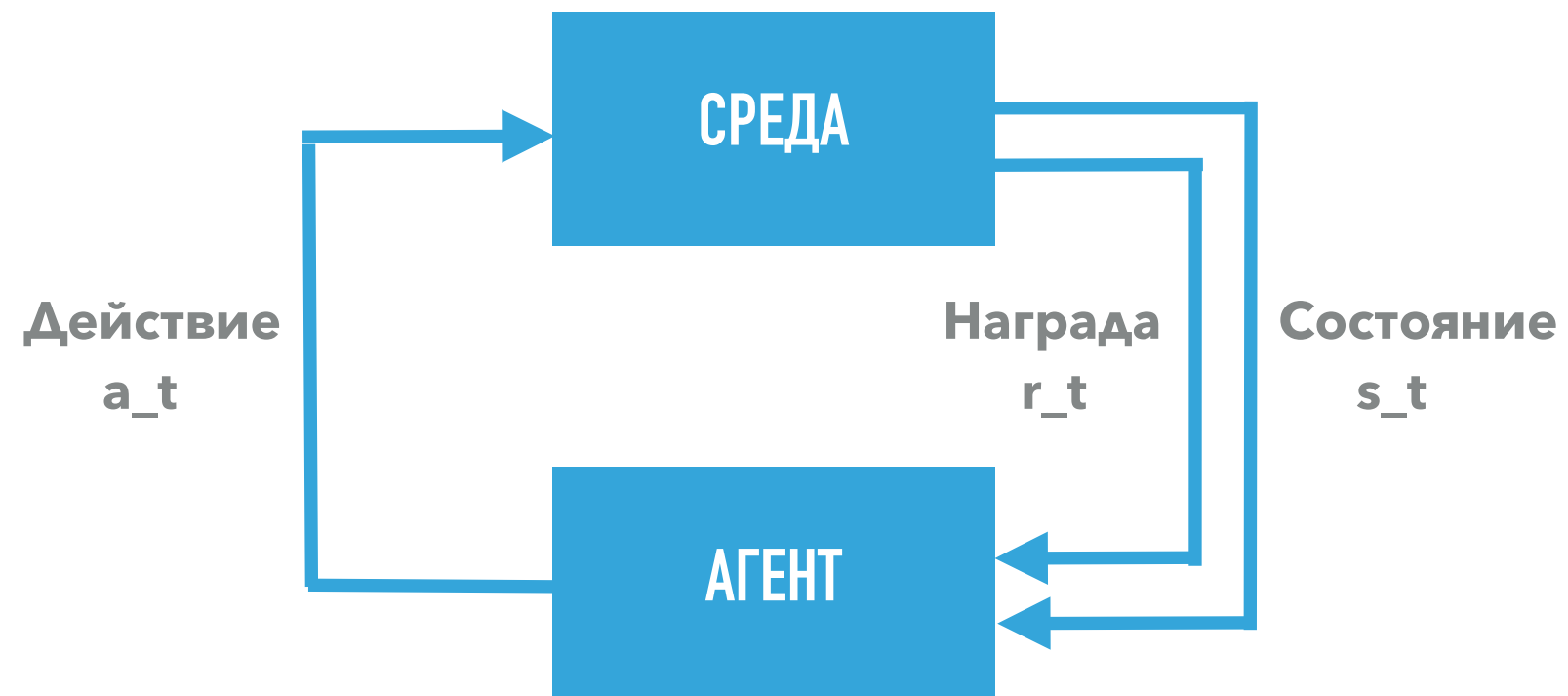


МАШИННОЕ ОБУЧЕНИЕ

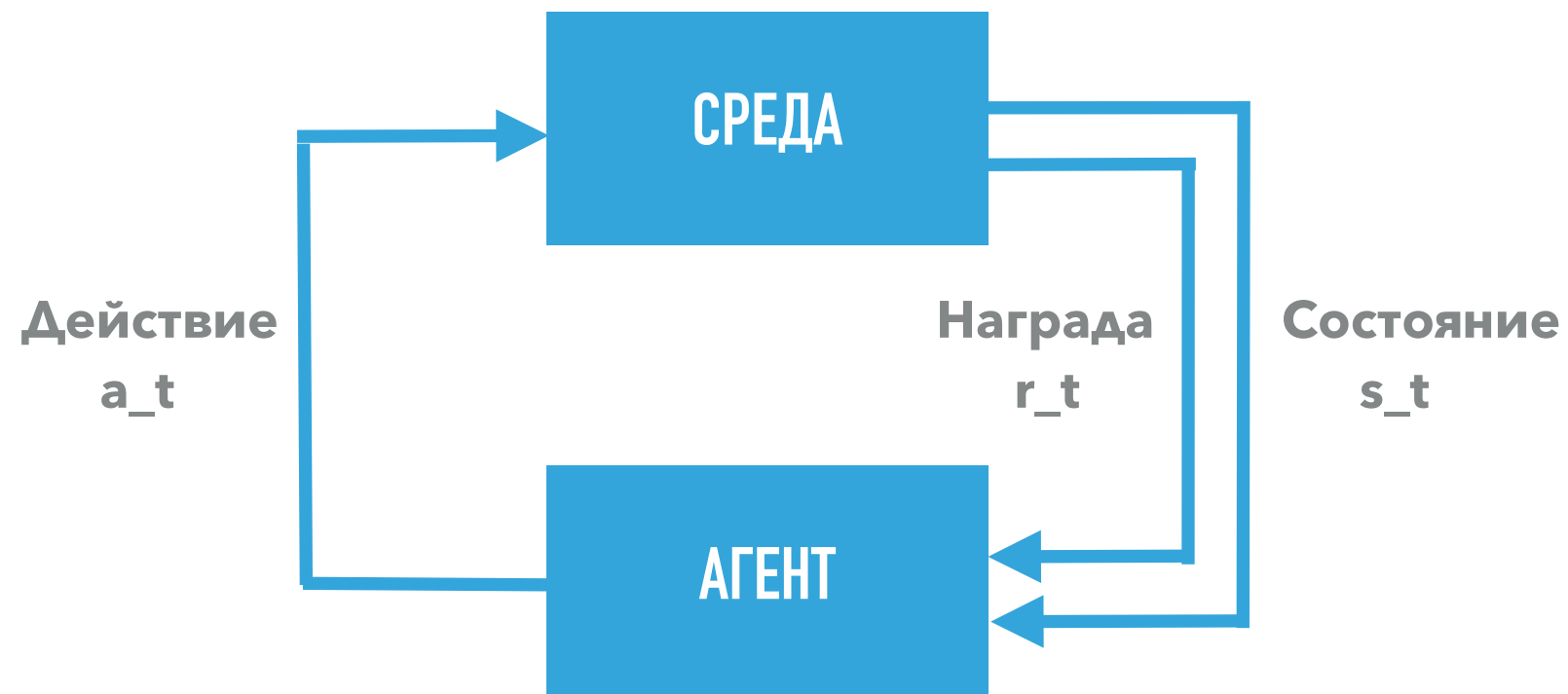
REINFORCEMENT LEARNING

- ▶ Пусть имеется некоторый **агент** (agent), способный выполнять **действия** (actions) из заданного набора
- ▶ Имеется **среда** (environment), с которой взаимодействует агент
- ▶ В ответ на действия агента среда определяет его в некоторое **состояние** (state) и выдает ему **вознаграждение** (reward)
- ▶ Целью агента является выработка **стратегии** (policy), приводящей к максимальному вознаграждению

ПОСТАНОВКА ЗАДАЧИ RL



ПОСТАНОВКА ЗАДАЧИ RL



На что похоже?

ПОСТАНОВКА ЗАДАЧИ RL

Похоже на задачу (адаптивного) оптимального управления:



Больше о подобных связях:

M. Annaswamy, Alexander L. Fradkov. A Historical Perspective of Adaptive Control and Learning

<https://arxiv.org/abs/2108.11336>

К КАКОМУ КЛАССУ ЗАДАЧ ОТНОСИТСЯ RL?

- ▶ Обучение с учителем?

К КАКОМУ КЛАССУ ЗАДАЧ ОТНОСИТСЯ RL?

- ▶ Обучение с учителем?

Нет. Мы сами добываем информацию, исходя из наших действий

К КАКОМУ КЛАССУ ЗАДАЧ ОТНОСИТСЯ RL?

- ▶ Обучение с учителем?

Нет. Мы сами добываем информацию, исходя из наших действий

- ▶ Обучение без учителя?

К КАКОМУ КЛАССУ ЗАДАЧ ОТНОСИТСЯ RL?

- ▶ Обучение с учителем?

Нет. Мы сами добываем информацию, исходя из наших действий

- ▶ Обучение без учителя?

Нет. Мы не ставим себе цель найти структуру данных

К КАКОМУ КЛАССУ ЗАДАЧ ОТНОСИТСЯ RL?

- ▶ Обучение с учителем?

Нет. Мы сами добываем информацию, исходя из наших действий

- ▶ Обучение без учителя?

Нет. Мы не ставим себе цель найти структуру данных

- ▶ Это отдельный класс задач со своей спецификой

ДЛИТЕЛЬНОСТЬ ЗАДАЧ

Два типа задач RL



```
graph TD; A[Два типа задач RL] --> B[Эпизодические]; A --> C[Непрерывные]
```

Эпизодические

Непрерывные

ДЛИТЕЛЬНОСТЬ ЗАДАЧ

Два типа задач RL



```
graph TD; A[Два типа задач RL] --> B[Эпизодические]; A --> C[Непрерывные]
```

Эпизодические

- есть терминальное состояние
- вознаграждения рассматриваются в рамках таких эпизодов
- нас интересует совокупное вознаграждение за эпизод:

$$R_1 + \dots + R_n$$

Непрерывные

ДЛИТЕЛЬНОСТЬ ЗАДАЧ

Два типа задач RL



```
graph TD; A[Два типа задач RL] --> B[Эпизодические]; A --> C[Непрерывные]
```

Эпизодические

- есть терминальное состояние
- вознаграждения рассматриваются в рамках таких эпизодов
- нас интересует совокупное вознаграждение за эпизод:

$$R_1 + \dots + R_n$$

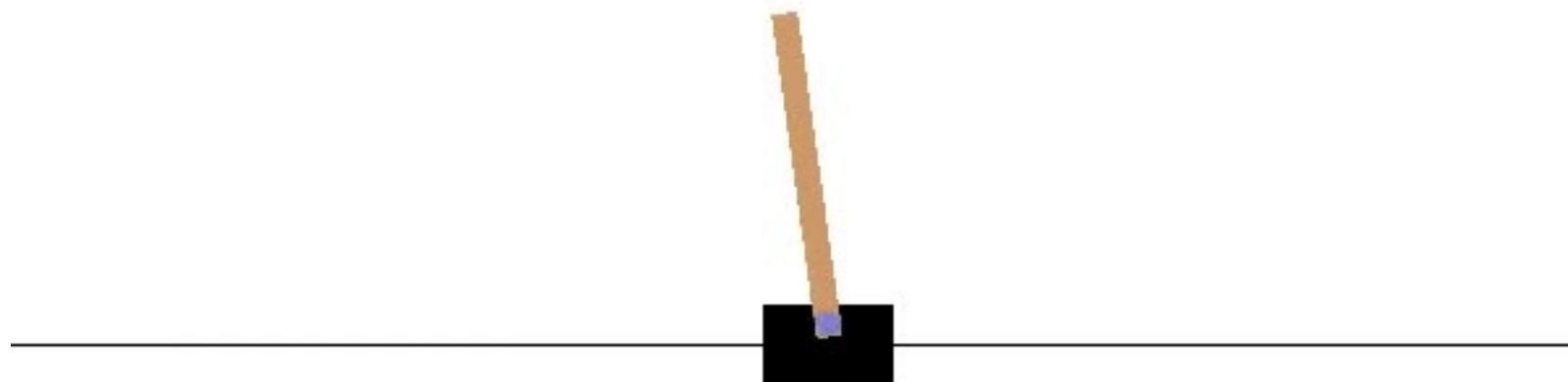
Непрерывные

- нет начала и конца
- зачастую рассматривают дисконтированное вознаграждение:

$$R_1 + \gamma R_2 + \dots + \gamma^{n-1} R_n + \dots$$

ДЛИТЕЛЬНОСТЬ ЗАДАЧ

Пример: маятник на тележке



ДЛИТЕЛЬНОСТЬ ЗАДАЧ

Пример: маятник на тележке

Как **непрерывная** задача: считать что награда в момент падения маятника падает на единицу, в остальные моменты не меняется

Как **эпизодическая**: награда растёт по мере удержания маятника, в момент падения эпизод заканчивается

ЖИЗНЕННЫЙ ПРИМЕР



- ▶ Перед нами набор n одноруких бандитов
- ▶ Хотим максимизировать выигрыш
- ▶ Как выбрать за каким автоматом играть в данный момент времени?

НЕМНОГО ФОРМАЛИЗУЕМ

- ▶ Пусть задано число n - количество бандитов
- ▶ Каждому бандиту соответствует стационарное распределение выигрыша. Пусть распределение i -го бандита - $N(\mu_i, 1)$
- ▶ Состояние всегда одно и тоже
- ▶ Действие: выбор i -го бандита
- ▶ Цель: максимизировать ожидаемый полный выигрыш

НЕМНОГО ФОРМАЛИЗУЕМ

- ▶ Хотим оценить ожидаемую награду при выборе действия **a** в момент времени **t**:

$$Q_t(a) \approx \mathbb{E}(R_t | A_t = a)$$

НЕМНОГО ФОРМАЛИЗУЕМ

- ▶ Хотим оценить ожидаемую награду при выборе действия **a** в момент времени **t**:

$$Q_t(a) \approx \mathbb{E}(R_t | A_t = a)$$

- ▶ Имея эту оценку можем действовать по **жадному** алгоритму:

$$A_t = \operatorname{argmax}_a (Q_t(a))$$

НЕМНОГО ФОРМАЛИЗУЕМ

- ▶ Пусть к моменту времени t выбрали $N_t(a)$ раз действие a , награды за эти действия были $R_1, R_2, \dots, R_{N_t(a)}$:

$$Q_t(a) = \frac{R_1 + R_2 + \dots + R_{N_t(a)}}{N_t(a)}$$

- ▶ При стремлении $N_t(a)$ к бесконечности наша оценка будет стремиться к реальному ожидаемому выигрышу

НЕМНОГО ПРЕОБРАЗУЕМ НАШУ ОЦЕНКУ

- Обозначим за Q_k оценку после **k**-й награды. Тогда:

$$Q_{k+1} = \frac{1}{k} \sum_{i=1}^k R_i = \frac{1}{k} \left(R_k + \sum_{i=1}^{k-1} R_i \right) =$$

$$\frac{1}{k} (R_k + (k-1)Q_k) = Q_k + \frac{1}{k} (R_k - Q_k)$$

НЕМНОГО ПРЕОБРАЗУЕМ НАШУ ОЦЕНКУ

- Обозначим за Q_k оценку после **k**-й награды. Тогда:

$$Q_{k+1} = \frac{1}{k} \sum_{i=1}^k R_i = \frac{1}{k} \left(R_k + \sum_{i=1}^{k-1} R_i \right) =$$

$$\frac{1}{k} (R_k + (k-1)Q_k) = Q_k + \frac{1}{k} (R_k - Q_k)$$

- Это часто возникающая ситуация:

$$\begin{aligned} \text{(новая оценка)} &= \text{(старая оценка)} + \\ &\quad \text{(step size)} * (\text{награда} - \text{старая оценка}) \end{aligned}$$

EXPLORE-EXPLOIT TRADEOFF

- ▶ Чтобы получать оценки необходимо совершать разные действия и смотреть на результат (exploration)
- ▶ Если перейти на жадность (exploit), то застрянем на одних и тех же действиях
- ▶ Сталкиваемся с характерной проблемой RL: действовать ли исходя из полученных знаний или исследовать еще

ВЫХОД: EPSILON-GREEDY АЛГОРИТМ

- ▶ Зададим параметр `epsilon` и с вероятностью `epsilon` на каждом шаге будем выполнять `exploration`
- ▶ Вместо константной вероятности можно использовать какую-нибудь затухающую функцию

см. [reinforcement-learning.ipynb](#)

ОПИСАНИЕ АЛГОРИТМА

- ▶ Пусть $Q^*(s, a)$ - ожидаемое (дискаунтированное) значение вознаграждения при совершении действия a в состоянии s
- ▶ Оценка Q^* происходит с использованием метода временных разниц (temporal differences) (см. далее)
- ▶ Агент использует таблицу оценок $Q(s, a)$ для множества всех состояний S и множества всех действий A

УРАВНЕНИЕ БЕЛЛМАНА

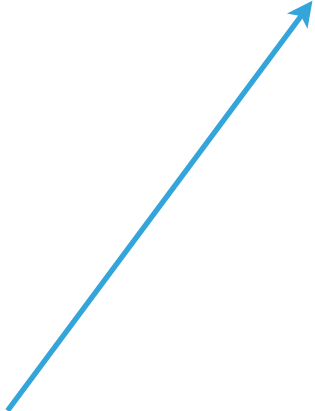
- ▶ Оптимальное значение $Q^*(s,a)$ должно удовлетворять следующему уравнению:

$$Q^*(s, a) = \mathbb{E}(R_{t+1} + \gamma \max_{a'} Q^*(s', a'))$$


УРАВНЕНИЕ БЕЛЛМАНА

- ▶ Оптимальное значение $Q^*(s,a)$ должно удовлетворять следующему уравнению:

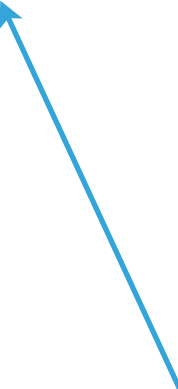
$$Q^*(s, a) = \mathbb{E}(R_{t+1} + \gamma \max_{a'} Q^*(s', a'))$$



Ожидаемое
значение награды,
совершая действие a
в состоянии s



Вознаграждение за
действие a в состоянии s



Ожидаемое значение награды
за будущие действия из нового
состояния s'

МЕТОД ВРЕМЕННЫХ РАЗНИЦ

- ▶ Тогда оценки Q будем обновлять по следующему правилу:

$$Q'(s, a) = Q(s, a) + \alpha[(R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a))]$$

где **alpha** - параметр learning rate

МЕТОД ВРЕМЕННЫХ РАЗНИЦ

- ▶ Тогда оценки Q будем обновлять по следующему правилу:

$$Q'(s, a) = Q(s, a) + \alpha [(R(s, a) + \gamma \max_{a'} Q(s', a')) - Q(s, a)]$$



Новое значение Q исходя из уравнения Беллмана и полученного на данном шаге вознаграждения

МЕТОД ВРЕМЕННЫХ РАЗНИЦ

- ▶ Тогда оценки Q будем обновлять по следующему правилу:

$$Q'(s, a) = Q(s, a) + \alpha [(R(s, a) + \gamma \max_{a'} Q(s', a')) - Q(s, a)]$$



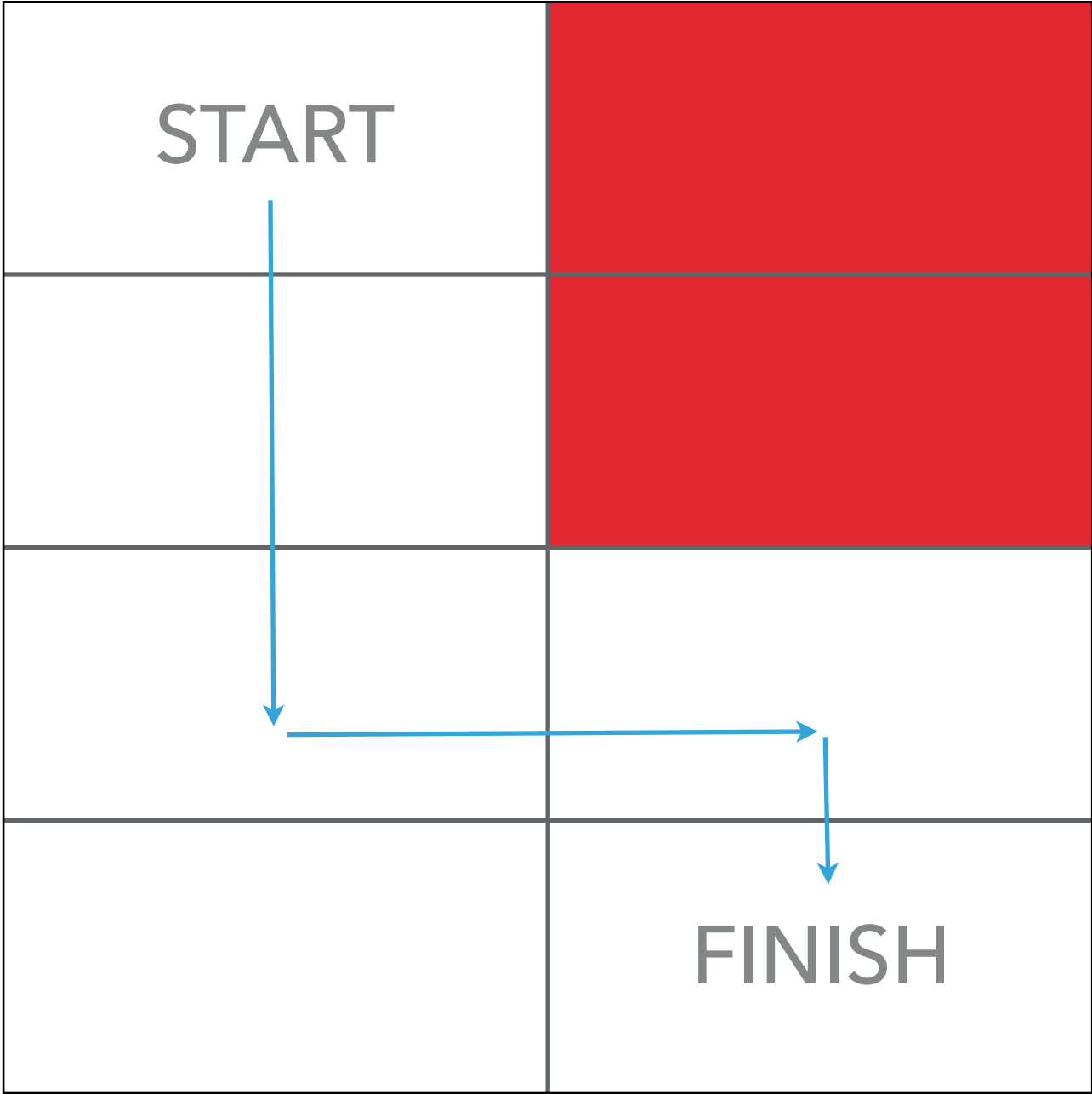
Новое значение Q исходя из уравнения Беллмана и полученного на данном шаге вознаграждения

- ▶ Т.е. с шагом **alpha** подгоняем Q под ожидаемое значение

ЗАДАЧА: ПОПАСТЬ ИЗ START В FINISH

START	
	FINISH

ЗАДАЧА: ПОПАСТЬ ИЗ START В FINISH



ОПИШЕМ СИСТЕМУ

START	
	FINISH

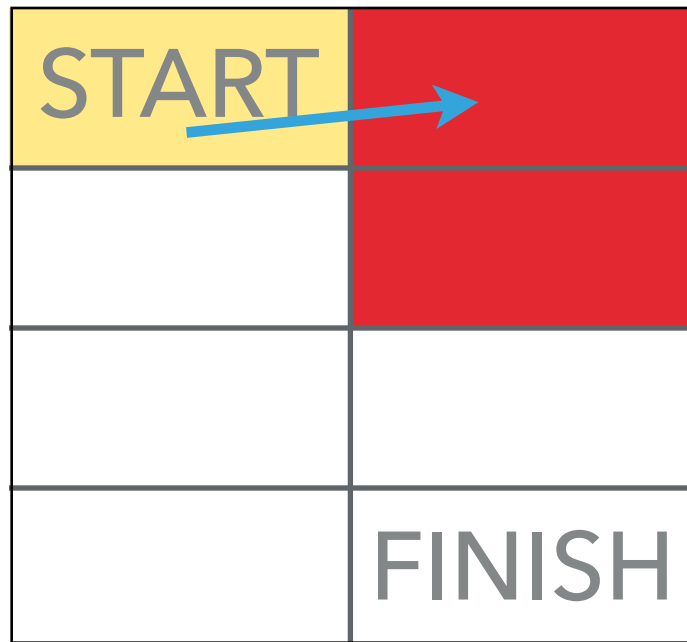
- ▶ Состояния:
[0, 0], [0, 1], [1, 0], [1, 1],
[2, 0], [2, 1], [3, 0], [3, 1]
- ▶ Действия:
вправо (R), влево (L),
вниз (D), вверх (U)
(все на клеточку)
- ▶ Награда:
-1 за попадание на красное
+1 за попадание в FINISH
0 за остальные случаи

ЗАВЕДЕМ Q-ТАБЛИЦУ

START	
	FINISH

	R	L	U	D
[0,0]	0	0	0	0
[0,1]	0	0	0	0
[1,0]	0	0	0	0
[1,1]	0	0	0	0
[2,0]	0	0	0	0
[2,1]	0	0	0	0
[3,0]	0	0	0	0
[3,1]	0	0	0	0

ПОЙДЕМ НАПРАВО

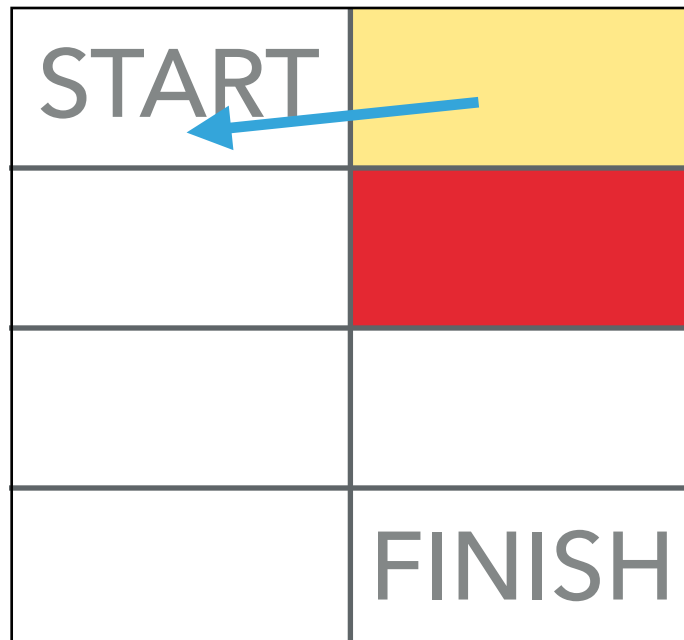


	R	L	U	D
[0,0]	0	0	0	0
[0,1]	0	0	0	0
[1,0]	0	0	0	0
[1,1]	0	0	0	0
[2,0]	0	0	0	0
[2,1]	0	0	0	0
[3,0]	0	0	0	0
[3,1]	0	0	0	0

$$Q'(s, a) = Q(s, a) + \alpha[(R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a))] \quad \alpha = 0.1, \gamma = 0.9$$

$$Q'([0, 0], R) = 0 + 0.1[-1 + 0.9 \max_{a'} Q([0, 1], a') - 0] = -0.1$$

ПОЙДЕМ ВЛЕВО

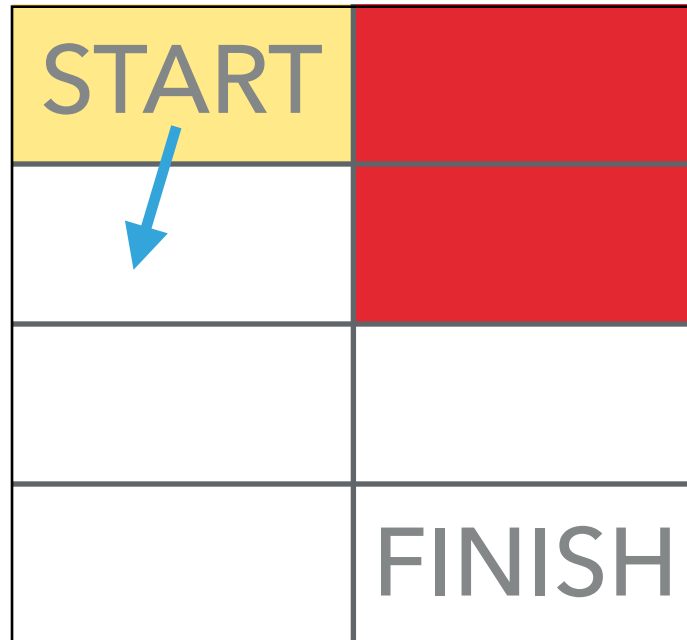


	R	L	U	D
[0,0]	-0.1	0	0	0
[0,1]	0	0	0	0
[1,0]	0	0	0	0
[1,1]	0	0	0	0
[2,0]	0	0	0	0
[2,1]	0	0	0	0
[3,0]	0	0	0	0
[3,1]	0	0	0	0

$$Q'(s, a) = Q(s, a) + \alpha[(R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a))] \quad \alpha = 0.1, \gamma = 0.9$$

$$Q'([0, 1], L) = 0 + 0.1[-1 + 0.9 \max_{a'} Q([0, 0], a') - 0] = -0.1$$

ПОЙДЕМ ВНИЗ



	R	L	U	D
[0,0]	-0.1	0	0	0
[0,1]	0	-0.1	0	0
[1,0]	0	0	0	0
[1,1]	0	0	0	0
[2,0]	0	0	0	0
[2,1]	0	0	0	0
[3,0]	0	0	0	0
[3,1]	0	0	0	0

$$Q'(s, a) = Q(s, a) + \alpha[(R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a))] \quad \alpha = 0.1, \gamma = 0.9$$

$$Q'([0, 0], D) = 0 + 0.1[-1 + 0.9 \max_{a'} Q([1, 0], a') - 0] = -0.1$$

..... ЧЕРЕЗ МНОГО ШАГОВ

START	
	FINISH

	R	L	U	D
[0,0]	-100	0	0	70
[0,1]	0	30	-50	20
[1,0]	-80	0	10	60
[1,1]	0	20	-40	10
[2,0]	50	0	15	50
[2,1]	0	10	5	40
[3,0]	40	0	10	0
[3,1]	0	0	0	0

$Q'(s, a) = Q(s, a) + \alpha[(R(s, a) + \gamma \max Q(s', a') - Q(s, a)] \quad \alpha = 0.1, \gamma = 0.9$

А ЧТО ДЕЛАТЬ ЕСЛИ СОСТОЯНИЙ МНОГО?

- ▶ Кажется естественным приблизить Q-таблицу какой-нибудь функцией

А ЧТО ДЕЛАТЬ ЕСЛИ СОСТОЯНИЙ МНОГО?

- ▶ Кажется естественным приблизить Q-таблицу какой-нибудь функцией
- ▶ Например, нейронной сетью
- ▶ Чтобы приблизить функцию $Q(s, a)$ пусть сеть принимает на вход состояние s в векторном виде (например, ONE), а на выходе дает вектор размерности действий, где каждый выход будет соответствовать значению Q для этого действия

ЗАМЕНА Q-ТАБЛИЦЫ НА DEEP Q-NETWORK:

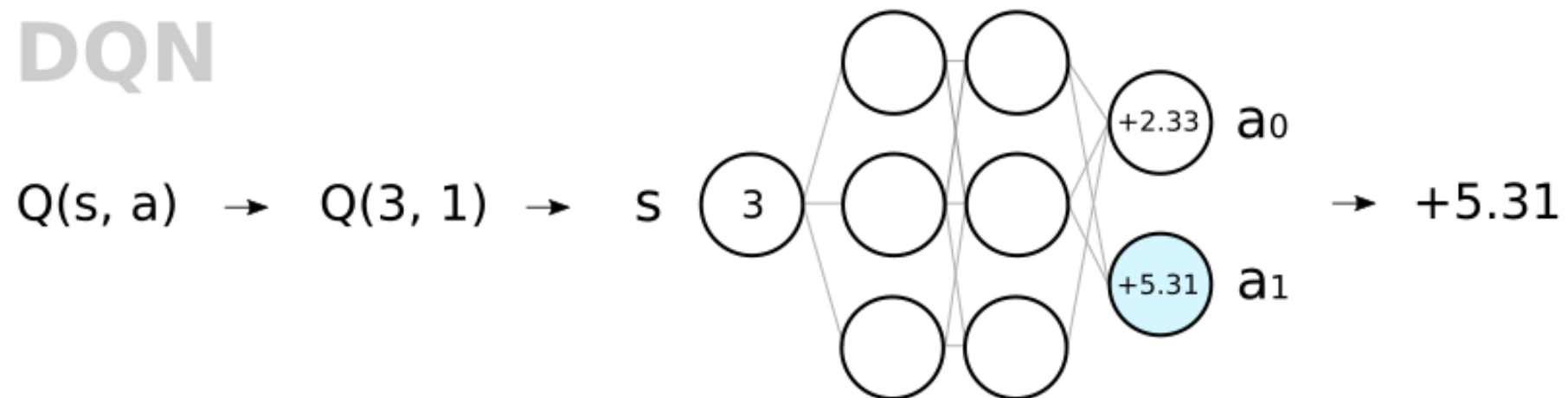
Q-Table

$Q(s, a) \rightarrow Q(3, 1) \rightarrow$

	S0	S1	S2	S3	S4
a0	+4.21	+3.24	+1.84	+2.33	+3.73
a1	+2.53	+7.44	+3.34	+5.31	+6.22

$\rightarrow +5.31$

DQN



КАК БУДЕТ ОБУЧАТЬСЯ СЕТЬ?

- ▶ Правило "обучения" Q-таблицы было таким:

$$Q'(s, a) = Q(s, a) + \alpha [R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

КАК БУДЕТ ОБУЧАТЬСЯ СЕТЬ?

- ▶ Правило "обучения" Q-таблицы было таким:

$$Q'(s, a) = Q(s, a) + \alpha [R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

- ▶ Будем обучать сеть так, чтобы ее выход соответствовал левой части этого уравнения

КАК БУДЕТ ОБУЧАТЬСЯ СЕТЬ?

- ▶ Правило "обучения" Q-таблицы было таким:

$$Q'(s, a) = Q(s, a) + \alpha [R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

- ▶ Будем обучать сеть так, чтобы ее выход соответствовал левой части этого уравнения
- ▶ Можно считать что **alpha=1**, поскольку внутри backpropagation уже есть learning rate:

$$Q'(s, a) = R(s, a) + \gamma \max_{a'} Q(s', a')$$

ИТОГО, АЛГОРИТМ ТАКОЙ:

- ▶ Выбираем архитектуру сети с входами и выходами исходя из постановки задачи
- ▶ Выполняем некоторое количество exploration шагов, где целевое значение берем исходя из уравнения Беллмана, а для вычисления $\max(\mathbf{Q}(s', a'))$ совершаем инференс сети с входом s'
- ▶ Начинаем чередовать exploration с exploitation
- ▶ Используем обученную сеть для решения задачи

ЗАМЕЧАНИЯ

- ▶ При программной реализации нужно аккуратно чередовать инференс и обучения
- ▶ Обучать сеть по одному семплу (одному действию) может быть не очень эффективно. Лучше делать это батчами (см. replay memory далее)
- ▶ Переход к нейронным сетям дает нам возможность этой же сетью извлекать полезные фичи из состояния. Например, подавать на вход изображения

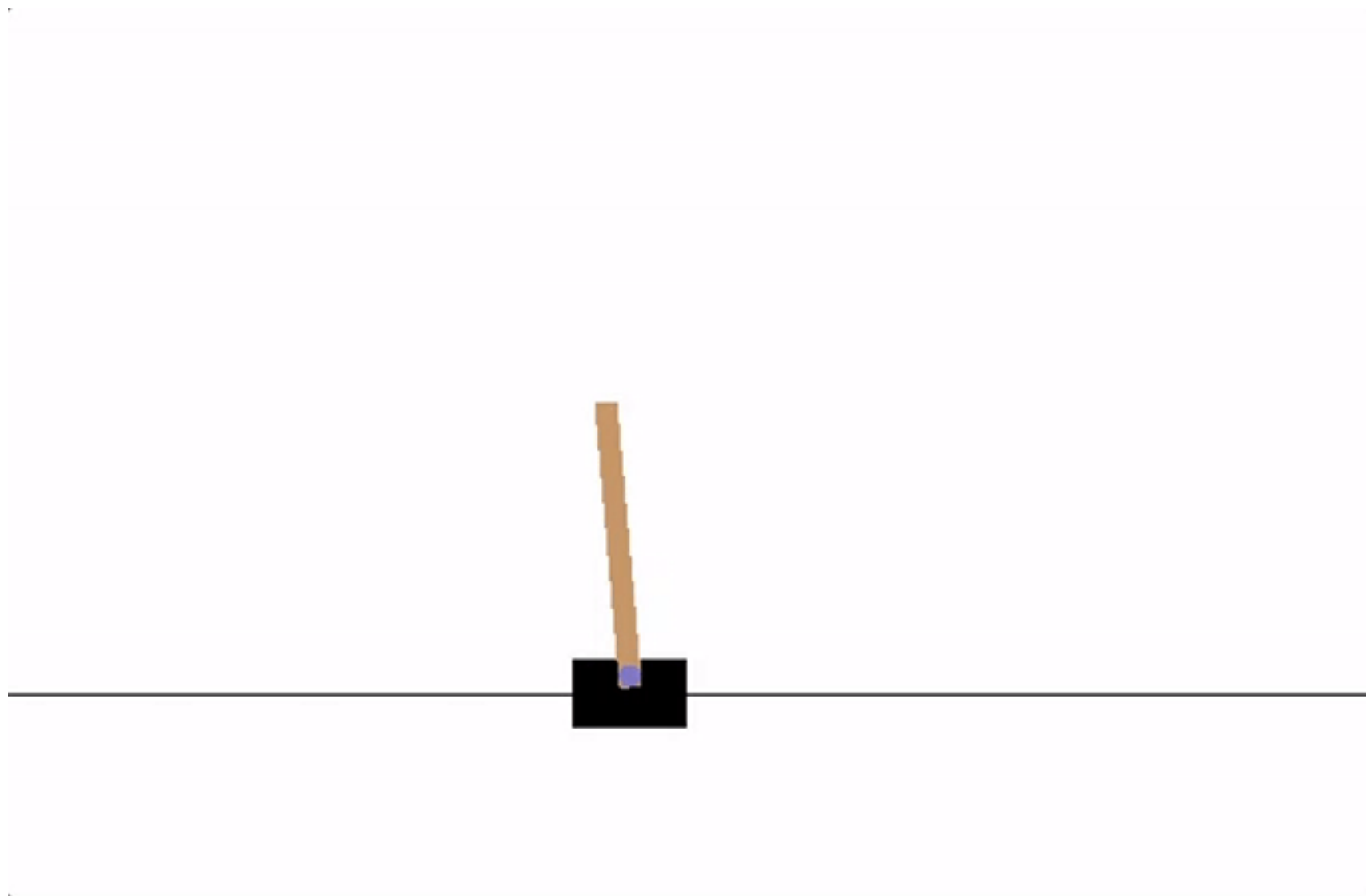
REPLAY MEMORY

- ▶ Идея подхода состоит в следующем: хранить в памяти буффер заданного размера
- ▶ После выполнения каждого из действий сохранять в буфере четверки
(состояние, действие, следующее состояние, награда)
- ▶ При этом параметры сети после шагов не меняются
- ▶ После каждых k шагов выполнять обучение сети, случайным батчем из буфера

REPLAY MEMORY

- ▶ На практике имеет смысл хранить две сети: одну сеть использовать для обучения, а вторую для вычисления Q
- ▶ Тогда алгоритм будет следующий:
 1. Используя вторую сеть выполнили некоторое количество exploration/exploitation шагов
 2. Обновили веса обучаемой сети батчем из буфера
 3. Скопировали обновленные веса на сеть для вычисления
 4. Перешли к шагу 1.

ОБРАТНЫЙ МАЯТНИК НА ТЕЛЕЖКЕ



ОБРАТНЫЙ МАЯТНИК НА ТЕЛЕЖКЕ

- ▶ В классической постановке нам даны положения, скорости и прочее от тележки и маятника
- ▶ И можно решить задачу, подавая это на вход сети

ОБРАТНЫЙ МАЯТНИК НА ТЕЛЕЖКЕ

- ▶ В классической постановке нам даны положения, скорости и прочее от тележки и маятника
- ▶ И можно решить задачу, подавая это на вход сети
- ▶ Но можно подавать на вход сразу изображения
- ▶ Чтобы учитывать скорость будут подаваться не непосредственно изображения, а разницы между соседними кадрами

REPLAY MEMORY

```
Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))

class ReplayMemory(object):

    def __init__(self, capacity):
        self.memory = deque([],maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

АРХИТЕКТУРА СЕТИ

```
class DQN(nn.Module):

    def __init__(self, h, w, outputs):
        super(DQN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=2)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)
        self.bn3 = nn.BatchNorm2d(32)

        # Number of Linear input connections depends on output of conv2d layers
        # and therefore the input image size, so compute it.
        def conv2d_size_out(size, kernel_size = 5, stride = 2):
            return (size - (kernel_size - 1) - 1) // stride + 1
        convw = conv2d_size_out(conv2d_size_out(conv2d_size_out(w)))
        convh = conv2d_size_out(conv2d_size_out(conv2d_size_out(h)))
        linear_input_size = convw * convh * 32
        self.head = nn.Linear(linear_input_size, outputs)

        # Called with either one element to determine next action, or a batch
        # during optimization. Returns tensor([[left@exp, right@exp]...]).
    def forward(self, x):
        x = x.to(device)
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        return self.head(x.view(x.size(0), -1))
```

ИНИЦИАЛИЗАЦИЯ

```
BATCH_SIZE = 128
GAMMA = 0.999
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 200
TARGET_UPDATE = 10
```

```
policy_net = DQN(screen_height, screen_width, n_actions).to(device)
target_net = DQN(screen_height, screen_width, n_actions).to(device)
target_net.load_state_dict(policy_net.state_dict())
target_net.eval()

optimizer = optim.RMSprop(policy_net.parameters())
memory = ReplayMemory(10000)

steps_done = 0
```

ВЫБОР ДЕЙСТВИЯ АГЕНТА

```
def select_action(state):
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * \
        math.exp(-1. * steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            # t.max(1) will return largest column value of each row.
            # second column on max result is index of where max element was
            # found, so we pick action with the larger expected reward.
            return policy_net(state).max(1)[1].view(1, 1)
    else:
        return torch.tensor([[random.randrange(n_actions)]]), device=device, dtype=torch.long)
```

ОБНОВЛЕНИЕ МОДЕЛИ [1]

```
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see https://stackoverflow.com/a/19343/3343043 for
    # detailed explanation). This converts batch-array of Transitions
    # to Transition of batch-arrays.
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch elements
    # (a final state would've been the one after which simulation ended)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                             batch.next_state)), device=device, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state
                                       if s is not None])

    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
    # columns of actions taken. These are the actions which would've been taken
    # for each batch state according to policy_net
    state_action_values = policy_net(state_batch).gather(1, action_batch)
```


ОБНОВЛЕНИЕ МОДЕЛИ [2]

```
# Compute V(s_{t+1}) for all next states.
# Expected values of actions for non_final_next_states are computed based
# on the "older" target_net; selecting their best reward with max(1)[0].
# This is merged based on the mask, such that we'll have either the expected
# state value or 0 in case the state was final.
next_state_values = torch.zeros(BATCH_SIZE, device=device)
next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0].detach()
# Compute the expected Q values
expected_state_action_values = (next_state_values * GAMMA) + reward_batch

# Compute Huber loss
criterion = nn.SmoothL1Loss()
loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

# Optimize the model
optimizer.zero_grad()
loss.backward()
for param in policy_net.parameters():
    param.grad.data.clamp_(-1, 1)
optimizer.step()
```

ОСНОВНОЙ ЦИКЛ [1]

```
num_episodes = 50
for i_episode in range(num_episodes):
    # Initialize the environment and state
    env.reset()
    last_screen = get_screen()
    current_screen = get_screen()
    state = current_screen - last_screen
    for t in count():
        # Select and perform an action
        action = select_action(state)
        _, reward, done, _ = env.step(action.item())
        reward = torch.tensor([reward], device=device)

        # Observe new state
        last_screen = current_screen
        current_screen = get_screen()
        if not done:
            next_state = current_screen - last_screen
        else:
            next_state = None
```

ОСНОВНОЙ ЦИКЛ [2]

```
# Store the transition in memory
memory.push(state, action, next_state, reward)

# Move to the next state
state = next_state

# Perform one step of the optimization (on the policy network)
optimize_model()
if done:
    episode_durations.append(t + 1)
    plot_durations()
    break

# Update the target network, copying all weights and biases in DQN
if i_episode % TARGET_UPDATE == 0:
    target_net.load_state_dict(policy_net.state_dict())
```

A 4TO C TRANSFER LEARNING?

А ЧТО С TRANSFER LEARNING?

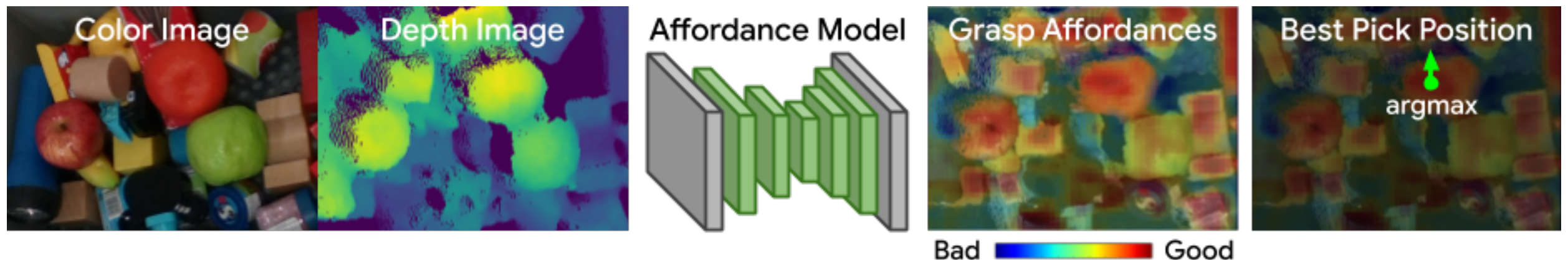
- ▶ У нас используется нейронная сеть
- ▶ Во многих приложениях ей на вход будут идти изображения
- ▶ Поможет ли отдельное обучение этой сети на похожих данных?

AFFORDANCE-BASED MANIPULATION

- ▶ Хотим обучить модель, определяющую возможность (affordance) применения той или иной манипуляции (схватить, толкнуть, бросить, присосать и т.д.)

AFFORDANCE-BASED MANIPULATION

- ▶ Хотим обучить модель, определяющую возможность (affordance) применения той или иной манипуляции (схватить, толкнуть, бросить, присосать и т.д.)
- ▶ Можно использовать модели типа сегментации:



AFFORDANCE-BASED MANIPULATION

- ▶ Предобучив сеть на обычные задачи классификации/сегментации удалось ускорить обучение RL модели:

Pre-trained Model	Random	ImageNet	COCO-backbone	COCO
Success (first 50 trials)	9	11	15	23

Number of successful grasps out of the first 50 attempts using: a random initialization of weights, backbone and head pre-trained on ImageNet, COCO pre-trained backbone only, and backbone and head trained on COCO.

AFFORDANCE-BASED MANIPULATION

- ▶ Предобучив сеть на обычные задачи классификации/сегментации удалось ускорить обучение RL модели:

Pre-trained Model	Random	ImageNet	COCO-backbone	COCO
Success (first 50 trials)	9	11	15	23

Number of successful grasps out of the first 50 attempts using: a random initialization of weights, backbone and head pre-trained on ImageNet, COCO pre-trained backbone only, and backbone and head trained on COCO.

- ▶ Аналогично может сработать и в других задачах

ДРУГИЕ ВАРИАНТЫ

- ▶ Перенос с симулятора на реального робота
- ▶ Перенос с разных типов датчиков
- ▶ Перенос с лаборатории на реальную среду

(ВОЗМОЖНЫЕ) ПРИЛОЖЕНИЯ

- ▶ Беспилотные автомобили
- ▶ Автоматизация производственных роботов
- ▶ Трейдинг роботы
- ▶ Машинные переводы, чатботы
- ▶ Назначение лечения
- ▶ Управление роботом-манипулятором

(ВРОДЕ КАК) РЕАЛЬНЫЕ ПРИЛОЖЕНИЯ

- ▶ Оптимизация стриминга видео в Facebook
<https://engineering.fb.com/2018/11/01/ml-applications/horizon/>
- ▶ Рекомендации в Netflix
<https://research.netflix.com/research-area/machine-learning>
- ▶ Рекомендации в Spotify
<https://dl.acm.org/doi/10.1145/3240323.3240354>
- ▶ Оптимизация охлаждений дата центра в Google
<https://deepmind.com/blog/article/deepmind-ai-reduces-google-data-centre-cooling-bill-40>
- ▶ В роботике много стартапов, но не знаю об успехах...
<https://www.ai-startups.org/top/robotics/>

ОЧЕНЬ СЛОЖНО ПРИМЕНЯТЬ RL :(

- ▶ Требуется очень много семплов чтобы обучить
- ▶ Сложности определения функции вознаграждений
- ▶ Сложно добиться обобщений (проблема переобучения)
- ▶ Все очень плохо с воспроизводимостью экспериментов

<https://www.alexirpan.com/2018/02/14/rl-hard.html>

А ЕСТЬ ЕЩЕ СЛОЖНЕЕ ЗАДАЧИ

- ▶ Зачастую подобрать функцию вознаграждения в принципе нереально
- ▶ В этом случае можно сформулировать обратную задачу обучения с подкреплением (inverse reinforcement learning):
 - ▶ По оптимальному поведению некоторого агента определить какая должна быть функция вознаграждения
 - ▶ Или поставить задачу imitation learning, где также по поведению агента вырабатывается оптимальная политика