\theta)|_{\theta=\theta_{old}} = \nabla_\theta \eta(\pi_\theta)|_{\theta=\theta_{old}}.$$

- Lower bounds on the improvement of η

$$\eta(\pi_{\theta_{new}}) \geq L_{\pi_{\theta_{old}}}(\pi_{\theta_{new}}) - \frac{2\epsilon\gamma}{(1-\gamma)^2}\alpha^2$$

where

$$\epsilon = \max_s |E_{a\sim\pi_{\theta_{new}}} A_{\pi_{\theta_{old}}}(s,a)|$$
$$\alpha = D_{TV}^{max}(\pi_{\theta_{old}}||\pi_{\theta_{new}}) = \max_s D_{TV}(\pi_{\theta_{old}}(\cdot|s)||\pi_{\theta_{new}}(\cdot|s))$$

Lower bound

- TV divergence between two distribution p, q (discrete case)

$$D_{TV}(p\|q) = \frac{1}{2}\sum_{X}|p(X) - q(X)|$$

- KL divergence between two distribution p, q (discrete case)

$$D_{KL}(p\|q) = \sum_{X} p(X)\log\frac{p(X)}{q(X)}$$

- $(D_{TV}(p||q))^2 \leq D_{KL}(p||q)$ (Pollard(2000),Ch.3)

- Thus obtain a lower bound

$$\eta(\pi_{\theta_{new}}) \geq L_{\pi_{\theta_{old}}}(\pi_{\theta_{new}}) - \frac{2\epsilon\gamma}{(1-\gamma)^2}\alpha$$

where

$$\alpha = D_{KL}^{max}(\pi_{\theta_{old}}||\pi_{\theta_{new}}) := \max_{s} D_{KL}(\pi_{\theta_{old}}(\cdot|s)||\pi_{\theta_{new}}(\cdot|s))$$

Practical algorithm

- The penalty coefficient $\frac{2\epsilon\gamma}{(1-\gamma)^2}$ is large in practice, which yields small update
- Take a constraint on the KL divergence, i.e., a trust region constraint:

$$\max_{\theta} \quad L_{\pi_{\theta_{old}}}(\pi_\theta)$$
$$s.t.\ D_{KL}^{max}(\pi_{\theta_{old}}||\pi_\theta) \leq \delta$$

- A heuristic approximation

$$\max_{\theta} \quad L_{\pi_{\theta_{old}}}(\pi_\theta)$$
$$s.t.\ \overline{D}_{KL}^{\rho_{\pi_{\theta_{old}}}}(\pi_{\theta_{old}}||\pi_\theta) \leq \delta$$

where

$$\overline{D}_{KL}^{\rho_{\pi_{\theta_{old}}}}(\pi_{\theta_{old}}||\pi_\theta) = E_{\pi_{\theta_{old}}}(D_{KL}(\pi_{\theta_{old}}(\cdot|s)||\pi_\theta(\cdot|s))$$

TRPO

- The objective and constraint are both zero when $\theta = \theta_k$. Furthermore, the gradient of the constraint with respect to θ is zero when $\theta = \theta_k$.
- The theoretical TRPO update isn't the easiest to work with, so TRPO makes some approximations to get an answer quickly. We Taylor expand the objective and constraint to leading order around θ_k:

$$\mathcal{L}(\theta_k, \theta) \approx g^T(\theta - \theta_k)$$
$$\bar{D}_{KL}(\theta||\theta_k) \approx \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k)$$

resulting in an approximate optimization problem,

$$\theta_{k+1} = \arg\max_{\theta} g^T(\theta - \theta_k)$$
$$\text{s.t. } \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k) \leq \delta.$$

TRPO

- By happy coincidence, the gradient g of the surrogate advantage function with respect to θ, evaluated at $\theta = \theta_k$, is exactly equal to the policy gradient, $\nabla_\theta J(\pi_\theta)$
- This approximate problem can be analytically solved by the methods of Lagrangian duality, yielding the solution:

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g.$$

- TRPO adds a modification to this update rule: a backtracking line search,

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g,$$

where $\alpha \in (0, 1)$ is the backtracking coefficient, and j is the smallest nonnegative integer such that $\pi_{\theta_{k+1}}$ satisfies the KL constraint and produces a positive surrogate advantage.

TRPO

- computing and storing the matrix inverse, H^{-1}, is painfully expensive when dealing with neural network policies with thousands or millions of parameters. TRPO sidesteps the issue by using the 'conjugate gradient' algorithm to solve $Hx = g$ for $x = H^{-1}g$, requiring only a function which can compute the matrix-vector product Hx instead of computing and storing the whole matrix H directly.

$$Hx = \nabla_\theta \left(\left(\nabla_\theta \bar{D}_{KL}(\theta || \theta_k) \right)^T x \right)$$

Pseudocode: TRPO

Algorithm 2 Trust Region Policy Optimization

Input: initial policy θ_0, initial value function ϕ_0, KL-divergence limit δ, backtracking coefficient α

for $k = 0, 1, 2, \ldots$ **do**

Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

Compute rewards-to-go $\hat{R}_t$.

Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

Use the conjugate gradient algorithm to compute $\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k$, where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.

Update the policy by backtracking line search with $\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k$, where j is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

end for

Proximal Policy Optimization (PPO)

- PPO is motivated by the same question as TRPO: how can we take the biggest possible improvement step on a policy using the data we currently have, without stepping so far that we accidentally cause performance collapse? Where TRPO tries to solve this problem with a complex second-order method, PPO is a family of first-order methods that use a few other tricks to keep new policies close to old.
- **PPO-Penalty** approximately solves a KL-constrained update like TRPO, but penalizes the KL-divergence in the objective function instead of making it a hard constraint, and automatically adjusts the penalty coefficient over the course of training so that it's scaled appropriately.
- **PPO-Clip** doesn't have a KL-divergence term in the objective and doesn't have a constraint at all. Instead relies on specialized clipping in the objective function to remove incentives for the new policy to get far from the old policy.

Key Equations

- PPO-clip updates policies via

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a \sim \pi_{\theta_k}}{\mathrm{E}} \left[L(s, a, \theta_k, \theta) \right],$$

typically taking multiple steps of (usually minibatch) SGD to maximize the objective. Here L is given by

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \quad \text{clip} \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1 - \epsilon, 1 + \epsilon \right) A^{\pi_{\theta_k}}(s, a) \right),$$

in which ϵ is a (small) hyperparameter which roughly says how far away the new policy is allowed to go from the old.

- This is a pretty complex expression, and it's hard to tell at first glance what it's doing, or how it helps keep the new policy close to the old policy. As it turns out, there's a considerably simplified version of this objective which is a bit easier to grapple with

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \quad g(\epsilon, A^{\pi_{\theta_k}}(s, a)) \right),$$

  where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0. \end{cases}$$

- To figure out what intuition to take away from this, let's look at a single state-action pair (s, a), and think of cases.

Pseudocode: PPO

Algorithm 3 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

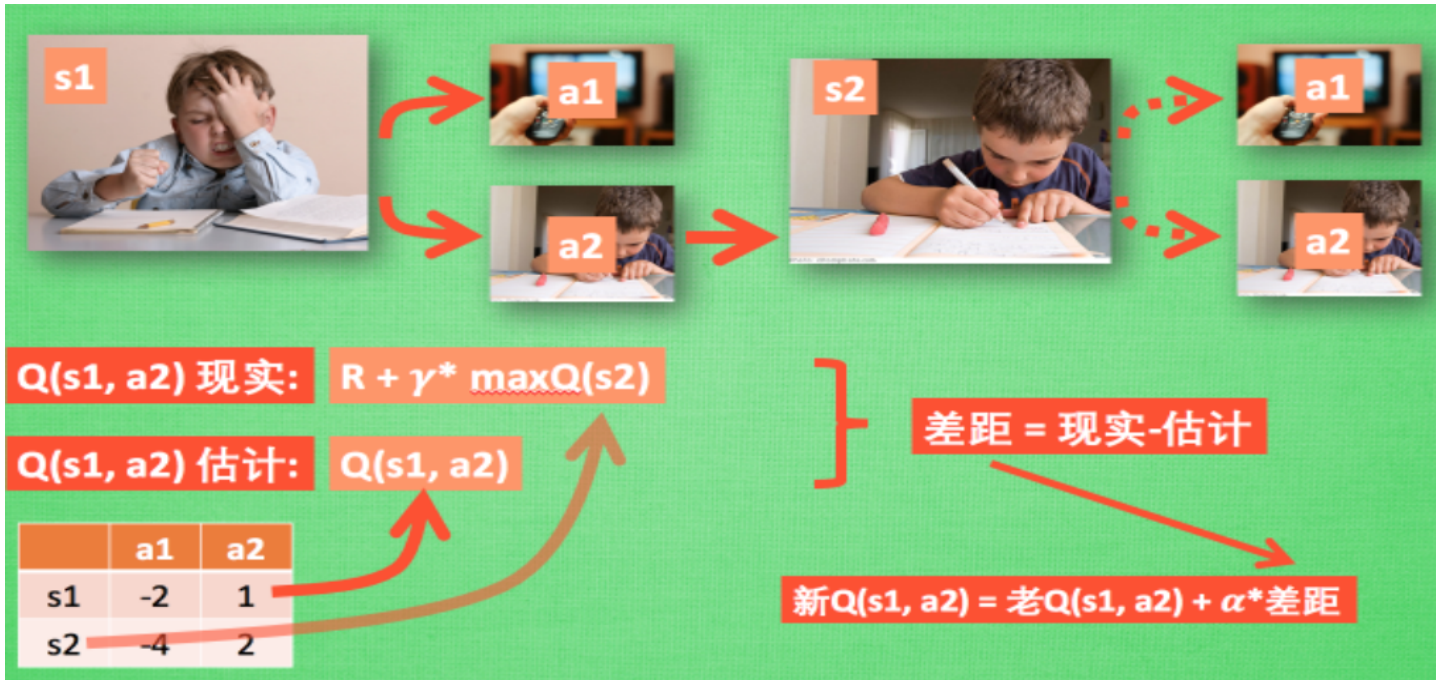
$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

Q-Learning

- Off-policy and model-free

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

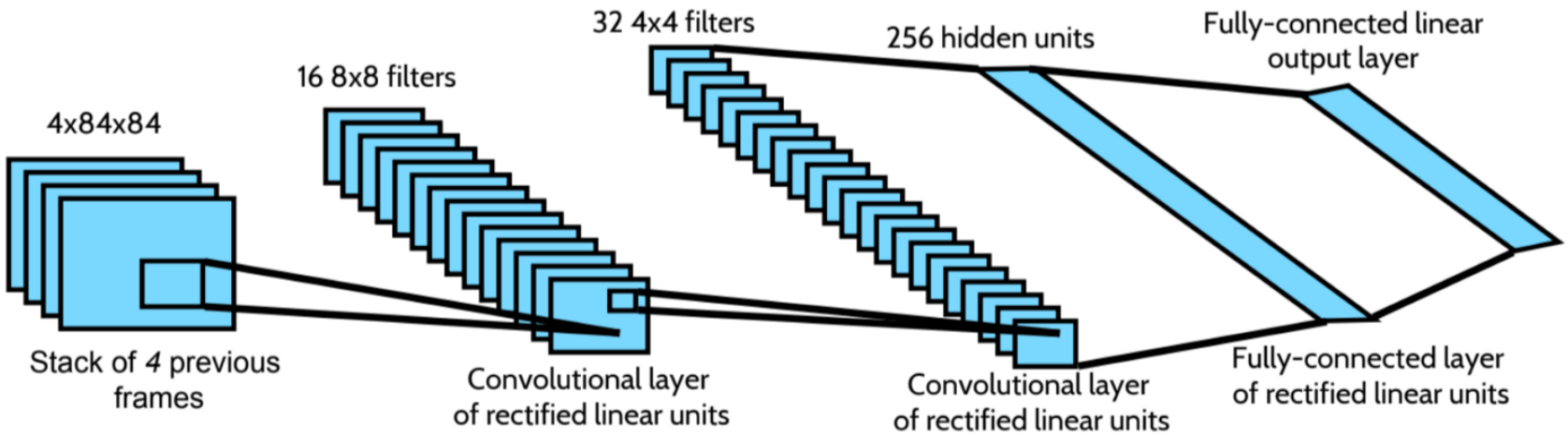


Deep Q-Networks — Motivation

- Bellman equation:

$$Q^*(s,a) = \mathrm{E}_{s' \sim \mathrm{P}(\cdot|\mathrm{s},\mathrm{a})}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')|s,a\right].$$

- Large state and/or action spaces make it intractable to learn Q value estimates for each state and action pair independently.
- Estimate $Q^*(s,a)$ with a function approximator : $Q(s,a;\theta)$.
- Refer a neural network function approximator with weight θ as Q-network.



Deep Q-Networks — Motivation

- A Q-network can be trained by minimizing a sequence of loss function $l(\theta^i)$ that changes at each iteration i:

$$\min_{\theta^i} l(\theta^i) := \mathrm{E}_{(s,a,r)\sim\mathcal{D}}\left[y_i - Q(s,a;\theta^i)\right]^2.$$

- $y_i = \mathrm{E}_{s'\sim \mathrm{P}(\cdot|\mathrm{s},\mathrm{a})}\left[r(s,a) + \gamma \max_{a'} Q(s',a';\theta^{i-1})|s,a\right]$ is the target.
- Model-free and off-policy.
- The optimal action $a^*(s)$ can be found by solving

$$a^*(s) = \arg\max_a Q(s,a;\theta^*).$$

Deep Q-Networks — 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

Deep Deterministic Policy Gradient

- Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.
- This approach is closely connected to Q-learning, and is motivated the same way: if you know the optimal action-value function $Q^*(s,a)$, then in any given state, the optimal action $a^*(s)$ can be found by solving

$$a^*(s) = \arg\max_a Q^*(s,a).$$

- DDPG interleaves learning an approximator to $Q^*(s,a)$ with learning an approximator to $a^*(s)$, and it does so in a way which is specifically adapted for environments with continuous action spaces. But what does it mean that DDPG is adapted *specifically* for environments with continuous action spaces? It relates to how we compute the max over actions in $\max_a Q^*(s,a)$.

- When there are a finite number of discrete actions, the max poses no problem, because we can just compute the Q-values for each action separately and directly compare them. (This also immediately gives us the action which maximizes the Q-value.) But when the action space is continuous, we can't exhaustively evaluate the space, and solving the optimization problem is highly non-trivial. Using a normal optimization algorithm would make calculating $\max_a Q^*(s, a)$ a painfully expensive subroutine. And since it would need to be run every time the agent wants to take an action in the environment, this is unacceptable.

- Because the action space is continuous, the function $Q^*(s, a)$ is presumed to be differentiable with respect to the action argument. This allows us to set up an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact. Then, instead of running an expensive optimization subroutine each time we wish to compute $\max_a Q(s, a)$, we can approximate it with $\max_a Q(s, a) \approx Q(s, \mu(s))$.

The Q-Learning Side of DDPG

- First, let's recap the Bellman equation describing the optimal action-value function, $Q^*(s,a)$. It's given by

$$Q^*(s,a) = \underset{s' \sim P}{\mathrm{E}} \left[r(s,a) + \gamma \max_{a'} Q^*(s',a') \right]$$

  where $s' \sim P$ is shorthand for saying that the next state, s', is sampled by the environment from a distribution $P(\cdot|s,a)$.

- This Bellman equation is the starting point for learning an approximator to $Q^*(s,a)$. Suppose the approximator is a neural network $Q_\phi(s,a)$, with parameters ϕ, and that we have collected a set $\mathcal{D}$ of transitions (s,a,r,s',d) (where d indicates whether state s' is terminal). We can set up a **mean-squared Bellman error (MSBE)** function, which tells us roughly how closely Q_ϕ comes to satisfying the Bellman equation:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d) \sim \mathcal{D}}{\mathrm{E}} \left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d) \max_{a'} Q_\phi(s',a') \right) \right)^2 \right]$$

- Here, in evaluating $(1-d)$, we've used: "True" to 1 and "False" to zero. Thus, when "d==True"—which is to say, when s' is a terminal state—the Q-function should show that the agent gets no additional rewards after the current state.

- Q-learning algorithms for function approximators, such as DQN (and all its variants) and DDPG, are largely based on minimizing this MSBE loss function. There are two main tricks employed by all of them which are worth describing, and then a specific detail for DDPG.

- **Trick One: Replay Buffers**. All standard algorithms for training a deep neural network to approximate $Q^*(s,a)$ make use of an experience replay buffer. This is the set $\mathcal{D}$ of previous experiences. In order for the algorithm to have stable behavior, the replay buffer should be large enough to contain a wide range of experiences, but it may not always be good to keep everything. If you only use the very-most recent data, you will overfit to that and things will break; if you use too much experience, you may slow down your learning. This may take some tuning to get right.

- **Trick Two: Target Networks**. Q-learning algorithms make use of **target networks**. The term

$$r + \gamma(1 - d) \max_{a'} Q_\phi(s', a')$$

is called the **target**, because when we minimize the MSBE loss, we are trying to make the Q-function be more like this target. Problematically, the target depends on the same parameters we are trying to train: ϕ. This makes MSBE minimization unstable. The solution is to use a set of parameters which comes close to ϕ, but with a time delay—that is to say, a second network, called the target network, which lags the first. The parameters of the target network are denoted ϕ_{targ}.

- In DQN-based algorithms, the target network is just copied over from the main network every some-fixed-number of steps. In DDPG-style algorithms, the target network is updated once per main network update by polyak averaging:

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$$

where ρ is between 0 and 1 (usually close to 1).

- **DDPG Detail: Calculating the Max Over Actions in the Target.** As mentioned earlier: computing the maximum over actions in the target is a challenge in continuous action spaces. DDPG deals with this by using a **target policy network** to compute an action which approximately maximizes $Q_{\phi_{\text{targ}}}$. The target policy network is found the same way as the target Q-function: by polyak averaging the policy parameters over the course of training.

- Putting it all together, Q-learning in DDPG is performed by minimizing the following MSBE loss with stochastic gradient descent:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s')) \right) \right)^2 \right],$$

where $\mu_{\theta_{\text{targ}}}$ is the target policy.

The Policy Learning Side of DDPG

- Policy learning in DDPG is fairly simple. We want to learn a deterministic policy $\mu_\theta(s)$ which gives the action that maximizes $Q_\phi(s, a)$. Because the action space is continuous, and we assume the Q-function is differentiable with respect to action, we can just perform gradient ascent (with respect to policy parameters only) to solve

$$\max_\theta \underset{s\sim\mathcal{D}}{\mathrm{E}} \left[Q_\phi(s, \mu_\theta(s))\right].$$

- Note that the Q-function parameters are treated as constants here.

Pseudocode: DDPG

Algorithm 4 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
    Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
    Execute a in the environment
    Observe next state s', reward r, and done signal d to indicate whether s' is terminal
    Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
    If s' is terminal, reset environment state.
    if it's time to update **then**
        for however many updates **do**
            Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
            Compute targets $y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$
            Update Q-function by one step of gradient descent using $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$
            Update policy by one step of gradient ascent using $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$
            Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

        end for
    end if
until convergence

Twin Delayed DDPG (TD3)

- A common failure mode for DDPG is that the learned Q-function begins to dramatically overestimate Q-values, which then leads to the policy breaking, because it exploits the errors in the Q-function.
- **Trick One: Clipped Double-Q Learning**. TD3 learns *two* Q-functions instead of one (hence "twin"), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.
- **Trick Two: "Delayed" Policy Updates**. TD3 updates the policy (and target networks) less frequently than the Q-function. The paper recommends one policy update for every two Q-function updates.
- **Trick Three: Target Policy Smoothing**. TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

Key Equations: target policy smoothing

TD3 concurrently learns two Q-functions, Q_{ϕ_1} and Q_{ϕ_2}, by mean square Bellman error minimization, in almost the same way that DDPG learns its single Q-function.

- **target policy smoothing**. Actions used to form the Q-learning target are based on the target policy, $\mu_{\theta_{\text{targ}}}$, but with clipped noise added on each dimension of the action. After adding the clipped noise, the target action is then clipped to lie in the valid action range (all valid actions, a, satisfy $a_{Low} \leq a \leq a_{High}$). The target actions are thus:

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

- Target policy smoothing essentially serves as a regularizer for the algorithm. It addresses a particular failure mode that can happen in DDPG: if the Q-function approximator develops an incorrect sharp peak for some actions, the policy will quickly exploit that peak and then have brittle or incorrect behavior. This can be averted by smoothing out the Q-function over similar actions, which target policy smoothing is designed to do.

clipped double-Q learning

- Both Q-functions use a single target, calculated using whichever of the two Q-functions gives a smaller target value:

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{i,\text{targ}}}(s', a'(s')),$$

and then both are learned by regressing to this target:

$$L(\phi_1, \mathcal{D}) = \mathop{\mathbf{E}}_{(s,a,r,s',d)\sim\mathcal{D}} \left[\left(Q_{\phi_1}(s, a) - y(r, s', d) \right)^2 \right],$$

$$L(\phi_2, \mathcal{D}) = \mathop{\mathbf{E}}_{(s,a,r,s',d)\sim\mathcal{D}} \left[\left(Q_{\phi_2}(s, a) - y(r, s', d) \right)^2 \right].$$

- Using the smaller Q-value for the target, and regressing towards that, helps fend off overestimation in the Q-function.

- the policy is learned just by maximizing Q_{ϕ_1}:

$$\max_{\theta} \underset{s \sim \mathcal{D}}{\mathrm{E}} \left[Q_{\phi_1}(s, \mu_\theta(s)) \right],$$

  which is pretty much unchanged from DDPG. However, in TD3, the policy is updated less frequently than the Q-functions are. This helps damp the volatility that normally arises in DDPG because of how a policy update changes the target.

- **Exploration vs. Exploitation**: TD3 trains a deterministic policy in an off-policy way. Because the policy is deterministic, if the agent were to explore on-policy, in the beginning it would probably not try a wide enough variety of actions to find useful learning signals. To make TD3 policies explore better, we add noise to their actions at training time, typically uncorrelated mean-zero Gaussian noise. To facilitate getting higher-quality training data, you may reduce the scale of the noise over the course of training.

Pseudocode: TD3

Algorithm 5 Twin Delayed DDPG

Input: initial policy θ, Q-function ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$. Set $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute target actions $a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \quad \epsilon \sim \mathcal{N}(0, \sigma)$
      Compute targets $y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$
      Update Q-functions by one gradient step: $\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi,i}(s, a) - y(r, s', d)\right)^2$ for $i = 1, 2$
      if $j \mod \texttt{policy\_delay} = 0$ **then**
        Update policy by one step of gradient ascent using $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi,1}(s, \mu_\theta(s))$
        Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \text{ for } i = 1, 2, \quad \theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

      end if
    end for
  end if
until convergence

Soft Actor-Critic (SAC)

- Soft Actor Critic (SAC) is an algorithm which optimizes a stochastic policy in an off-policy way, forming a bridge between stochastic policy optimization and DDPG-style approaches. It isn't a direct successor to TD3 (having been published roughly concurrently), but it incorporates the clipped double-Q trick, and due to the inherent stochasticity of the policy in SAC, it also winds up benefiting from something like target policy smoothing.
- A central feature of SAC is **entropy regularization**. The policy is trained to maximize a trade-off between expected return and 'entropy', a measure of randomness in the policy. This has a close connection to the exploration-exploitation trade-off: increasing entropy results in more exploration, which can accelerate learning later on. It can also prevent the policy from prematurely converging to a bad local optimum.

Entropy-Regularized Reinforcement Learning

- Entropy is a quantity which, roughly speaking, says how random a random variable is. If a coin is weighted so that it almost always comes up heads, it has low entropy; if it's evenly weighted and has a half chance of either outcome, it has high entropy.
- Let x be a random variable with probability mass or density function P. The entropy H of x is computed from its distribution P according to

$$H(P) = \underset{x \sim P}{\mathbf{E}} \left[-\log P(x) \right].$$

- In entropy-regularized reinforcement learning, the agent gets a bonus reward at each time step proportional to the entropy of the policy at that timestep. This changes 'the RL problem' to:

$$\pi^* = \arg\max_{\pi} \underset{\tau \sim \pi}{\mathbf{E}} \left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right) \right) \right],$$

where $\alpha > 0$ is the trade-off coefficient.

- We can now define the slightly-different value functions in this setting. V^π is changed to include the entropy bonuses from every timestep:

$$V^\pi(s) = \mathop{\mathbf{E}}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\Big(R(s_t,a_t,s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right)\Big)\,\middle|\, s_0 = s\right]$$

- Q^π is changed to include the entropy bonuses from every timestep except the first:

$$Q^\pi(s,a) = \mathop{\mathbf{E}}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t,s_{t+1}) + \alpha\sum_{t=1}^{\infty}\gamma^t H\left(\pi(\cdot|s_t)\right)\,\middle|\, s_0 = s, a_0 = a\right]$$

- With these definitions, V^π and Q^π are connected by:

$$V^\pi(s) = \mathop{\mathbf{E}}_{a\sim\pi}\left[Q^\pi(s,a)\right] + \alpha H\left(\pi(\cdot|s)\right)$$

and the Bellman equation for Q^π is

$$\begin{aligned} Q^\pi(s,a) &= \mathop{\mathbf{E}}_{\substack{s'\sim P\\ a'\sim\pi}}\left[R(s,a,s') + \gamma\left(Q^\pi(s',a') + \alpha H\left(\pi(\cdot|s')\right)\right)\right] \\ &= \mathop{\mathbf{E}}_{s'\sim P}\left[R(s,a,s') + \gamma V^\pi(s')\right]. \end{aligned}$$

Soft Actor-Critic: Learning Q

- SAC concurrently learns a policy π_θ, two Q-functions Q_{ϕ_1}, Q_{ϕ_2}, and a value function V_ψ.
- **Learning Q**. The Q-functions are learned by MSBE minimization, using a **target value network** to form the Bellman backups. They both use the same target, like in TD3, and have loss functions:

$$L(\phi_i, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_{\phi_i}(s,a) - \left(r + \gamma(1-d) V_{\psi_{\text{targ}}}(s')\right) \right)^2 \right].$$

- The target value network, like the target networks in DDPG and TD3, is obtained by polyak averaging the value network parameters over the course of training.

Learning V

- **Learning V**. The value function is learned by exploiting (a sample-based approximation of) the connection between Q^π and V^π. Before we go into the learning rule, let's first rewrite the connection equation by using the definition of entropy to obtain:

$$\begin{aligned} V^\pi(s) &= \mathop{\mathbf{E}}_{a\sim\pi}\left[Q^\pi(s,a)\right] + \alpha H\left(\pi(\cdot|s)\right) \\ &= \mathop{\mathbf{E}}_{a\sim\pi}\left[Q^\pi(s,a) - \alpha \log \pi(a|s)\right]. \end{aligned}$$

- The RHS is an expectation over actions, so we can approximate it by sampling from the policy:

$$V^\pi(s) \approx Q^\pi(s,\tilde{a}) - \alpha \log \pi(\tilde{a}|s), \qquad \tilde{a} \sim \pi(\cdot|s).$$

- SAC sets up a mean-squared-error loss for V_ψ based on this approximation. But what Q-value do we use? SAC uses **clipped double-Q** like TD3 for learning the value function, and takes the minimum Q-value between the two approximators. So the SAC loss for value function parameters is:

$$L(\psi, \mathcal{D}) = \underset{\substack{s \sim \mathcal{D} \\ \tilde{a} \sim \pi_\theta}}{\mathbf{E}} \left[\left(V_\psi(s) - \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}) - \alpha \log \pi_\theta(\tilde{a}|s) \right) \right)^2 \right].$$

- Importantly, we do NOT use actions from the replay buffer here: these actions are sampled fresh from the current version of the policy.

Learning the Policy

- The policy should, in each state, act to maximize the expected future return plus expected future entropy. That is, it should maximize $V^{\pi}(s)$, which we expand out (as before) into

$$\underset{a\sim\pi}{\mathbf{E}}\left[Q^{\pi}(s,a) - \alpha \log \pi(a|s)\right].$$

- The way we optimize the policy makes use of the **reparameterization trick**, in which a sample from $\pi_{\theta}(\cdot|s)$ is drawn by computing a deterministic function of state, policy parameters, and independent noise. To illustrate: following the authors of the SAC paper, we use a squashed Gaussian policy, which means that samples are obtained according to

$$\tilde{a}_{\theta}(s,\xi) = \tanh\left(\mu_{\theta}(s) + \sigma_{\theta}(s) \odot \xi\right), \qquad \xi \sim \mathcal{N}(0,I).$$

This policy has two key differences from the policies we use in the other policy optimization algorithms:

- The squashing function. The $\tanh$ in the SAC policy ensures that actions are bounded to a finite range. This is absent in the VPG, TRPO, and PPO policies. It also changes the distribution: before the $\tanh$ the SAC policy is a factored Gaussian like the other algorithms' policies, but after the $\tanh$ it is not. (You can still compute the log-probabilities of actions in closed form, though: see the paper appendix for details.)

- The way standard deviations are parameterized. In VPG, TRPO, and PPO, we represent the log std devs with state-independent parameter vectors. In SAC, we represent the log std devs as outputs from the neural network, meaning that they depend on state in a complex way. SAC with state-independent log std devs, in our experience, did not work. (Can you think of why? Or better yet: run an experiment to verify?)

- The reparameterization trick allows us to rewrite the expectation over actions (which contains a pain point: the distribution depends on the policy parameters) into an expectation over noise (which removes the pain point: the distribution now has no dependence on parameters):

$$\underset{a\sim\pi_\theta}{\mathbf{E}}\left[Q^{\pi_\theta}(s,a)-\alpha\log\pi_\theta(a|s)\right]=\underset{\xi\sim\mathcal{N}}{\mathbf{E}}\left[Q^{\pi_\theta}(s,\tilde{a}_\theta(s,\xi))-\alpha\log\pi_\theta(\tilde{a}_\theta(s,\xi)|s)\right]$$

- To get the policy loss, the final step is that we need to substitute Q^{π_θ} with one of our function approximators. The same as in TD3, we use Q_{ϕ_1}. The policy is thus optimized according to

$$\max_\theta \underset{\substack{s\sim\mathcal{D}\\ \xi\sim\mathcal{N}}}{\mathbf{E}}\left[Q_{\phi_1}(s,\tilde{a}_\theta(s,\xi))-\alpha\log\pi_\theta(\tilde{a}_\theta(s,\xi)|s)\right],$$

  which is almost the same as the DDPG and TD3 policy optimization, except for the stochasticity and entropy term.

Pseudocode

Algorithm 6 Soft Actor-Critic

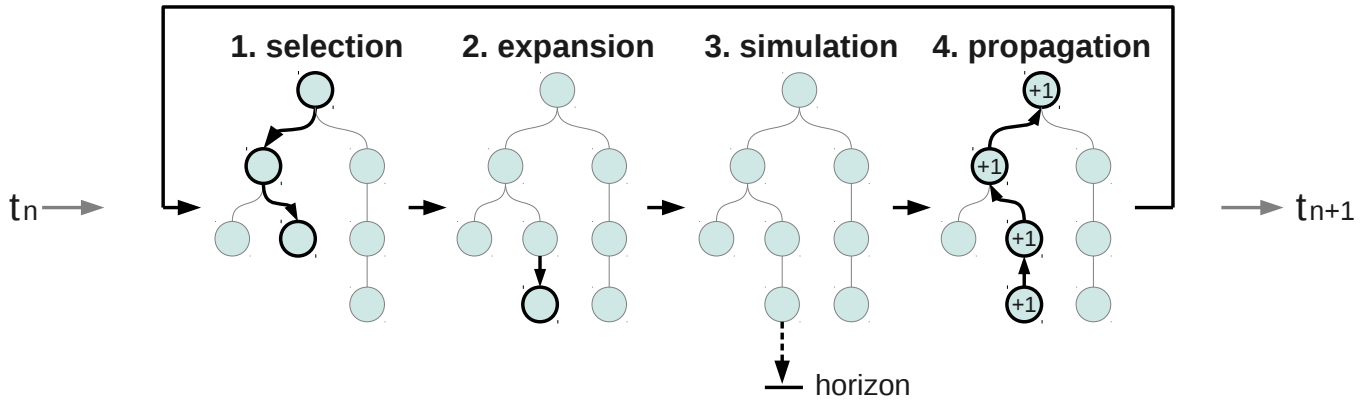
Input: initial policy θ, Q-function ϕ_1, ϕ_2, V-function ψ, empty replay buffer $\mathcal{D}$. Set $\psi_{\text{targ}} \leftarrow \psi$.
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for Q and V functions:

$$y_q(r, s', d) = r + \gamma(1 - d)V_{\psi_{\text{targ}}}(s'), \quad y_v(s) = \min_{i=1,2} Q_{\phi_i}(s, \tilde{a}) - \alpha \log \pi_\theta(\tilde{a}|s), \quad \tilde{a} \sim \pi_\theta(\cdot|s)$$

      Update Q-functions by one gradient step: $\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi,i}(s, a) - y_q(r, s', d)\right)^2$ for $i = 1, 2$
      Update V-function by one step of gradient descent using $\nabla_\psi \frac{1}{|B|} \sum_{s \in B} \left(V_\psi(s) - y_v(s)\right)^2$
      Update policy by one step of gradient ascent using $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(Q_{\phi,1}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\Big)$, where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target value network with $\psi_{\text{targ}} \leftarrow \rho\psi_{\text{targ}} + (1 - \rho)\psi$
    end for
  end if
until convergence

Outline

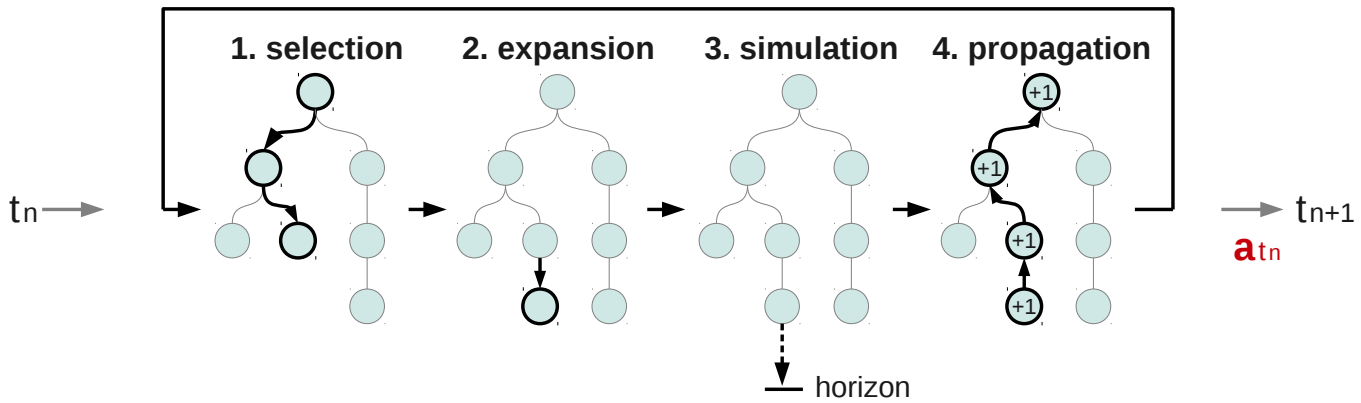
Monte-Carlo Tree Search (MCTS)



Starting from an initial state:

1. select the state we want to expand from
2. add the generated state in memory
3. evaluate the new state with a default policy until horizon is reached
4. back-propagation of some information:
   - $n(s,a)$: number of times decision a has been simulated in s
   - $n(s)$: number of time s has been visited in simulations
   - $\hat{Q}(s,a)$: mean reward of simulations where a was whosen in s

Main steps of MCTS

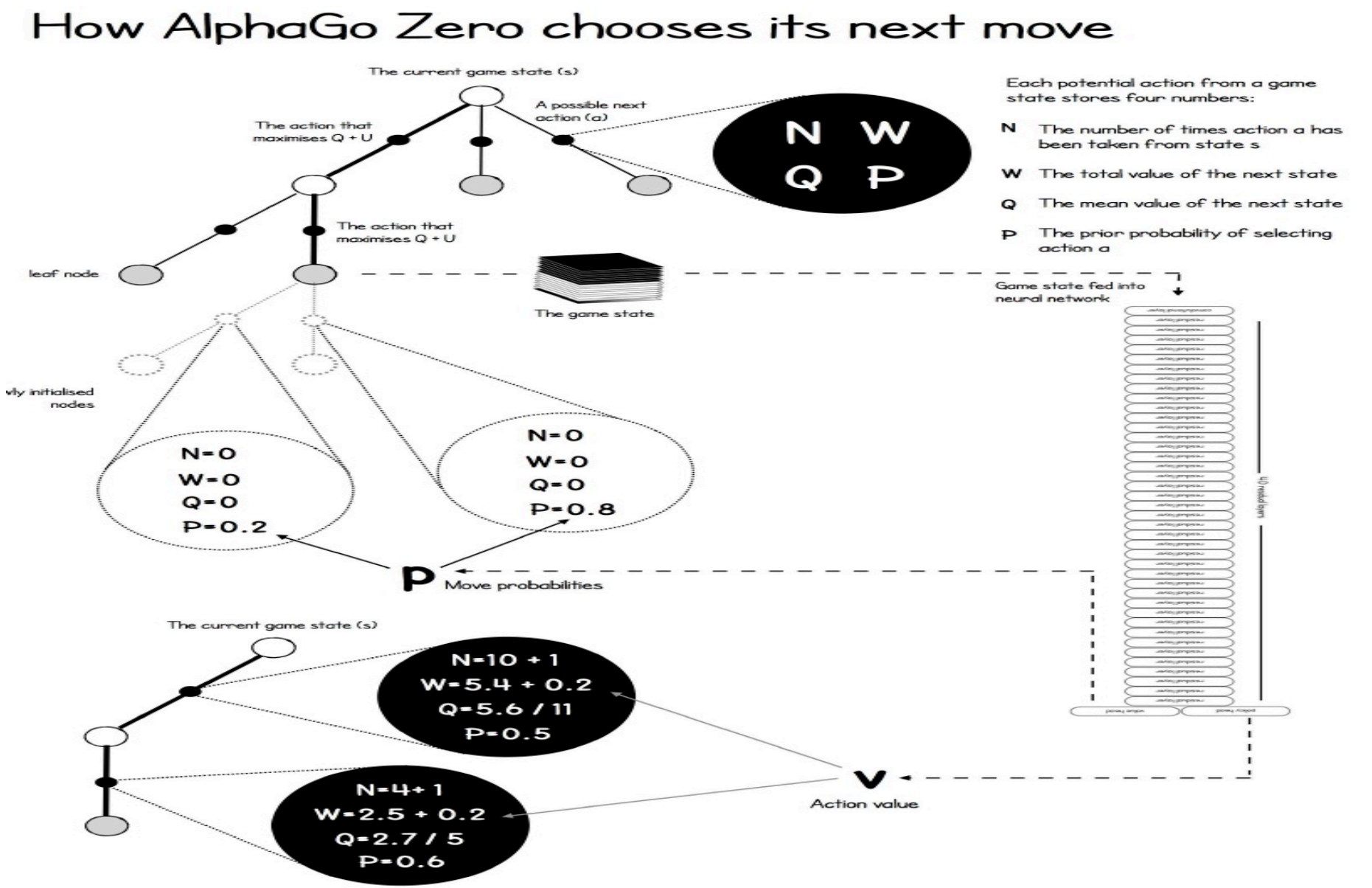


The selected decision

a_{t_n} = the most visited decision from the current state (root node)

Thanks: `https://github.com/jdhp-docs/mcts`

AlphaGo Zero



AlphaGo Zero

First, run the following simulation 1,600 times...

Start at the root node of the tree (the current game state)

1. Choose the action that maximises...

$$Q + U$$

Q: The mean value of the next state

U: A function of **P** and **N** that increases if an action hasn't been explored much, relative to the other actions, or if the prior probability of the action is high

Early on in the simulation, U dominates (more exploration), but later; Q is more important (less exploration)

2. Continue until a leaf node is reached

The game state of the leaf node is passed into the neural network, which outputs predictions about two things:

p Move probabilities

v Value of the state (for the current player)

The move probabilities p are attached to the new feasible actions from the leaf node

3. Backup previous edges

Each edge that was traversed to get to the leaf node is updated as follows:

$$N \longrightarrow N + 1$$
$$W \longrightarrow W + v$$
$$Q = W / N$$

AlphaGo Zero

...then select a move

After 1,600 simulations, the move can either be chosen:

Deterministically (for competitive play)

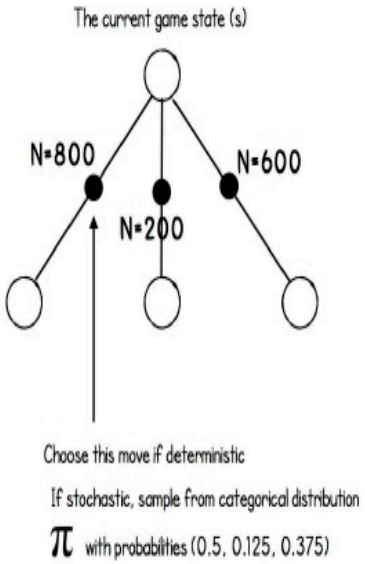
Choose the action from the current state with greatest N

Stochastically (for exploratory play)

Choose the action from the current state from the distribution

$$\pi \sim N^{1/\tau}$$

where τ is a temperature parameter, controlling exploration



Other points

- The sub-tree from the chosen move is retained for calculating subsequent moves
- The rest of the tree is discarded

AlphaGo Zero

SELF PLAY

Create a 'training set'

The best current player plays 25,000 games against itself

See MCTS section to understand how AlphaGo Zero selects each move

At each move, the following information is stored

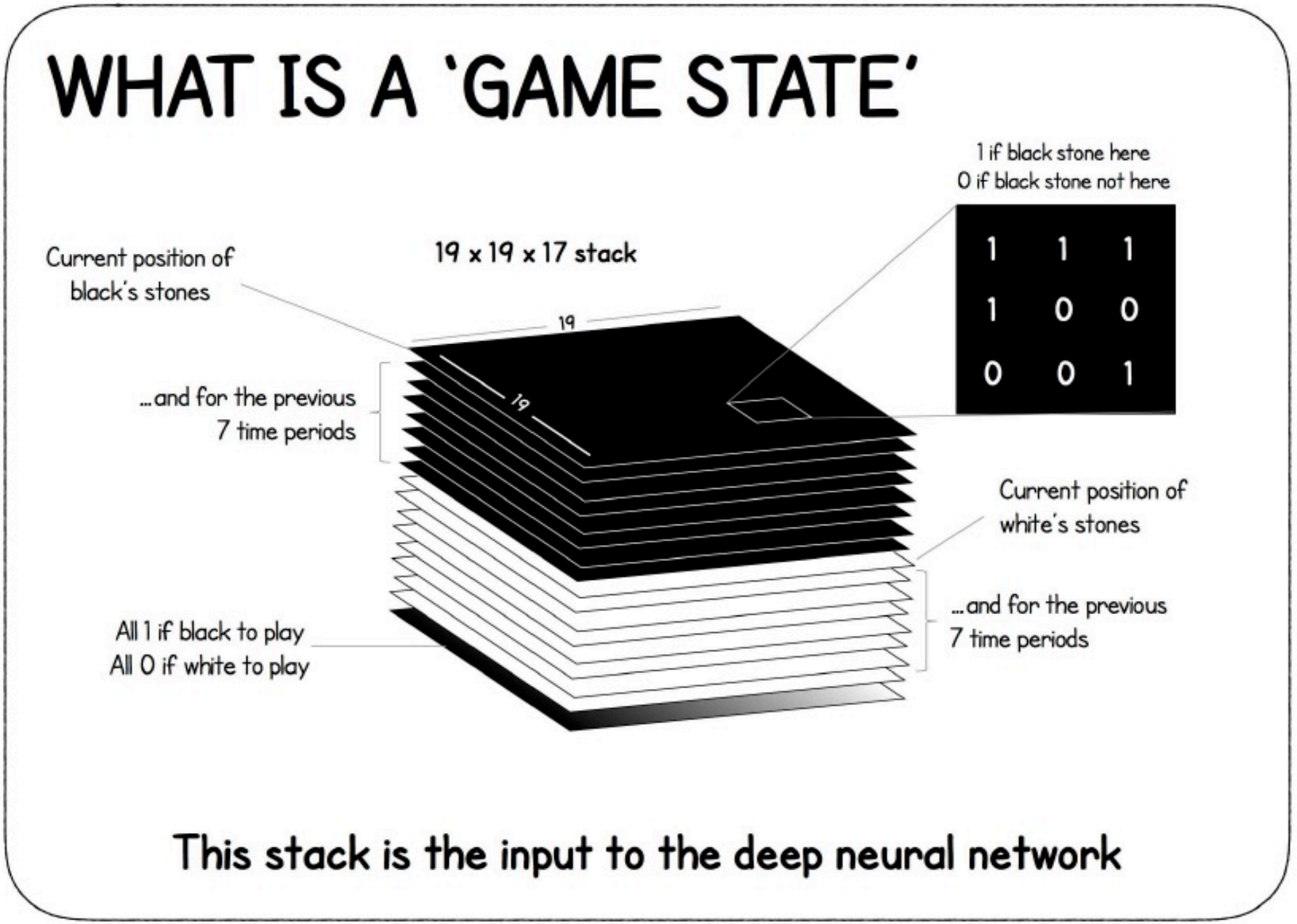
The game state
(see 'What is a Game State section')

π

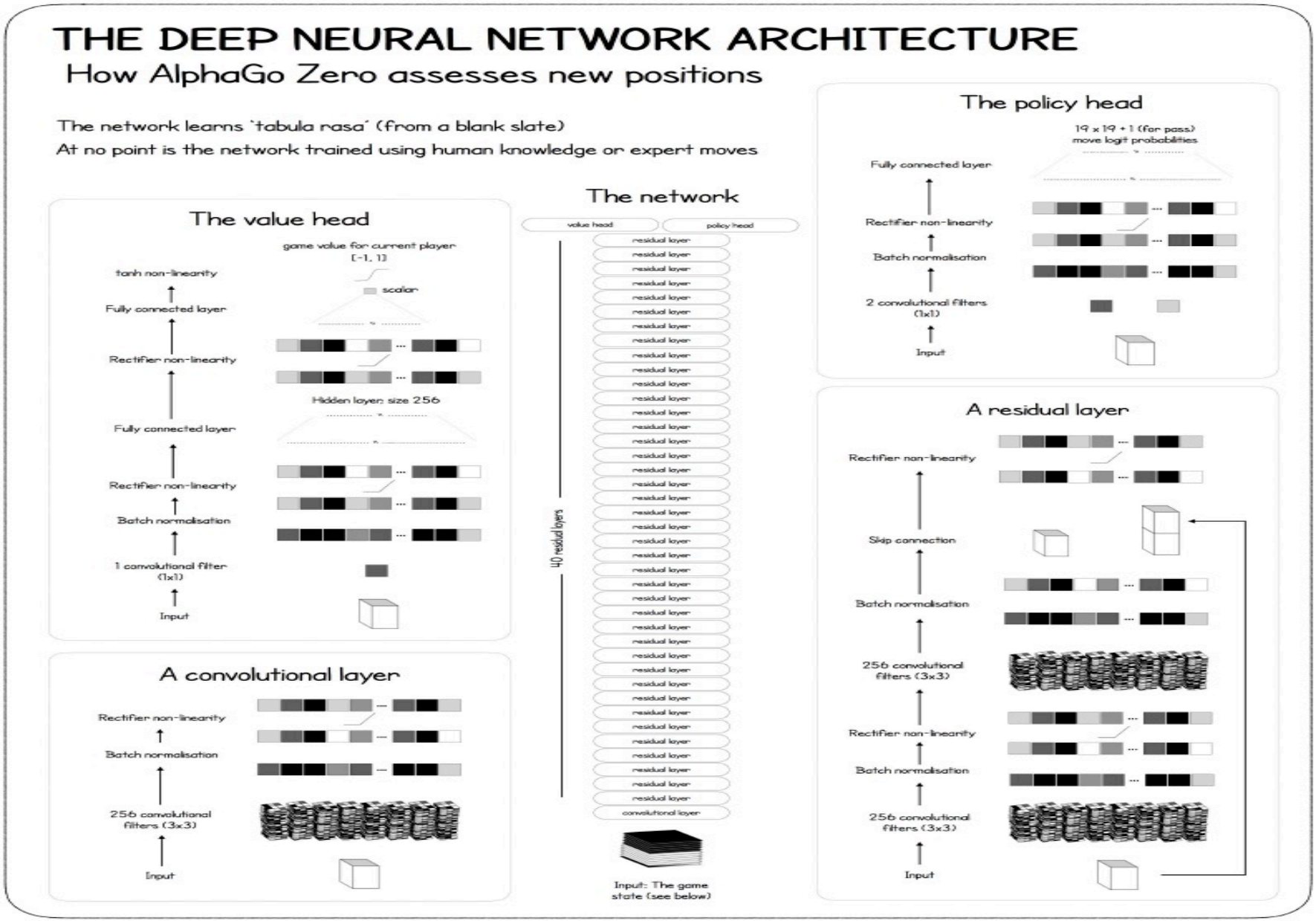
The search probabilities
(from the MCTS)

The winner
(+1 if this player won, -1 if this player lost - added once the game has finished)

AlphaGo Zero



AlphaGo Zero



AlphaGo Zero

RETRAIN NETWORK

Optimise the network weights

A TRAINING LOOP

Sample a mini-batch of 2048 positions from the last 500,000 games

Retrain the current neural network on these positions
- The game states are the input (see 'Deep Neural Network Architecture')

Loss function
Compares predictions from the neural network with the search probabilities and actual winner

PREDICTIONS: $\mathbf{p}$ Cross-entropy π :ACTUAL

\+

$\mathbf{v}$ Mean-squared error 

\+

Regularisation

After every 1,000 training loops, evaluate the network

AlphaGo Zero

EVALUATE NETWORK

Test to see if the new network is stronger

Play 400 games between the latest neural network and the current best neural network

Both players use MCTS to select their moves, with their respective neural networks to evaluate leaf nodes

Latest player must win 55% of games to be declared the new best player