

Лекция 14

Reinforcement learning

Храбров Кузьма

1 декабря 2022 г.



План лекции

Что такое обучение с подкреплением

Многорукие бандиты

Марковский процесс принятия решений

Value iteration

Policy iteration

Q-learning

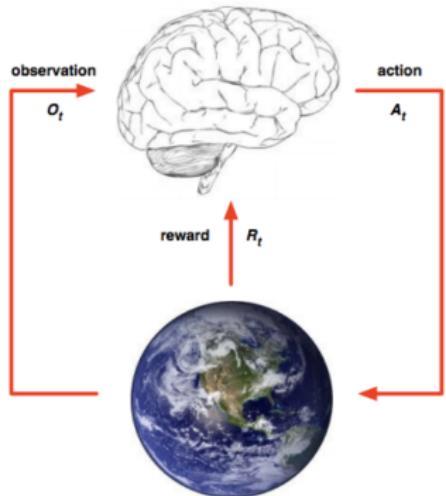
DQN

Policy gradients

Monte Carlo Tree Search

AlphaGo

Введение



► Агент (Actor)

- Получает награду R_t
- Получает наблюдение O_t
- Совершает действие A_t

► Среда (Environment)

- Получает действие A_t
- Генерирует состояние O_{t+1}
- Генерирует reward R_{t+1}

Примеры

- ▶ Настольные игры ($R = +1$ - победа, $R = -1$ - проигрыш)
- ▶ Управление роботом/машиной (+ за движение по траектории)
- ▶ Portfolio management
- ▶ Классические задачи с учителем

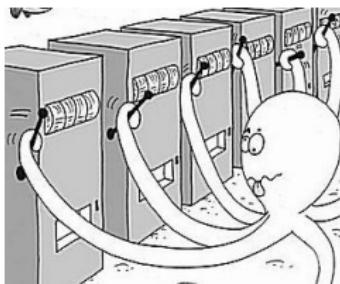
Основные понятия. Награда

- ▶ R_t - скаляр (случайная величина)
- ▶ Задача агента максимизировать среднюю сумму полученных R_τ

Definition (Reward hypothesis)

Любая задача может быть сформулирована в виде максимизации суммы R_τ

Многорукие бандиты



Рассмотрим задачу обучения с подкреплением, в которой состояние среды/агента от действия к действию не изменяется. Иначе говоря, у агента есть некоторый набор возможных действий, агент выбирает одно из них, получает за это некоторое вознаграждение (которое является случайной величиной), а затем снова может выбирать из тех же действий.

Многорукие бандиты. Формальное определение

Definition

Бернулиевской многорукий бандит это кортеж $(\mathcal{A}, \mathcal{R})$, где

- ▶ \mathcal{A} - набор действий $\{1, \dots, K\}$ соответствующих K -машинам с вероятностями награды $\{\theta_1, \dots, \theta_K\}$
- ▶ \mathcal{R} - функция награды. На шаге t :
 $\mathcal{R}(a_t) = r_t \sim Bernoulli(\theta_i)$, где $a_t = i$

Definition

Q -функция – ожидаемая награда от действия.

$$Q(a_t) = \mathbb{E}[r|a_t]$$

При условии выбора i -й ручки $Q(a) = \theta_i$.

$$Q^* = \max_i \theta_i$$

Многорукие бандиты. Жадное решение

Оценим каждую из θ_i методом Монте-Карло и будем все время дергать за наиболее ручку с самой большой оценкой $\hat{\theta}$.

Пусть $N_t(a)$ количество раз, сколько было выбрано действие a .

$$\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{\tau=1}^t r_\tau I[a_\tau = a]$$

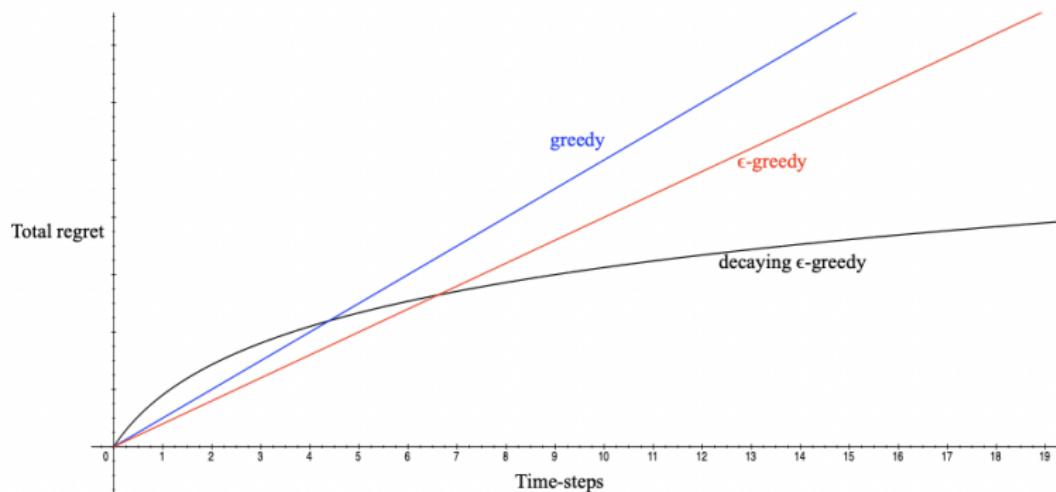
$$\hat{a}_t^* = \operatorname{argmax}_{a \in \mathcal{A}} \hat{Q}_t(a)$$

Проблема: существует ненулевая вероятность, что $\operatorname{argmax}_{a \in \mathcal{A}} \hat{Q}_t(a) \neq \operatorname{argmax}_{a \in \mathcal{A}} Q_t(a)$, то есть суммарная упущененная награда L_t (Regret) будет расти линейно с ростом t .

$$L_t = \mathbb{E} \sum_{\tau=1}^t [Q^* - Q(a_\tau)]$$

Многорукие бандиты. ϵ -Жадное решение

С вероятностью ϵ делаем случайный шаг, а с вероятностью $1 - \epsilon$ выберем argmax .



Многорукие бандиты. Sub-linear regret

Будем выбирать действия с максимальной оценкой успеха, но с поправкой на уверенность в этой оценке.

Алгоритм Upper Confidence Bound 1:

$$\hat{a}_t = \operatorname{argmax}_{a \in \mathcal{A}} Q(a) + \sqrt{\frac{2 \log t}{N_t(a)}}$$

Основные понятия. Состояние среды

В процессе взаимодействия со средой агент накапливает историю $H_t = R_1, O_1, A_1, \dots, R_t, O_t, A_t$. Часто для принятия решения хранение всей истории крайне избыточно:

- ▶ Games: O_t - скриншот (1200x700x3)
- ▶ Markets: оборот NYSE - 474m акций в день

Мы хотим иметь такое представление истории $S_t = f(H_t)$, которое было бы "достаточной статистикой" для будущего.

Марковское свойство

Мы хотим иметь такое представление истории $S_t = f(H_t)$, которое было бы "достаточной статистикой" для будущего.

Definition

Пусть S_t - последовательность случайных величин (векторов). Последовательность обладает марковским свойством, если

$$Pr(S_{t+1}|S_t) = Pr(S_{t+1}|S_t, S_{t-1}, \dots, S_1)$$

Т.е. S_t достаточно для предсказания будущих состояний.

Матрица переходов

Пусть S_t - последовательность дискретных состояний.

Поскольку последовательность задается распределением $Pr(S_{t+1}|S_t)$ естественно упорядочить его в матрицу.

$$\mathcal{P}_{ss'} = Pr(S_{t+1} = s' | S_t = s)$$

$$\mathcal{P} = \underset{\text{from}}{\left[\begin{array}{ccc} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{n1} & \dots & \mathcal{P}_{nn} \end{array} \right]} \underset{\text{to}}{}$$

Какие суммы вероятностей должны равняться единице?

Марковский процесс

Definition

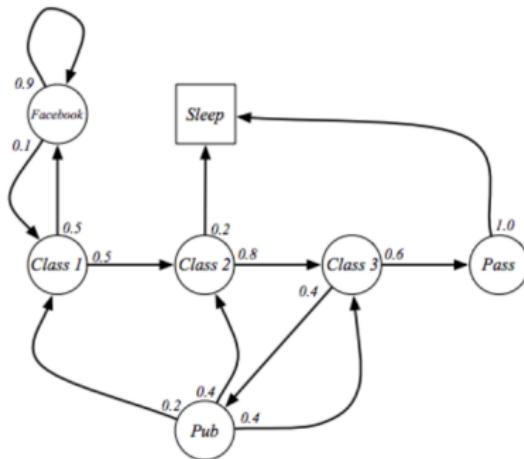
Марковский процесс (цепь) это кортеж (S, P) , где

- ▶ S - состояния (дискретное пространство)
- ▶ P - матрица переходов (transition matrix)

$$P_{ss'}^a = \Pr(S_{t+1} = s' | S_t = s, A_t = a)$$

Строго говоря необходимо еще распределение начальных состояний (но мы предполагаем, что оно вырождено, т.е. мы знаем где начинаем с вероятностью 1). Марковский процесс - основа для RL. Мы будем постепенно усложнять эту модель, добавляя rewards и actions.

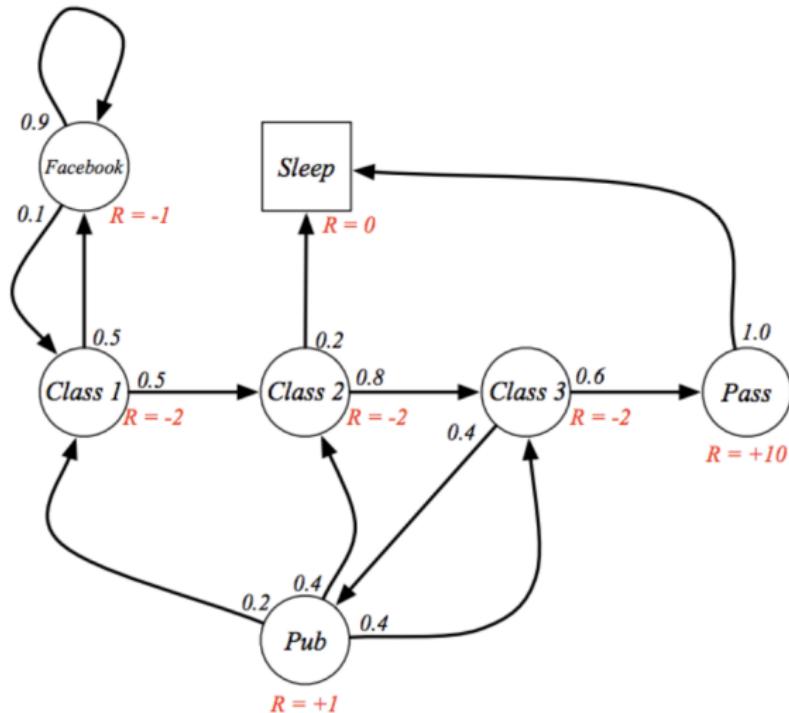
Пример



$$\mathcal{P} = \begin{bmatrix} & C1 & C2 & C3 & Pass & Pub & FB \\ C1 & 0.5 & 0.8 & 0.6 & 0.4 & & \\ C2 & & 0.2 & & & & \\ C3 & & & 0.4 & & & \\ Pass & 0.2 & 0.4 & 0.4 & & & \\ Pub & 0.1 & & & & & \\ FB & & & & & & \\ Sleep & & & & & & 0.9 \end{bmatrix}$$

Марковская цепь описывает блуждание по конечному (в нашем случае) пространству состояний, что не очень интересно. В нашем примере состояния не равнозначны, добавим ценность нахождения в каждом при помощи rewards.

Пример с наградами



Основная цель: максимизировать получаемые rewards.

Полная награда (Return)

Definition (Return)

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{i=0}^{\infty} \gamma^i R_{t+1+i}$$

Под G_t мы понимаем дисконтированную сумму всех будущих rewards.

- ▶ $\gamma \in [0, 1]$ - discount factor
- ▶ Единая форма для "конечных" и "бесконечных" моделей
- ▶ Обеспечивает сходимость ряда ($\max|R_t| < C = \text{const}$)
- ▶ Нетерпеливость (impatience) - насколько важно получить reward сейчас, чем потом.

По определению G_t - случайная величина (не привязанная ни к чему). Покажем, как ее можно использовать.

Value function

Definition (Value function)

$$v(s) = \mathbb{E}[G_t | S_t = s]$$

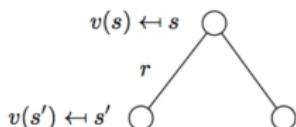
$v(s)$ - ценность состояния s (ожидаемая сумма всех полученных rewards, если стартовать из s).

$$\begin{aligned} v(s) &= \mathbb{E}[G_t | S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) | S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) | S_t = s] \end{aligned} \tag{1}$$

Bellman equations

Bellman equation для value-function

$$v(s) = \mathbb{E} [R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s]$$

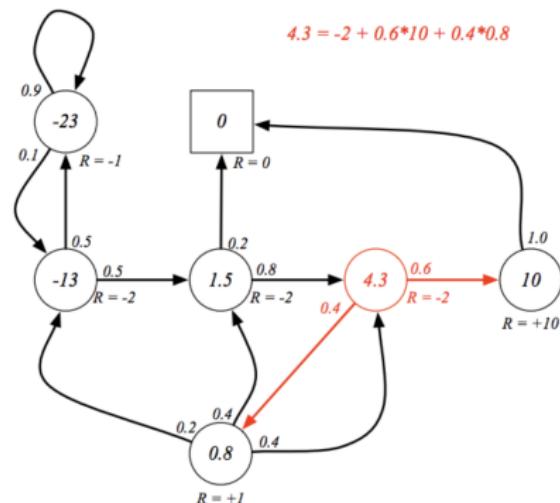


$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s')$$

Эти уравнения должны напоминать backprop (как минимум идеально). Из них очень легко найти value для любого состояния.

$$v = \mathcal{R} + \gamma \mathcal{P}v$$

Пример



$$4.3 = -2 + 0.6*10 + 0.4*0.8$$

Марковский процесс с наградами

Definition

MRP это кортеж (S, R, P, γ) , где

- ▶ S - состояния (дискретное пространство)
- ▶ R - функция rewards, $R_s = \mathbb{E}[R_{t+1}|S_t = s]$
- ▶ P - матрица переходов (transition matrix)
 $P_{ss'} = Pr(S_{t+1} = s'|S_t = s)$
- ▶ γ - discount factor

Марковский процесс принятия решений

Definition

MDP это кортеж (S, A, R, P, γ) , где

- ▶ S - состояния (дискретное пространство)
- ▶ A - действия (дискретное пространство)
- ▶ R - функция rewards, $R_s^a = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$
- ▶ P - матрица переходов (transition matrix)
 $P_{ss'}^a = Pr(S_{t+1} = s' | S_t = s, A_t = a)$
- ▶ γ - discount factor

Стратегия(политика) и Q-функция

Definition (Policy)

$\pi(a|s) = Pr(A_t = a|S_t = s)$ - стратегия, т.е. то как мы выбираем действия оказавшись в состоянии s .

Definition (Value function)

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

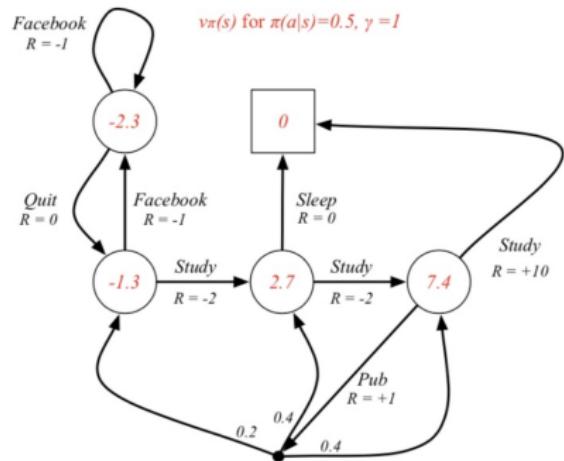
Definition (Q-function)

$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$ - ценность действия в состоянии s .

Совместное распределение

$$\begin{aligned} P_{ss'} &= \Pr(S_{t+1} = s' | S_t = s) \\ &= \sum_a \Pr(S_{t+1} = s', A_t = a | S_t = s) \\ &= \sum_a \Pr(A_t = a | S_t = s) \Pr(S_{t+1} = s' | S_t = s, A_t = a) \quad (2) \\ &= \sum_a \pi(a|s) P_{ss'}^a \end{aligned}$$

Пример



Bellman equations

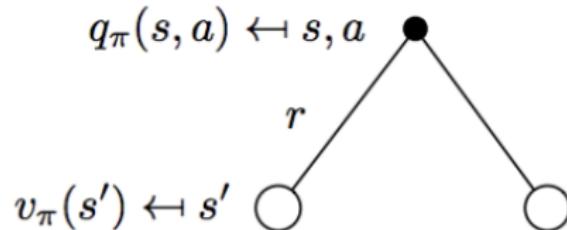
Bellman equation для value-function

$$v_{\pi}(s) = \mathbb{E}_{\pi} [R_{t+1} + \gamma v_{\pi}(S_{t+1}) \mid S_t = s]$$

Bellman equation для q-function

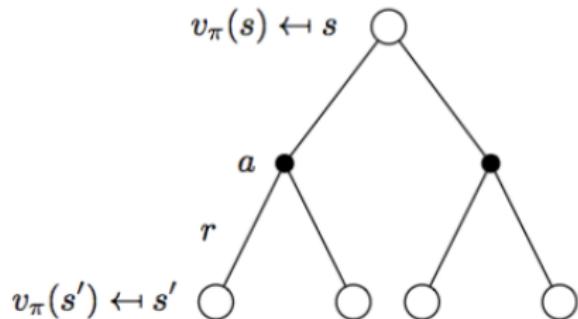
$$q_{\pi}(s, a) = \mathbb{E}_{\pi} [R_{t+1} + \gamma q_{\pi}(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a]$$

Q-функция в свою очередь зависит от value-function следующего состояния



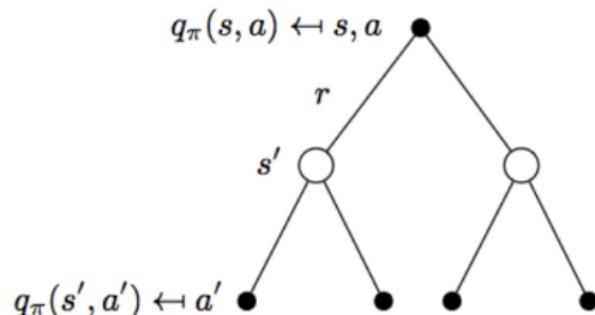
$$q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_\pi(s')$$

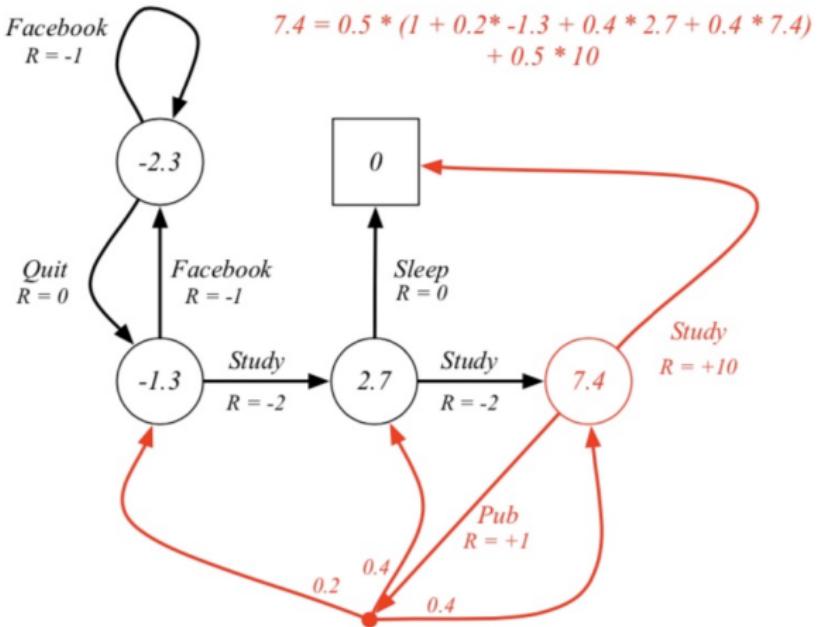
Соберем две предыдущие картинки вместе



$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

Точно так же можно вывести зависимость q-function





Moar Bellman equations

Зафиксируем некоторую policy. Возьмем уравнение Беллмана в случае MRP

$$v = \mathcal{R} + \gamma \mathcal{P} v$$

Возьмем уравнение Беллмана в случае MDP

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

И сделаем замену:

$$\mathcal{P}_{s,s'}^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}_{ss'}^a$$

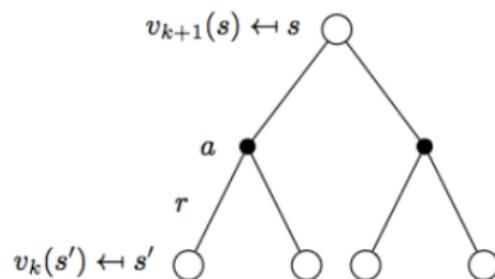
$$\mathcal{R}_s^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}_s^a$$

Алгоритм Value Evaluation

Получим следующее уравнение:

$$v_{\pi} = \mathcal{R}^{\pi} + \gamma \mathcal{P}^{\pi} v_{\pi}$$

И будем решать методом итераций:

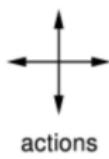


$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$
$$\mathbf{v}^{k+1} = \mathcal{R}^{\pi} + \gamma \mathcal{P}^{\pi} \mathbf{v}^k$$

Результаты

Можно показать, что этот процесс сходится и в результате для каждого состояния мы получаем его $v(s)$. Теперь зададимся вопросом, а как улучшить нашу policy?

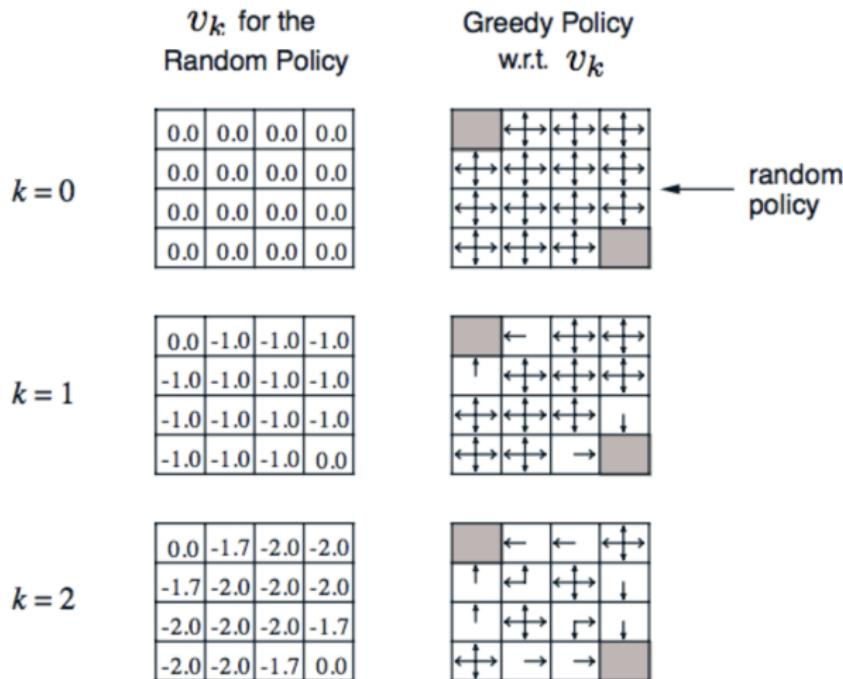
Игра



		-	
4	5	6	7
8	9	10	11
12	13	14	

$r = -1$
on all transitions

Iteration 1



Iteration 2

$k = 3$

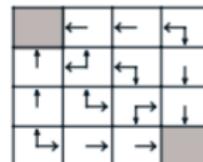
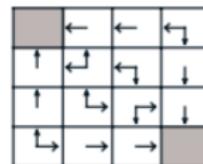
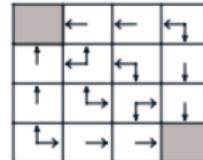
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

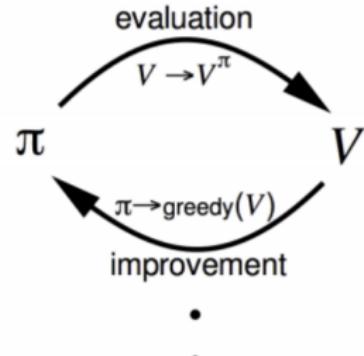
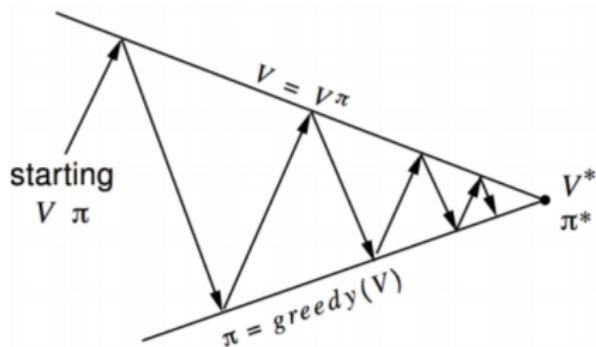
0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0



optimal
policy

Алгоритм Policy iteration

Для того чтобы улучшить policy надо выбирать действия, которые приводят в более выгодные состояния. Полученный алгоритм называется policy iteration.



Оптимальные value-functions

Оптимальная q/value-function - максимальное принимаемое значение

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

Как только мы знаем Q^* мы можем выбрать оптимальную стратегию

$$\pi^*(s) = \operatorname{argmax}_a Q^*(s, a)$$

Оптимальное значение - максимум по всем принимаемым решениям:

$$\begin{aligned} Q^*(s, a) &= r_{t+1} + \gamma \max_{a_{t+1}} r_{t+2} + \gamma^2 \max_{a_{t+2}} r_{t+3} + \dots \\ &= r_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \end{aligned}$$

Соответствующее уравнение Беллмана:

$$Q^*(s, a) = \mathbb{E}_{s'} \left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a \right]$$

Q-learning

Будем решать уравнение Беллмана:

$$Q^*(s, a) = \mathbb{E}_{s'} \left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a \right]$$

методом конечных приращений, то есть будем повторять:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a'))$$

где α - learning rate. Обычно берут $\alpha = 0.9$.

Упражнение: покажите, что таким образом минимизируется квадрат разности.

Подходы к обучению с подкреплением

1. Value-based RL

Оцениваем оптимальную Q -функцию $Q^*(s, a)$.

Максимальное значение принимаемое при любой стратегии.

2. Policy-based RL

Ищем оптимальную стратегию π^* . Стратегия обеспечивающая максимальное будущее вознаграждение.

3. Model-based RL

Строим и используем модель внешней среды.

Deep Q-learning

В каждом подходе к RL можно применить нейронные сети.

Разберем подробно Value-based случай.

Будем аппроксимировать $Q(s, a) = Q(s, a, w)$ с помощью нейронной сети с весами w . Тогда для решения соответствующего уравнения Беллмана можно минимизировать функцию потерь MSE

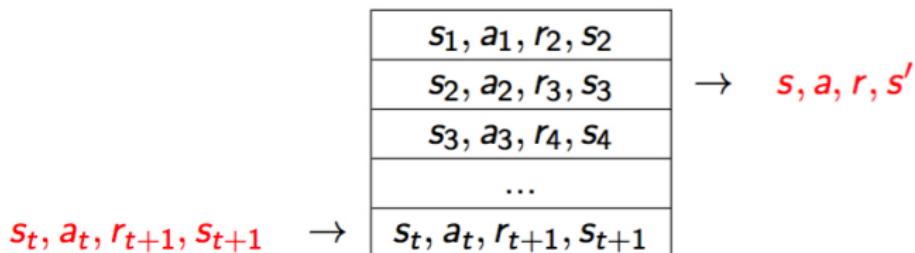
$$l = \left(r + \gamma \max_a Q(s', a', w) - Q(s, a, w) \right)^2$$

Проблемы:

1. Корреляции между входами
2. Нестационарные целевые переменные

DQN-2

Чтобы убрать корреляции в данных: используем опыт агента (Replay memory)

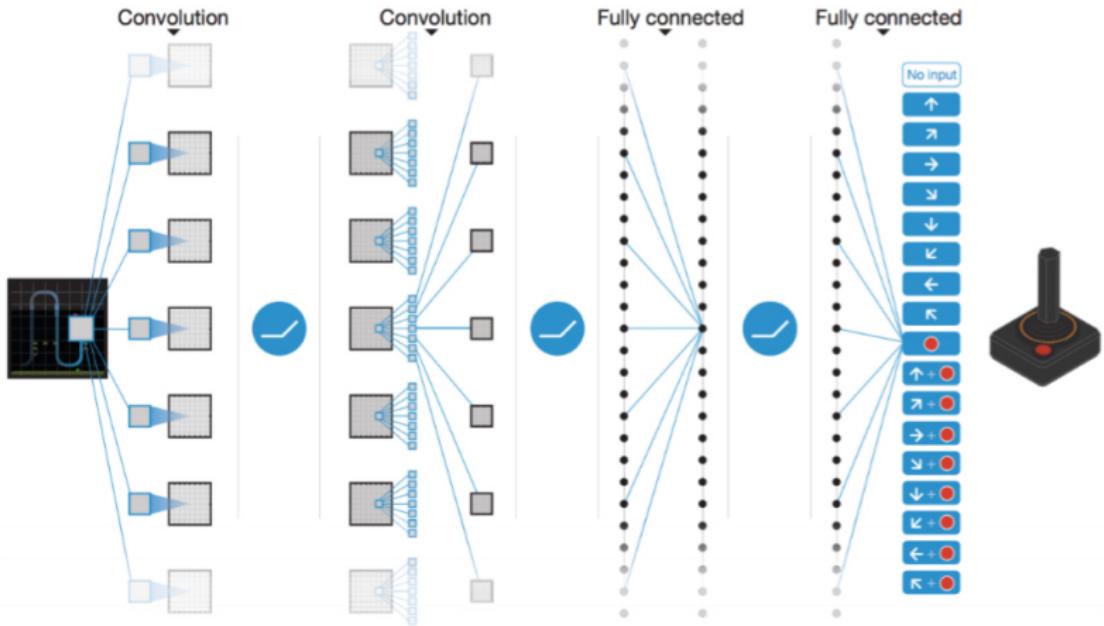


Сэмплируем опыт из данных и обновляем

$$l = \left(r + \gamma \max_a Q(s', a', w^-) - Q(s, a, w) \right)^2$$

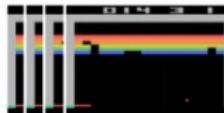
Причем w^- оставляем фиксированными, чтобы убрать нестационарность.

ATARI



ATARI

$Q(s, a; \theta)$ нейросеть
с весами θ



Текущее состояние s_t : 84x84x4 стек последних четырех
фреймов
(RGB->grayscale конвертирование, даунсамплинг, и кроп)

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

ATARI-improvements

1. Double DQN <https://arxiv.org/abs/1509.06461>
2. Prioritised experience replay
<https://arxiv.org/abs/1511.05952>
3. Duelling network <https://arxiv.org/abs/1511.06581>

Policy gradients.

В чем проблема Q-learning?

Функция оценки ценности действия может быть очень сложной.

Пример: движение манипулятора робота для захвата может быть многомерной проблемой → тяжело выучить точные значения ценности для каждой пары (состояние, действие).

Политика может быть более простой: возьми объект недалеко от себя.

Можно ли выучить политику напрямую, т.е. выбирать лучшую политику из набора политик?

Policy gradients. Идея

- ▶ Функция потерь для обучения с учителем:

$$\sum_i \log p(y_i|x_i)$$

- ▶ Для обучения с подкреплением мы:
 - ▶ не имеем правильных меток только псевдометки - действия.
 - ▶ моделируем лосс как $\sum_i A_i \log p(y_i|x_i)$, где A – функция выгоды. Например, $A = 1$ для выигранных партий и -1 для проигранных.
- ▶ Мы максимизируем вероятность тех действий, которые ведут к выигрышу.

Policy gradients

Используем gradient accent для оптимизации стратегии.

Более строго хотим максимизировать функцию:

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}[R_\tau], \text{ где } \tau - \text{ траектория длины } T, \text{ а}$$
$$R_\tau = \sum_{\tau} R(s_t, a_t)$$

Policy : π_θ

Objective function : $J(\theta)$

Gradient : $\nabla_\theta J(\theta)$

Update : $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \mathbb{E}_{\pi_\theta} \left[\underbrace{\sum_t R(s_t, a_t)}_{J(\theta)} \right]$$

Policy gradients

Как посчитать производную?

Вероятность τ : $p_\theta(\tau) = p_\theta(s_1, a_1, \dots, s_T, a_T) = p(s_1) \prod_{t=1}^T \pi_\theta(a_t | s_t) p(s_{t+1} | s_t, a_t)$

log-derivative trick: $p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) = p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = \nabla_\theta p_\theta(\tau)$

$$\log p_\theta(\tau) = \log(p(s_1) \prod_{t=1}^T \pi_\theta(a_t | s_t) p(s_{t+1} | s_t, a_t)) = \log p(s_1) + \sum_{t=1}^T (\log \pi_\theta(a_t | s_t) + \log p(s_{t+1} | s_t, a_t))$$

$$\nabla_\theta \log p_\theta(\tau) = \underbrace{\nabla_\theta \log p(s_1)}_{=0} + \sum_{t=1}^T \left(\nabla_\theta \log \pi_\theta(a_t | s_t) + \underbrace{\nabla_\theta \log p(s_{t+1} | s_t, a_t)}_{=0} \right) = \sum_{t=1}^T \nabla_\theta \log \pi_\theta(a_t | s_t)$$

$$\nabla_\theta J(\theta) = E_\pi [\nabla_\theta (\underbrace{\log \pi(\tau | \theta)}_{\text{Policy function}}) \underbrace{R(\tau)}_{\text{Score function}}]$$

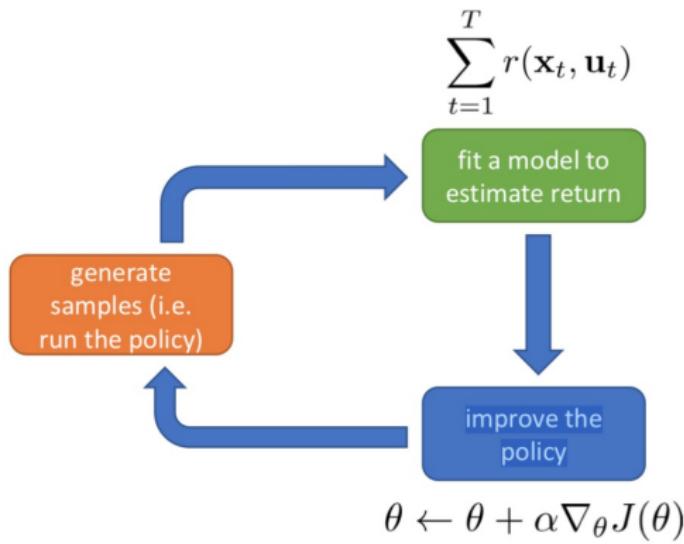
Policy gradients

$$\text{Policy gradient} : E_{\pi}[\underbrace{\nabla_{\theta}(\log \pi(s, a, \theta))}_{\text{Policy function}} \underbrace{R(\tau)}_{\text{Score function}}]$$

$$\text{Update rule} : \Delta \theta = \alpha * \nabla_{\theta}(\log \pi(s, a, \theta)) R(\tau)$$

Change in parameters Learning rate

Reinforce



Reinforce. Проблемы

- ▶ Функция выгоды обладает большой дисперсией.
- ▶ Один из способов уменьшения – сдвиг значений функции к нулевому среднему и единичной дисперсии
- ▶ Нужно знать распределение функции, что не всегда достигается

Baseline

Заметим, что если к функции $R(t)$ добавить константу, то матожидание не изменится, зато можно уменьшить дисперсию. Будем использовать для каждого состояния будущий выигрыш вместо полного выигрыша.

$$\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

Алгоритм Actor-Critic

$$A^{\pi_\theta}(s_t, a_t) = Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t) \text{ - функция выгоды}$$

- Как ее найти? Используем Q-learning для оценки $Q^{\pi_\theta}(s_t, a_t)$ и $V^{\pi_\theta}(s_t)$ - критик. Актер – Policy Gradients для оценки политики.
 - Актер решает какое действие выбрать, а критик говорит насколько хорошим это действие было и как его стоит подправить
 - Работа критика облегчается, поскольку оцениваются пары действие-состояние генерируемых политикой
 - Можно добавить “experience replay”

Алгоритм Actor-Critic

```
Initialize policy parameters  $\theta$ , critic parameters  $\phi$ 
For iteration=1, 2 ... do
    Sample m trajectories under the current policy
     $\Delta\theta \leftarrow 0$ 
    For i=1, ..., m do
        For t=1, ..., T do
             $A_t = \sum_{t' \geq t} \gamma^{t'-t} r_t^i - V_\phi(s_t^i)$ 
             $\Delta\theta \leftarrow \Delta\theta + A_t \nabla_\theta \log(a_t^i | s_t^i)$ 
             $\Delta\phi \leftarrow \sum_i \sum_t \nabla_\phi ||A_t^i||^2$ 
             $\theta \leftarrow \alpha \Delta\theta$ 
             $\phi \leftarrow \beta \Delta\phi$ 
    End for
```

Recurrent Attention Model (RAM)

Цель: Классификация изображения

Принимая последовательность патчей (“glimpses”) выборочно фокусироваться на регионе изображения для предсказания класса

Наблюдения за тем, как исследует изображение человек

Экономим вычислительные ресурсы

Способность игнорировать нерелевантные части изображения

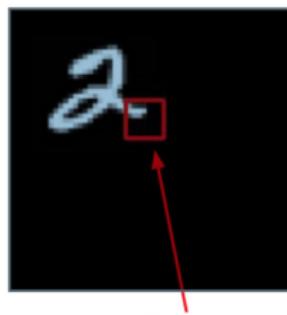
Состояния: Патчи (glimpses)

Действия: (x, y) координаты (центра патча) откуда получить следующую картинку изображения

Награда: 1 если добрались до корректной классификации картинки на финальном шаге, 0 если нет

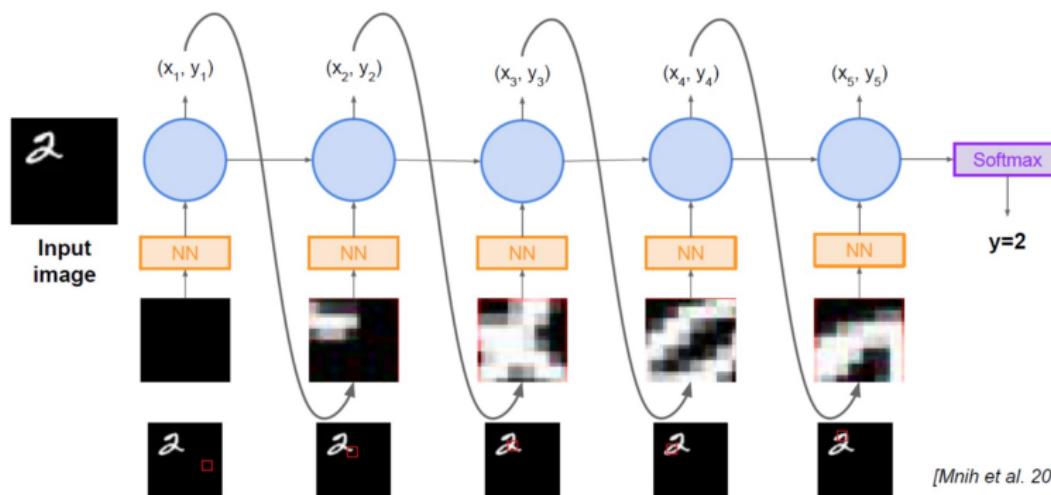
Патчи операция не дифференцируемая => обучаем политику как семплировать патчи, используя Policy Gradients (REINFORCE)

Для задания состояния отбираемых патчей используем RNN для моделирования состояния и предсказания следующего действия.



Actor critic

Recurrent Attention Model (RAM)



[Mnih et al. 2014]

Actor critic. Модификации

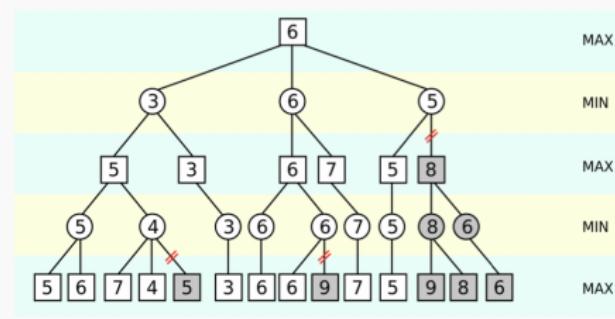
- ▶ Хотим использовать модификации из DQN. Off-Policy Policy Gradients: используем дополнительные "замороженные" стратегии, чтобы использовать Experience replay (<https://arxiv.org/abs/2002.04014>)
- ▶ A3C, A2C: Asynchronous Advantage Actor-Critic, вместо одного агента - N агентов обучаю параметры стратегии и функции полезности.
(<https://arxiv.org/pdf/1602.01783.pdf>)
- ▶ Deterministic policy gradient (DPG) / Deep Deterministic policy gradient DDPG: строим детерминированную функцию $a = \mu(s)$.
(<http://proceedings.mlr.press/v32/silver14.pdf>,
<https://arxiv.org/abs/1509.02971>)

Actor critic. Модификации

- ▶ TRPO: Trust region policy optimization. Не даем сильно менять стратегию (регуляризация с помощью KL-дивергенции). (<https://arxiv.org/abs/1502.05477>)
- ▶ SAC: Soft Actor-Critic. Дополнительно максимизируем энтропию стратегии.
(<https://arxiv.org/abs/1801.01290>)

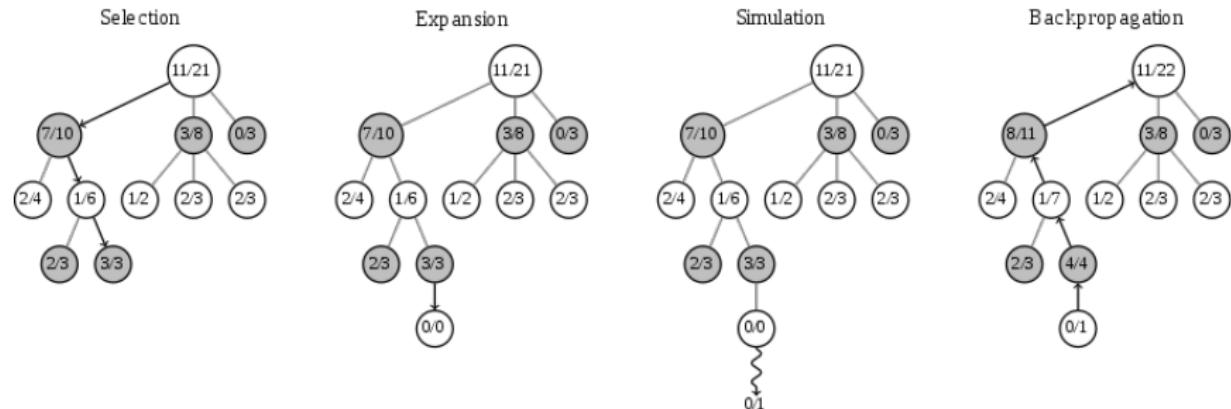
Min-Max strategy

Alpha-beta search: строим минимакс-дерево ходов, выкидываем ходы, которые заведомо хуже других, ищем в глубину.



- ▶ Шахматы 30-40 возможных ходов + эффективные эвристики для отсечки плохих ходов
 - ▶ В игре Го 250 возможных ходов + в среднем 100 относительно хороших ходов

Monte Carlo Tree Search (MCTS)



Monte Carlo Tree Search (MCTS)

1. Selection (Выбор)

Начиная с корня дерева выбираем двигаемся по ветвям, выбирая путь согласно ($UCB1$) пока не доберемся до листа L .

2. Expansion (Расширение)

Если L - не терминальное состояние, делаем еще одно действие добавляя узел C .

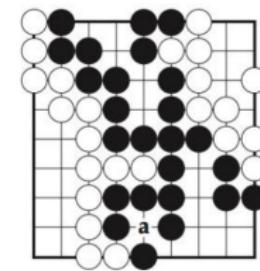
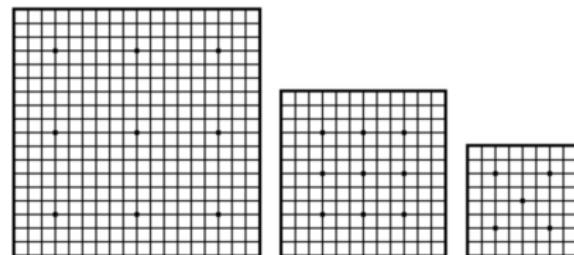
3. Simulation (Симуляция)

Запускаем симуляцию из C до окончания игры.

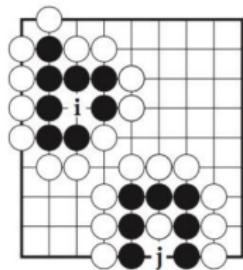
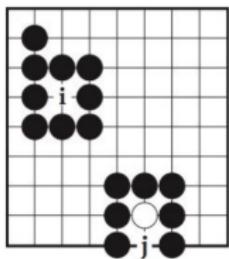
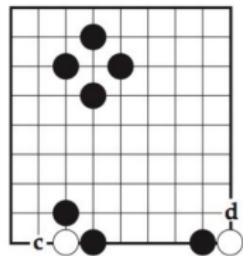
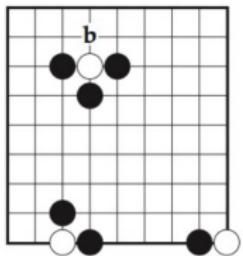
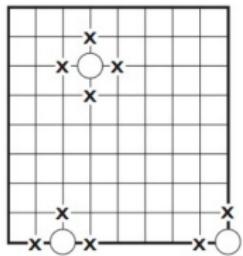
4. Backpropagation

Обновляем информацию во всех узлах дерева.

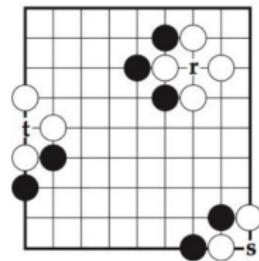
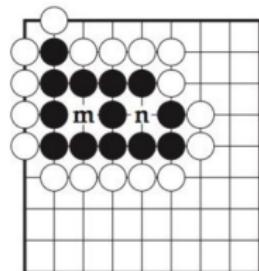
Игра ГО



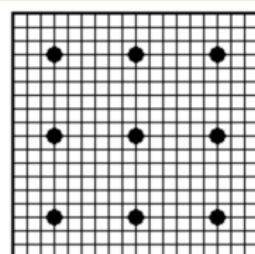
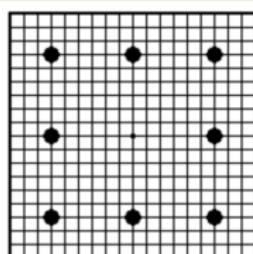
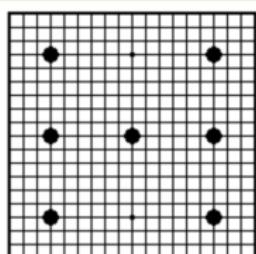
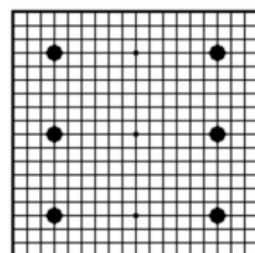
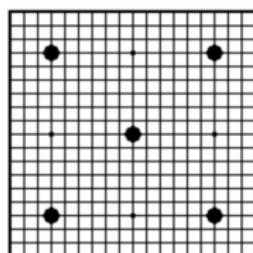
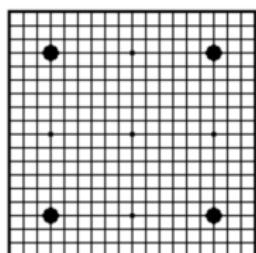
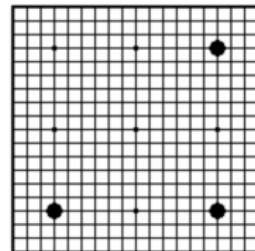
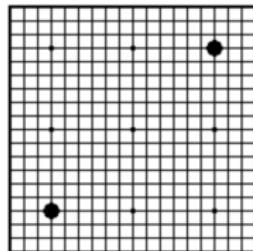
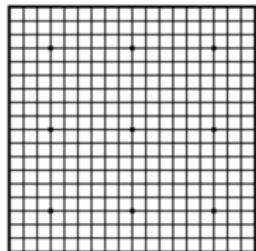
Игра ГО



Игра ГО

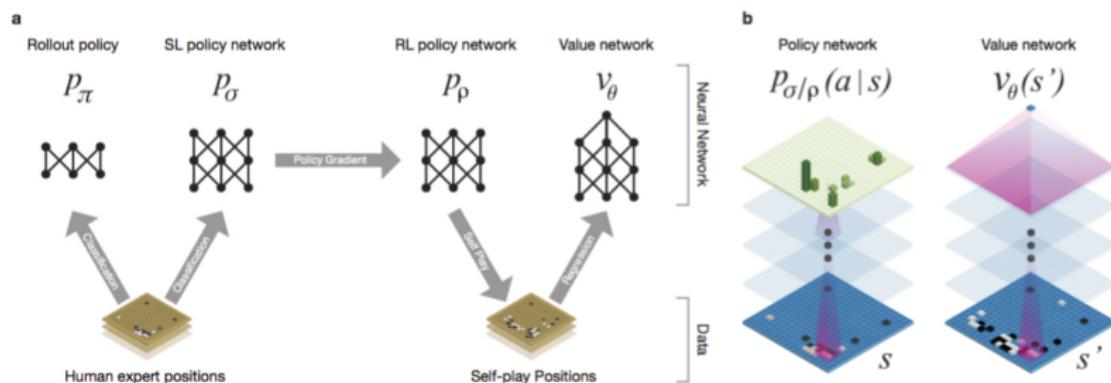


Игра ГО



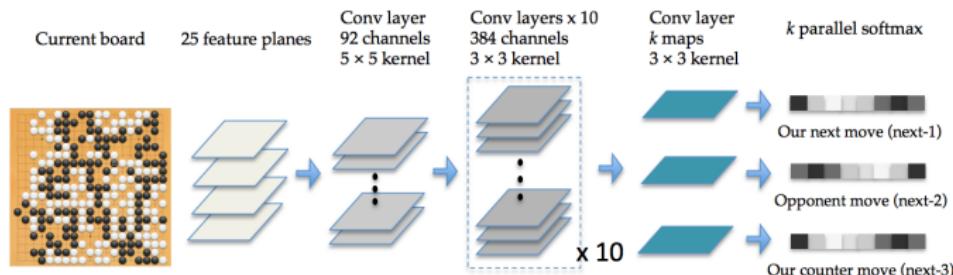
AlphaGo

Ключевая идея алгоритма: будем приближать value-function и policy с помощью двух различных нейросетей, а играть и получать новую информацию с помощью MCTS. Плюс обучим классификаторы шагов на известных партиях экспертов.



AlphaGo. SL

Большая сеть p_σ обучена на позициях из KGS data set (29.4 миллиона позиций из 160000 игр сыгранных игроками от бго до 9го дана). Состояние системы - положение камней на доске, действие - ход игрока. Вход в сеть - onehot-encoding состояния. Маленькие сети p_π и p_τ (Rollout Policy и Tree Policy) - линейный классификатор, обученный на локальных состояниях (3×3) игровой доски вокруг следующего и предыдущего действий + множества других признаков.



AlphaGo. SL

Feature	# of planes	Description
Stone colour	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (empty adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether a move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

Extended Data Table 2: **Input features for neural networks.** Feature planes used by the policy network (all but last feature) and value netwkr (all features).

Feature	# of patterns	Description
Response	1	Whether move matches one or more response features
Save atari	1	Move saves stone(s) from capture
Neighbour	8	Move is 8-connected to previous move
Nakade	8192	Move matches a <i>nakade</i> pattern at captured stone
Response pattern	32207	Move matches 12-point diamond pattern near previous move
Non-response pattern	69338	Move matches 3 × 3 pattern around move
Self-atari	1	Move allows stones to be captured
Last move distance	34	Manhattan distance to previous two moves
Non-response pattern	32207	Move matches 12-point diamond pattern centred around move

Extended Data Table 4: **Input features for rollout and tree policy.** Features used by the rollout policy (first set) and tree policy (first and second set). Patterns are based on stone colour (black/white/empty) and liberties (1, 2, ≥ 3) at each intersection of the pattern.

AlphaGo. RL Policy and Value Networks

1. Тренируем сеть p_ρ аппроксимировать оптимальную policy, инициализировав веса с помощью p_σ .

Каждый батч обучения: n игр со старыми версиями policy $p_{p_}$.

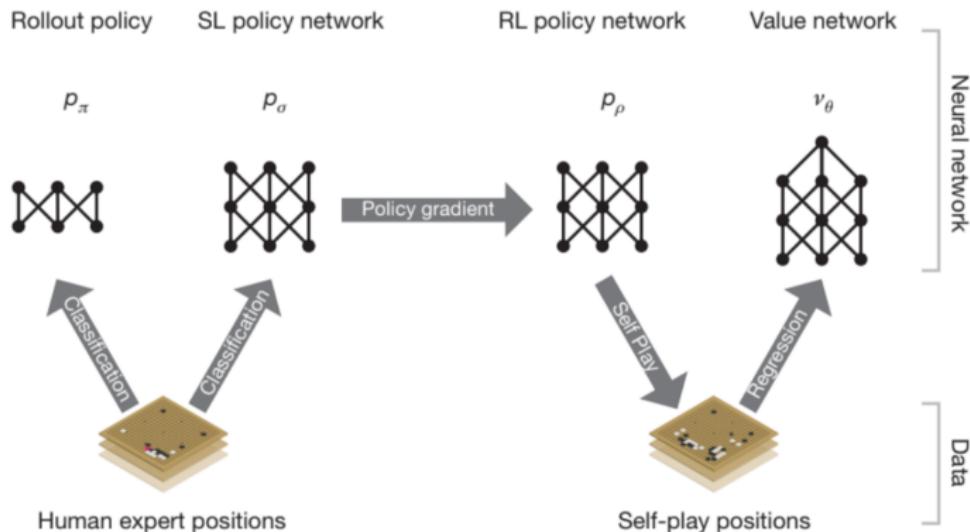
$$\Delta \rho = \frac{\alpha}{n} \sum_{i=1}^n \sum_{t=1}^{T^i} \frac{\partial \log p_\rho(a_t^i | s_t^i)}{\partial \rho} (z_t^i - v(s_t^i))$$

2. Тренируем сеть $v_\theta(s)$ аппроксимировать value-function.

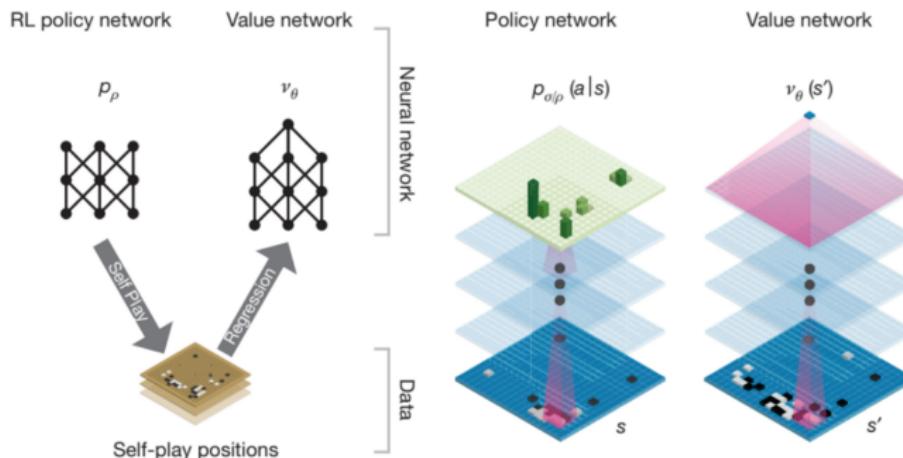
Обучающая выборка - случайный набор позиций из "игр с собой".

$$\Delta \theta = \frac{\alpha}{m} \sum_{k=1}^m (z^k - v_\theta(s^k)) \frac{\partial v_\theta(s^k)}{\partial \theta}$$

AlphaGo. RL Policy and Value Networks



AlphaGo. RL Policy and Value Networks



AlphaGo. MCTS

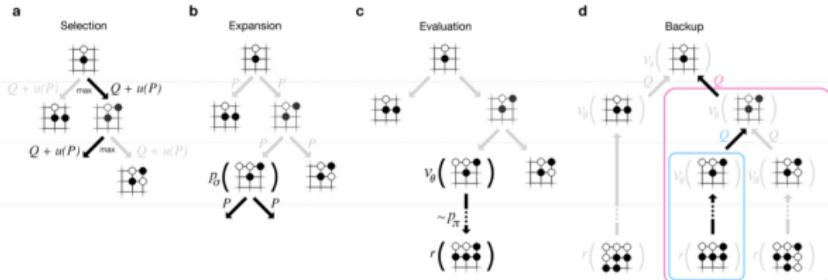


Figure 3: Monte-Carlo tree search in *AlphaGo*. **a** Each simulation traverses the tree by selecting the edge with maximum action-value Q , plus a bonus $u(P)$ that depends on a stored prior probability P for that edge. **b** The leaf node may be expanded; the new node is processed once by the policy network p_σ and the output probabilities are stored as prior probabilities P for each action. **c** At the end of a simulation, the leaf node is evaluated in two ways: using the value network v_θ ; and by running a rollout to the end of the game with the fast rollout policy p_π , then computing the winner with function r . **d** Action-values Q are updated to track the mean value of all evaluations $r(\cdot)$ and $v_\theta(\cdot)$ in the subtree below that action.

AlphaGo. MCTS

В каждом ребре храним:

$$\{P(s, a), N_v(s, a), N_r(s, a), W_v(s, a), W_r(s, a), Q(s, a)\}$$

1. Selection (Выбор)

Начиная с корня дерева выбираем двигаемся по ветвям, выбирая путь $a_t = \operatorname{argmax}_a(Q(s_t, a) + u(s_t, a))$

$$u(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N_r(s, b)}}{1 + N_r(s, a)}$$

2. Evaluation

Добавляем лист L в очередь для вычисления $v_\theta(s_L)$, в случае если это значение еще не известно. Запускаем "rollout" с помощью p_π .

3. Expansion

Если $N_r(s, a) > n_{thr}$ добавляем новый узел с помощью p_τ .

4. Backpropagation

Обновляем информацию во всех узлах дерева.

$$Q(s, a) = (1 - \lambda) \frac{W_v(s, a)}{N_v(s, a)} + \lambda \frac{W_r(s, a)}{N_r(s, a)}$$

AlphaGo. Замечания

1. Можно использовать симметрии доски (D_4) для аугментации.
2. AlphaGo сдается, если $\max_a Q(s, a) < -0.8$ (вероятность выиграть < 10%).

AlphaGo Zero. Without human knowledge

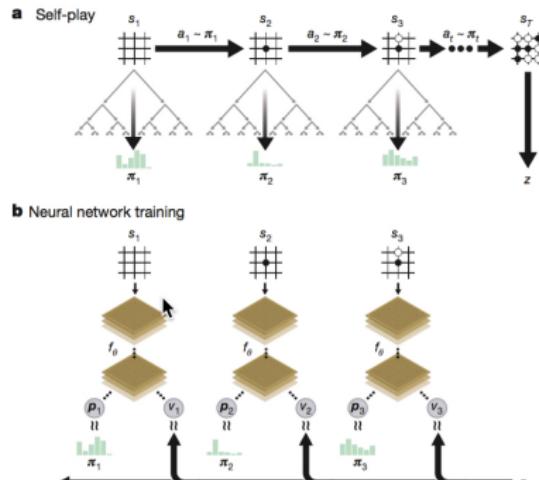


Figure 1 | Self-play reinforcement learning in AlphaGo Zero. **a**, The program plays a game s_1, \dots, s_T against itself. In each position s_t , an MCTS α_t is executed (see Fig. 2) using the latest neural network f_θ . Moves are selected according to the search probabilities computed by the MCTS, $a_t \sim \pi_t$. The terminal position s_T is scored according to the rules of the game to compute the game winner z . **b**, Neural network training in AlphaGo Zero. The neural network takes the raw board position s_t as its input, passes it through many convolutional layers with parameters θ , and outputs both a vector p_t , representing a probability distribution over moves, and a scalar value v_t , representing the probability of the current player winning in position s_t . The neural network parameters θ are updated to maximize the similarity of the policy vector p_t to the search probabilities π_t , and to minimize the error between the predicted winner v_t and the game winner z (see equation (1)). The new parameters are used in the next iteration of self-play as in **a**.

AlphaGo. Without human knowledge

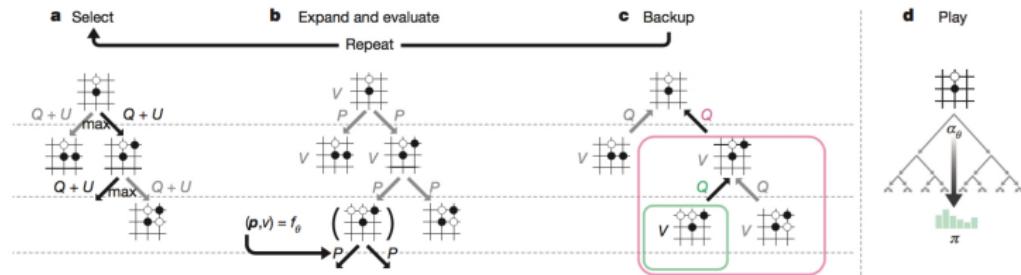


Figure 2 | MCTS in AlphaGo Zero. **a**, Each simulation traverses the tree by selecting the edge with maximum action value Q , plus an upper confidence bound U that depends on a stored prior probability P and visit count N for that edge (which is incremented once traversed). **b**, The leaf node is expanded and the associated position s is evaluated by the neural network $(P(s, \cdot), V(s)) = f_\theta(s)$; the vector of P values are stored in

the outgoing edges from s . **c**, Action value Q is updated to track the mean of all evaluations V in the subtree below that action. **d**, Once the search is complete, search probabilities π are returned, proportional to $N^{1/\tau}$, where N is the visit count of each move from the root state and τ is a parameter controlling temperature.



Вопросы