

NAIL139 AI for Computer Games

Lecture 11: Deep Reinforcement Learning 2

Adam Dingle

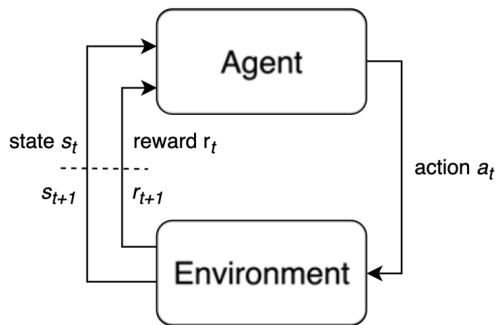
December 11, 2025

Deep reinforcement learning: algorithms

- ▶ Deep Q-networks (DQN)
 - ▶ Playing Atari with Deep Reinforcement Learning (DeepMind, 2013)
 - ▶ Human-Level Control through Deep Reinforcement Learning (DeepMind, 2015)
- ▶ AlphaZero
 - ▶ Mastering the game of Go without human knowledge (DeepMind, 2017)
 - ▶ A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play (DeepMind, 2018)

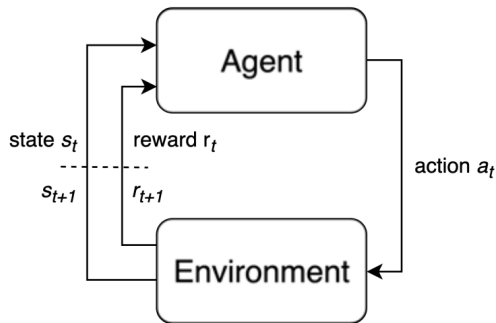
Review: reinforcement learning model

- ▶ An **agent** is in an **environment**
- ▶ The environment has a current **state**
- ▶ At each **time step** t the agent observes state s_t , chooses an **action** a_t
- ▶ The environment produces the next state plus a **reward** r_t



Review: reinforcement learning model

- ▶ A **policy** is a function from a state to an action
- ▶ **Objective**: maximize the (weighted) sum of rewards
- ▶ At each time step (s_t, a_t, r_t) is an **experience**
- ▶ An **episode** lasts from time $t = 0$ until the environment **terminates**
- ▶ A **trajectory** is a sequence of experiences over an episode



Review: states, actions, rewards

- ▶ $s_t \in S$ is the **state** ($S =$ state space)
- ▶ $a_t \in A$ is the **action** ($A =$ action space)
- ▶ $r_t = R(s_t, a_t, s_{t+1})$ is the **reward** ($R =$ reward function)
- ▶ The **return** after time t is the sum of future rewards, possibly with a discount factor $\gamma \in [0, 1]$:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \dots + \gamma^{T-t} r_T$$

- ▶ The agent wants to maximize the **expected return**

$$J_t = \mathbb{E}_{\tau \sim \pi} [R_t]$$

Review: Learnable functions

What can an agent learn?

1. A **policy** π that maps states to actions: $a \sim \pi(s)$
2. A **value function** $V^\pi(s)$ or $Q^\pi(s, a)$ that estimates $\mathbb{E}_{\tau \sim \pi}[R_t]$
3. A **model** of the environment: $P(s' | s, a)$

A policy may be stochastic!

Value functions come in two forms:

- ▶ $V^\pi(s)$ estimates the return from being in a state
 - ▶ Not too useful without a model!
- ▶ $Q^\pi(s, a)$ estimates the return from taking an action in a state

The Q-function

- ▶ $Q^\pi(s, a)$ is the **expected return** from starting in s , taking action a , and then following policy π :

$$Q^\pi(s, a) = \mathbb{E}_{s_0=s, a_0=a, \tau \sim \pi} [R_t]$$

- ▶ π^* is the **optimal policy** that will provide maximal returns from every state
- ▶ $Q^*(s, a)$ is the **optimal action-value function**, defined as

$$Q^*(s, a) = \max_{\pi} Q_{\pi}(s, a) = Q^{\pi^*}(s, a)$$

- ▶ **Our goal**: learn to estimate $Q^*(s, a)$ as accurately as possible

Temporal difference learning

- ▶ We use a neural network that takes (s, a) pairs as inputs and produces estimates of $Q^*(s, a)$
- ▶ We generate trajectories τ and use the network to produce a **predicted value** $\hat{Q}(s, a)$ for each (s, a) pair
- ▶ We use the trajectories to generate **target** Q-values $Q_{\text{tar}}(s, a)$
- ▶ We compute a **loss** between \hat{Q} and Q_{tar} , and train the network to minimize loss
- ▶ How to generate target Q-values Q_{tar} ?

The Bellman equation

- ▶ We saw that $Q^*(s, a)$ is the optimal action-value function:

$$Q^*(s, a) = \max_{\pi} Q_{\pi}(s, a) = Q^{\pi^*}(s, a)$$

- ▶ The **Bellman equation** lets us express $Q_*(s, a)$ recursively:
- ▶ If action a in state s always leads to state s' with reward r , then

$$Q^*(s, a) = r + \gamma \max_{a'} Q^*(s', a')$$

- ▶ If the environment is stochastic, $Q^*(s, a)$ will have a slightly more complex form:

$$Q^*(s, a) = \sum_{s', r} p(s', r | s, a) \left[r + \gamma \max_{a'} Q^*(s', a') \right]$$

Q-learning

- ▶ **Q-learning** is a **temporal difference algorithm** (Watkins, 1989)
- ▶ We saw that the Bellman equation is

$$Q^*(s, a) = r + \gamma \max_{a'} Q^*(s', a')$$

- ▶ Suppose that we are in a state s , take action a , receive a reward r and end up in state s' .
- ▶ We can use this information to compute a target Q-value:

$$Q_{\text{tar}}^{\pi}(s, a) = r + \gamma \max_{a'} \hat{Q}(s', a')$$

Q-learning: an off-policy algorithm

- ▶ A target Q-value is computed as

$$Q_{\text{tar}}^{\pi}(s, a) = r + \gamma \max_{a'} \hat{Q}(s', a')$$

- ▶ The target Q-value doesn't depend on the policy we are using to generate actions, so Q-learning is an **off-policy** algorithm
- ▶ Advantage: we can learn from any recorded experiences, and even use a recorded experience multiple times for learning
- ▶ Disadvantage: off-policy algorithms tend to be less stable

Q-learning: action selection

- ▶ We must gather experiences for learning using some policy
- ▶ One idea: act **greedily** with respect to estimated Q-values
 - ▶ Select the action with the highest Q-value
 - ▶ Only works in discrete action spaces
- ▶ However, action values are randomly initialized, so good actions may have low Q-values at first
- ▶ Problem: the agent might never try some actions at all!

Q-learning: exploration and exploitation

- ▶ We can use an ϵ -greedy policy
 - ▶ Select greedy action with probability $1 - \epsilon$
 - ▶ Select a random action with probability ϵ
- ▶ This is the **exploration-exploitation tradeoff** we saw in MCTS
- ▶ Typically we start with a high value of ϵ , decay over time

Tabular Q-learning

- ▶ Suppose there are only a small number of states and actions
 - ▶ Example: game of blackjack has only 200 states, 400 state-action pairs
- ▶ We can use a table to store a value $Q(s, a)$ for each state/action pair

- 1 Algorithm parameters: learning rate α , small $\epsilon > 0$
- 2 Initialize $Q(s_t, *) = 0$ where s_t is the terminal state
- 3 Initialize $Q(s, a)$ arbitrarily for other states and actions
- 4 Loop for each episode:
 - 5 $s =$ start state
 - 6 **while** s is not terminal:
 - 7 Choose action a from s using Q with ϵ -greedy policy
 - 8 Take action a , observe reward r , next state s'
 - 9 $Q(s, a) = Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - q(s, a)]$
 - 10 $s = s'$

Deep Q-learning with a neural network

- ▶ Many environments have far too many states to store in a table
- ▶ An agent must be able to handle state-action pairs it has never seen before!
- ▶ A neural network π_θ parameterized by a weight vector θ can perform **non-linear function approximation**
- ▶ $Q^{\pi_\theta}(s, a)$ denotes the approximation to Q^* computed by the network
- ▶ Ideally it will be able to **generalize** from visited (s, a) pairs to similar states, even if unvisited

Deep Q-Networks: experience replay

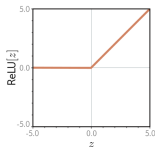
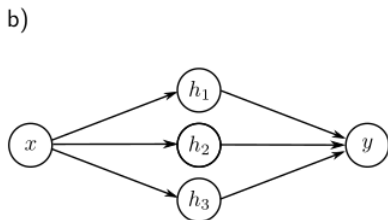
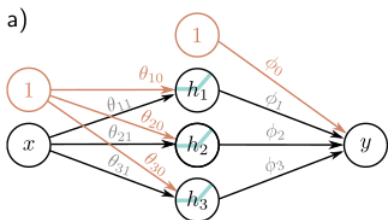
- ▶ If we learn only from a single episode at a time, experiences will be highly correlated
- ▶ In practice, this leads to high variance and slow learning
- ▶ We can use **experience replay** to learn from many past experiences at once
- ▶ We remember the k most recent experiences an agent has gathered
- ▶ In each training cycle, one or more **batches** are sampled uniformly from the experience memory

Deep Q-Networks: algorithm

```
1 for m = 1 to NUM_ITERATIONS do
2   Gather  $h$  experiences  $(s_i, a_i, r_i, s'_i)$  using Q with  $\epsilon$ -greedy policy
3   for b = 1 to NUM_BATCHES do
4     Sample a batch of experiences from experience memory
5     for u = 1 to NUM_UPDATES do
6       for i = 1 to BATCH_SIZE do
7         # Calculate a target Q-value for each example
8          $y_i = r_i + \gamma \max_{a'_i} Q^{\pi_\theta}(s'_i, a'_i)$ 
9       end for
10      # Calculate the mean-squared error loss
11       $L(\theta) = \frac{1}{N} \sum_i (y_i - Q^{\pi_\theta}(s_i, a_i))^2$ 
12      # Update the network's parameters
13       $\theta = \theta - \alpha \nabla_\theta L(\theta)$ 
14    end for
15  end for
16  Decay  $\epsilon$ 
17 end for
```

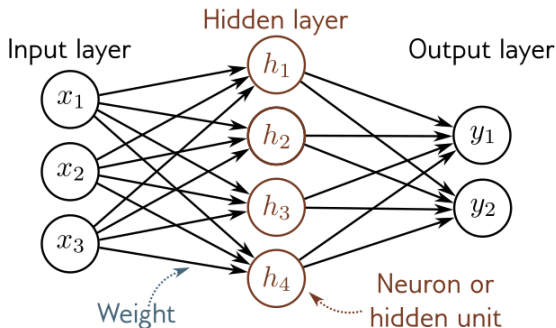
Neural network architecture

- ▶ Neural networks are composed of a series of **layers**
- ▶ A **dense** or **fully connected** layer computes an arbitrary linear function of multiple variables, then applies a non-linear **activation function** to each value
- ▶ Every connection has a **weight** θ_{ij} adjusted during training
- ▶ A common activation function is a rectified linear unit (ReLU)



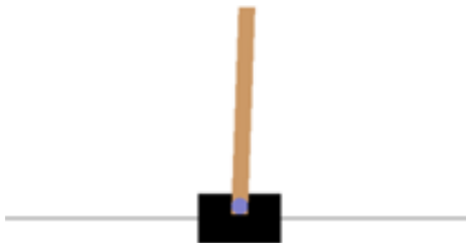
Shallow neural networks

- ▶ A network contains an **input layer**, one or more hidden layers, and an output layer
- ▶ If there is only one hidden layer, the network is **shallow**
- ▶ A **deep** network has multiple hidden layers



Deep Q-Networks: solving CartPole

- ▶ DQN with a shallow network is enough for simple environments such as CartPole
- ▶ Typical parameters:
 - ▶ One hidden layer of 64 units
 - ▶ Adam optimizer, learning rate of 0.01 decaying to 0
 - ▶ At each training step, sample 4 batches, each with 32 experiences
 - ▶ Store 10,000 recent experiences in replay memory
 - ▶ Train for 10,000 time steps



Improving DQN: target networks

- ▶ Problem: simple DQN with a deep network can be unstable
- ▶ $Q_{tar}^{\pi}(s, a)$ constantly changes because it depends on $\hat{Q}_{\pi}(s, a)$
- ▶ We can use a **target network** (Mnih et al, 2015) to stabilize training
- ▶ A target network π_{ϕ} is a **lagged copy** of the network π_{θ}
- ▶ Previously:

$$Q_{tar}^{\pi_{\theta}}(s, a) = r + \gamma \max_{a'} Q^{\pi_{\theta}}(s', a')$$

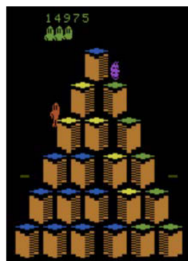
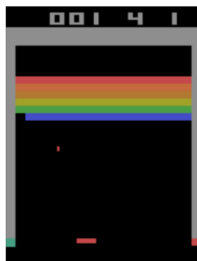
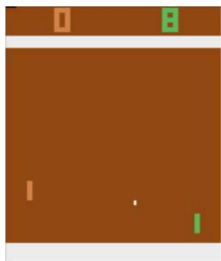
- ▶ Now:

$$Q_{tar}^{\pi_{\theta}}(s, a) = r + \gamma \max_{a'} Q^{\pi_{\phi}}(s', a')$$

- ▶ Periodically we update ϕ to the current values for θ (a **replacement update**)

Atari games

- ▶ Atari 2600 was a home game console, released in 1977
- ▶ Hundreds of interesting games, low computational requirements
- ▶ A classic testbed for reinforcement learning algorithms
- ▶ Available through Gymnasium (formerly OpenAI Gym)



Atari 2600

- ▶ 8-bit 6507 processor running at 1.19 MHz
- ▶ Screen is 160 pixels wide, 210 pixels high
- ▶ 128 bytes of RAM!



Atari games using Gymnasium API

- ▶ State is an array of size (210, 160, 3)
- ▶ Action space is discrete, with 4-18 possible actions depending on game
 - ▶ e.g. in Pong 0 = no-op, 1 = fire, 2 = up, 3 = down
- ▶ Far more complex than CartPole: episodes last for thousands of time steps
- ▶ Can we learn to play these games using reinforcement learning from pixel data alone?

DQN for Atari games

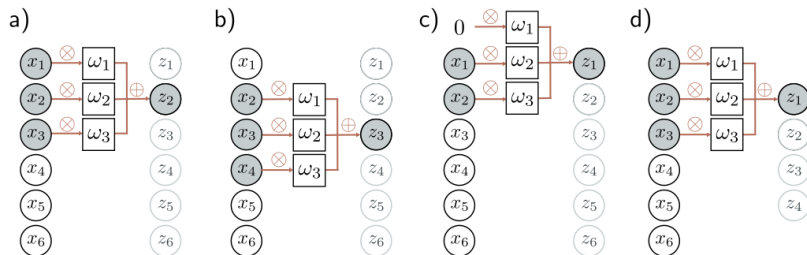
- ▶ In 2013-5 DeepMind showed that DQN can learn to play Atari games
- ▶ Target network to stabilize training
- ▶ **Deep network** with 3 hidden convolutional layers plus one hidden dense layer
- ▶ **State preprocessing**: downsizing, grayscaling, frame concatenation/skipping
- ▶ **Reward preprocessing**: rewards transformed to -1 , 0 , or $+1$
- ▶ **Environment reset** when a life is lost, depending on the game

Convolutional networks

- ▶ Images require a specialized module architecture, for several reasons:
- ▶ High-dimensional
 - ▶ A dense layer for $210 \cdot 160$ pixels would require over 1,000,000,000 weights!
- ▶ Nearby pixels are statistically related
- ▶ Interpretation is stable under geometric transformations
 - ▶ A tree shifted by a few pixels is still a tree, even if every pixel changes!
- ▶ **Convolutional layers** exploit spatial relationships between nearby pixels

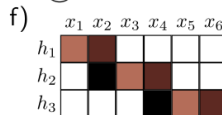
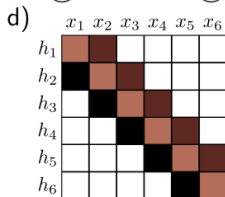
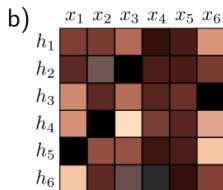
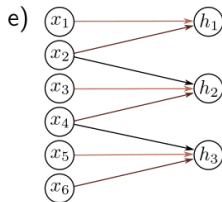
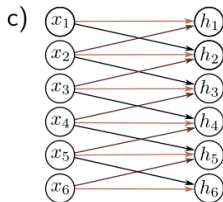
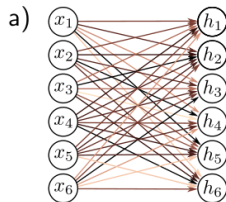
Convolutional networks for 1D inputs

- ▶ Same weights used at every position, collectively called the **kernel** or **filter**
- ▶ Below, the kernel size is 3
- ▶ At edges we may fill with zeros (c) or reduce output count (d)



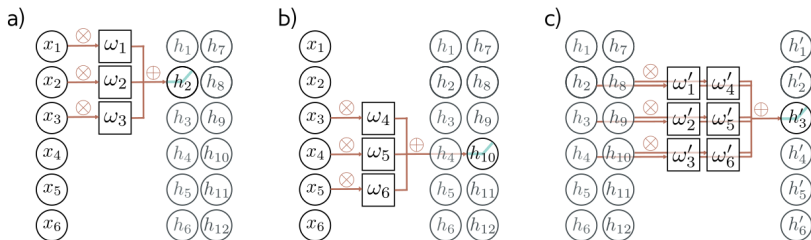
Fully connected versus convolutional layers

- ▶ A convolutional layer uses far fewer weights than a fully connected layer
- ▶ The kernel may shift by a **stride** that is greater than 1, further reducing weight count: stride = 2 in (e) below



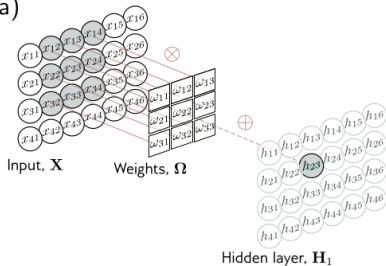
Channels

- ▶ A single convolution will inevitably lose information
- ▶ Typically we perform several convolutions in parallel, each producing a new set of hidden variables = **channel**

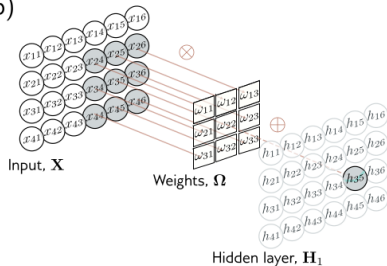


Convolutional networks for 2D inputs

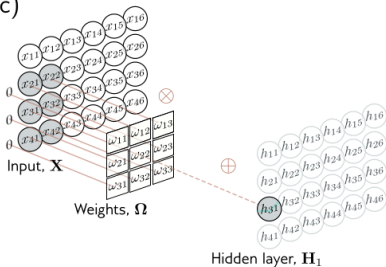
a)



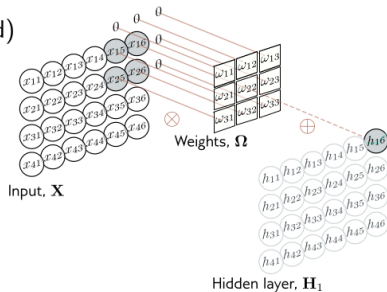
b)



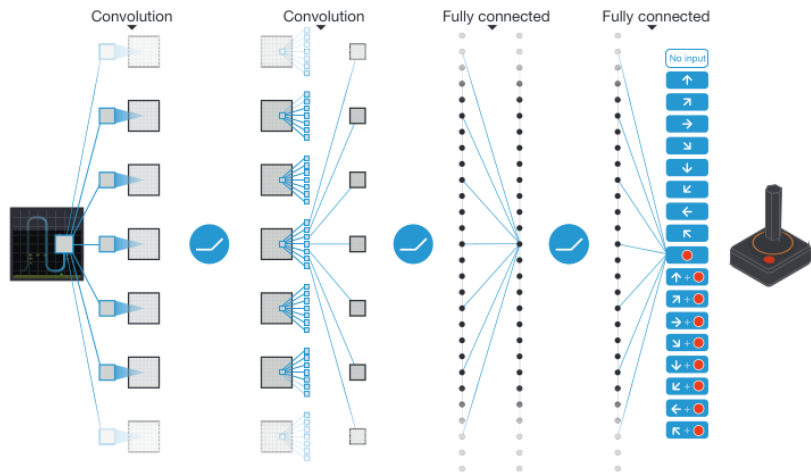
c)



d)



Network architecture for learning Atari games



DQN Network architecture for Atari: details

- ▶ Input: $84 \times 84 \times 4$ image (downscaled from original)
- ▶ Hidden layer 1: 32 filters of 8×8 with stride 4
- ▶ Hidden layer 2: 64 filters of 4×4 with stride 2
- ▶ Hidden layer 3: 64 filters of 3×3 with stride 1
- ▶ Final hidden layer: fully connected, 512 units
- ▶ Output layer: fully connected, 1 output for each valid action

More deep reinforcement learning algorithms

- ▶ DQN was a breakthrough, but was followed by more improvements/algorithms
- ▶ Double Q-Learning (van Hasselt et al, 2015)
- ▶ Prioritized Experience Replay for DQN (Schaul et al, 2015)
- ▶ PPO (Schulman et al, 2017)
- ▶ ...
- ▶ I recommend the implementations at CleanRL (<https://cleanrl.dev>)

AlphaZero: history

- ▶ 2016: AlphaGo
 - ▶ First Go program to achieve superhuman performance (defeated Lee Sedol 4-1)
 - ▶ MCTS + reinforcement learning + supervised learning from database of expert human moves
- ▶ 2017: AlphaGo Zero
 - ▶ Only MCTS + reinforcement learning; no human guidance beyond basic game rules
 - ▶ Significantly stronger than AlphaGo
- ▶ 2018: AlphaZero
 - ▶ Generalized to Go, chess, shogi, ...
 - ▶ Even stronger gameplay

Review: Learnable functions

- ▶ What can an agent learn?
 1. A **policy** π that maps states to actions: $a \sim \pi(s)$
 2. A **value function** $V^\pi(s)$ or $Q^\pi(s, a)$ that estimates $\mathbb{E}_{\tau \sim \pi}[R_t]$
 3. A **model** of the environment: $P(s' | s, a)$
- ▶ AlphaZero learns both a policy π and a value function $V^\pi(s)$
- ▶ It is given the model (the game rules), so it doesn't need to learn it

AlphaZero: policy improvement

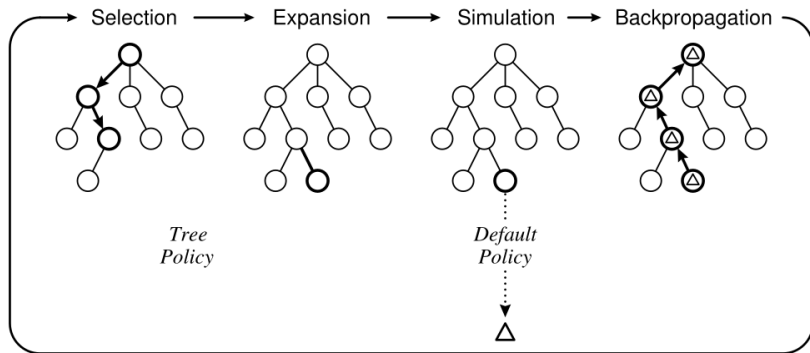
- ▶ Fundamental idea: we can use MCTS as a **policy improvement mechanism**
- ▶ If a neural network computes a policy π and value function V , we can use those to guide MCTS
- ▶ It will make moves that are stronger than π , and produce game outcomes that are more optimal than those predicted by V
- ▶ We can use those moves and game outcomes to train a neural network in a feedback loop!

AlphaZero: neural network

- ▶ A neural network f_θ is parameterized by weight vector θ
- ▶ Input: the board state s
- ▶ Output #1: estimated value $v_\theta(s) \in [-1, 1]$ from current player's perspective
- ▶ Output #2: policy $\vec{p}_\theta(s)$, a probability vector over possible actions
- ▶ We will use **self-play** with MCTS to generate training examples to improve our value and policy estimates

Monte Carlo Tree Search

- ▶ MCTS builds a game tree in memory, starting with an empty tree
- ▶ In classic MCTS each iteration has 4 phases: selection, expansion, simulation, backpropagation



MCTS in AlphaZero

In AlphaZero, each MCTS iteration has these phases:

1. **Selection** (also called **Simulation**): descend the tree, using node statistics **and the neural network policy function** to choose moves
2. **Expansion**: create a new child node, as in classic MCTS.
 - ▶ There is **no rollout**; instead, we use the **neural network value function** to compute an evaluation V of the new node
 - ▶ Or, in a terminal state we use the actual reward (+1 or -1)
3. **Backpropagation**: as in classic MCTS

AlphaZero: node state

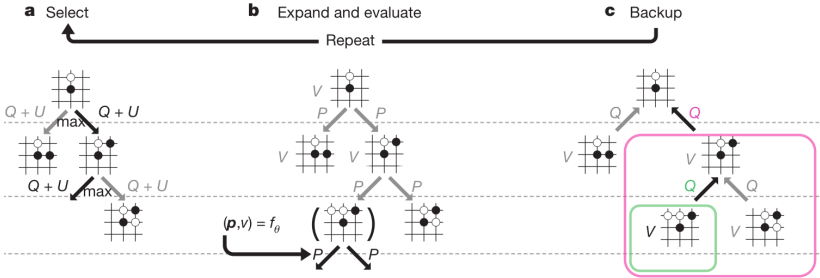
- ▶ For each state-action pair in the tree, we remember
 - ▶ $Q(s, a)$: the expected reward for taking action a from state s
 - ▶ $N(s, a)$: the number of times we took action a from state s during simulations
 - ▶ $P(s, a)$: the probability of selecting action a from state s according to the network policy $\vec{p}_\theta(s)$
- ▶ We use these to compute $U(s, a)$, an **upper confidence bound** on Q-values:

$$U(s, a) = C \cdot P(s, a) \cdot \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

- ▶ C is an exploration rate constant

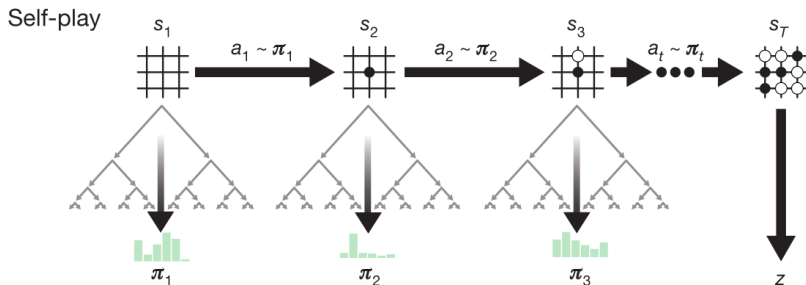
MCTS in AlphaZero

- ▶ Selection chooses moves that maximize $Q(s, a) + U(s, a)$
- ▶ Neural net produces move probabilities P and evaluations V
- ▶ Backpropagation updates Q to be the mean of all evaluations V below it in the tree



AlphaZero: self-play

- ▶ In each move, we perform a series of MCTS iterations
- ▶ Afterward, the $N(s, a)$ values at the root give us an improved stochastic policy $\vec{\pi}(s)$
- ▶ $\vec{\pi}(s)$ is the normalized counts $N(s, \cdot) / \sum_b (N(s, b))$
- ▶ We pick each move by sampling from $\vec{\pi}(s)$



AlphaZero: training the network

- ▶ Self-play games generate training examples $(s_i, \vec{\pi}_i, z_i)$
- ▶ $\vec{\pi}_i$ is the policy generated by MCTS
- ▶ $z_i \in \{-1, 1\}$ is the final game outcome (+1 if the player at s_i wins, -1 if they lose)
- ▶ We train the network to minimize the loss function

$$L = \sum_i (v_\theta(s_i) - z_i)^2 - \vec{\pi}_i \cdot \log(\vec{p}_\theta(s_i))$$

- ▶ $\vec{\pi}_i \cdot \log(\vec{p}_\theta(s_i))$ is a **cross entropy** term, commonly used in policy learning

AlphaZero: more details

- ▶ Network takes boards from last 7 time steps as input, since Go is not completely observable from current state
- ▶ Huge neural network
 - ▶ 19 residual blocks, each containing 2 convolutional layers
 - ▶ Each layer applies 256 filters of kernel size 3×3 , stride 1
- ▶ Lots of compute: network trained with 64 GPUs, 19 CPUs
- ▶ Compute power for self-play unclear from paper, but probably also huge