# Глубокое Q-Learning (Q-обучение)

Read the introduction to this notebook [here](https://medium.com/@fabiograetz/tutorial-double-deep-q-learning-with-dueling-network-architectures-4c1b3fb7f756)...

## Contents

1. Reinforcement Learning
2. Q-Learning
3. Dueling Networks
4. Exploration-exploitation trade-off
5. Replay memory
6. Target network and parameter update
7. Double Q-Learning
8. Putting everything together: DQN
9. Train it yourself
10. Use a trained network

## 1. Усиленное обучение (Reinforcement Learning )

Например, при контролируемом обучении нейронная сеть изучает функцию, которая отображает входные данные в соответствующий выход / цель на основе большого количества помеченных обучающих данных, состоящих из примеров пар ввода-вывода: Проще говоря, если вы обучаете нейронную сеть сеть, чтобы классифицировать, например, кошек и собак, вы неоднократно отображаете сетевые изображения кошек или собак, сравниваете прогноз сети с меткой и слегка адаптируете параметры сети, пока нейронная сеть не сможет классифицировать, какое животное показано на рисунке ,

Теперь, допустим, вы позволите ребенку поиграть в компьютерную игру, в которую он никогда раньше не играл. В случае прорыва игрок видит экран пикселей в качестве входных данных и должен решить, двигаться ли он влево или вправо. Вы, конечно, могли бы много раз показывать ребенку, в каких ситуациях ему нужно нажимать влево, а в каких ситуациях - вправо, чтобы выиграть игру (контролируемое обучение), но ребенок наверняка быстро скучает и попытается оттолкнуть вас в сторону, желая попробовать сама игра. И ребенок научился бы играть в игру быстро, без объяснения, как это сделать, просто оценивая, какие действия приводят к увеличению оценки. В обучении с подкреплением мы пытаемся заставить компьютер учиться таким же образом, позволяя ему исследовать окружающую среду и иногда давая ему вознаграждение, когда счет увеличивается.

Однако по сравнению с контролируемым обучением это создает проблему. На стр. 1 из Mnih et al. 2013 авторы говорят:

Алгоритмы RL [...] должны быть в состоянии учиться на скалярном сигнале вознаграждения, который часто [...] разрежен и задержан. Задержка между действиями и получающимися в результате наградами, которая может длиться тысячи раз, кажется особенно пугающей по сравнению с прямой связью между входами и целями, обнаруживаемыми в контролируемом обучении.

Что авторы имеют в виду под «разреженными и [...] задержанными»?

Представьте, что вы идете через лабиринт, пытаясь найти сокровища. Вы получаете награду, когда найдете золото. Теперь представьте, что вы столкнулись с развилкой на пути. Какой путь вы выберете? В отличие от контролируемого обучения, на развилке агент не получает немедленной обратной связи / вознаграждения за правильный путь, но только позже, когда находит золото. Тем не менее, возможно, было бы крайне важно взять, например, левый путь на развилке. Это то, что авторы имеют в виду с задержкой. Проблема решается путем дисконтирования будущих вознаграждений с коэффициентом *𝛾*   
(от 0 до 1).

[Sutton and Barto 2018](https://drive.google.com/file/d/1xeUDVGWGUUv1-ccUMAZHJLej2C7aAFWY/view) написал на странице 55:

Дополнительная концепция, которая нам нужна, - это дисконтирование. Согласно этому подходу агент пытается выбрать действия таким образом, чтобы сумма дисконтированных вознаграждений, которые он получает в будущем, была максимальной. В частности, он выбирает *𝐴𝑡*, чтобы максимизировать ожидаемый дисконтированный доход:

Дисконтированный доход *𝐺𝑡*рассчитывается следующим образом

*𝐺𝑡*=*𝑅𝑡*+1+*𝛾𝑅𝑡*+2+*𝛾*2*𝑅𝑡*+3+...

=*𝑅𝑡*+1+(*𝑅𝑡*+2+*𝛾𝑅𝑡*+3+...)

=*𝑅𝑡*+1+*𝛾𝐺𝑡*+1

где *𝑅𝑡* - вознаграждение, полученное агентом за время *𝑡*

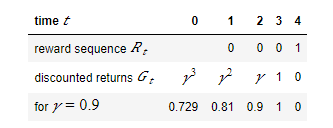
Авторы также объясняют:

вознаграждение, полученное за *𝑘* steps временных шагов в будущем, стоит всего  *𝛾𝑘*−1 раз, чем было бы, если бы оно было получено немедленно.

И

Когда *𝛾*  приближается к 1, цель возврата все сильнее учитывает будущие выгоды; агент становится более дальновидным.

Давайте посмотрим на очень простой пример, где есть только одна награда, не равная 0 (вычисление справа налево):



Незначительные детали: в этом примере нет вознаграждения при 𝑡 = 0, потому что первая награда *𝑅*1 является результатом действия *𝑎*0, предпринятого в состоянии  *𝑠*0, и награда всегда связана со следующим состоянием в Саттоне и Барто. В некоторых статьях награда *𝑅𝑖* связана с состоянием *𝑠𝑖* и действием *𝑎𝑖*, вызвавшим награду. В этом случае определение дисконтированной доходности изменится на *𝐺𝑡*=*𝑅𝑡*+*𝛾𝐺𝑡*+1, в таблице все 1 будут в одном столбце, и будет вознаграждение, равное 0 при 𝑡 = 0, но не при 𝑡 = 4

Проще говоря, путем дисконтирования доходов будущие вознаграждения увеличивают прошлые или текущие доходы, и чем ближе 𝛾 к 1, тем дальше агент может заглянуть в будущее.

А разреженный? В нашем примере с вымышленным лабиринтом награды тем меньше, чем меньше золота вы найдете. Для агента игру сложнее освоить, чем меньше награда. Понг - одна из игр, которую DQN может учить быстрее всего, потому что счет меняется довольно часто. Месть Монтесумы, с другой стороны, имеет очень редкие награды, и DQN (по крайней мере, без некоторых дополнительных уловок) вообще не в состоянии изучить игру.

## 2. Q-обечение (Q-Learning)

Так как же работает Q-Learning? Если агент (независимо от того, обучен или все еще не обучен) отображается состояние 𝑠 в игре, он должен решить, какое действие 𝑎 выполнить (например, переместиться влево или вправо). Как оно это делает? На странице 2 Mnih et al. 2013 определить так называемый 𝑄 - Функция:

Определим оптимальную функцию-значение *𝑄*∗(*𝑠*,*𝑎*), как максимальный ожидаемый доход, достижимый при следовании любой стратегии, после просмотра некоторой последовательности 𝑠 и затем принятия некоторых действий *𝑎*

Это означает, что с учетом состояния игры 𝑠 (на данный момент, пожалуйста, рассматривайте последовательности как состояния игры), *𝑄*∗(*𝑠*,*𝑎*)- лучший (дисконтированный) общий доход, который может получить агент, если он выполнит действие 𝑎 в текущем состоянии 𝑠. Так как же выбрать, какое действие выполнить, если мы уже знаем *𝑄*∗(*𝑠*,*𝑎*)? Одна очевидная стратегия состояла бы в том, чтобы всегда выбирать действие с максимальным значением 𝑄 ∗ (мы увидим позже, почему это немного проблематично). Но прежде всего нам нужно найти эту магическую функцию 𝑄 ∗:

Допустим, мы находимся в состоянии 𝑠, решить выполнить действие 𝑎 и перейти в следующее состояние 𝑠 ′. Если предположить, что в состоянии s 𝑄∗ - значения для всех возможных действий 𝑎′ уже известны, то action ∗ -значение в состоянии 𝑠 для действия 𝑎 (значение-действия в 𝑠 для действия 𝑎) будет наградой 𝑟 мы получили за выполнение действия 𝑎 плюс дисконтированный максимальный доход в будущем 𝑠 ′:

*𝑄*∗(*𝑠*,*𝑎*)=*𝑟*+*𝛾*max(*𝑄*∗(*𝑠*′,*𝑎*′))

Менее формально, но более интуитивно понятно: если бы мы были на определенной развилке в лабиринте (состояние state ) и хотим узнать, насколько хорошо было выбрать левый путь (выполнить действие 𝑎), мы добавляем найденное золото после перехода влево (вознаграждение, которое мы получили за выполнение действия 𝑎) и количество золота, которое мы ожидаем найти в будущем (максимальный дисконтированный доход в будущем). Если вам нужно выбрать между поиском золотой монеты сразу после разворота налево на развилке и ничем другим по дороге или поиском ничего сразу после движения вправо, кроме сокровищ по дороге, ну, правильный путь стоит больше / имеет более высокое действие -стоимость.

Это так называемое уравнение Беллмана. Глубокое Q-Learning использует нейронную сеть, чтобы найти приближение *𝑄*(*𝑠*,*𝑎*,*𝜃*)

*𝑄*∗(*𝑠*,*𝑎*) *𝜃* являются параметрами нейронной сети. Позже мы обсудим, как именно обновляются параметры сети. Теперь я объясню вам, как нейронная сеть отображает состояние 𝑠 в Q-значения для возможных действий 𝑎.

Ранее я упоминал, что я рассматриваю последовательность как состояние. Что я имел ввиду? Представьте, что у вас есть четкое изображение летящего футбольного мяча. Можете ли вы сказать, в каком направлении он движется? Нет, вы не можете, но вы могли бы, если бы на изображении было какое-то размытое изображение, или если вы сделали несколько снимков, сделанных быстро одно за другим, каждый раз показывая мяч в несколько ином положении. Та же проблема возникает в играх Atari. Из одного кадра игры Понг агент не может различить, в каком направлении движется мяч. DeepMind решил эту проблему, сложив несколько последовательных кадров и считая эту последовательность состоянием, которое передается в нейронную сеть. Из такой последовательности агент может определить направление и скорость движения, потому что мяч находится в разных позициях в каждом кадре.

Поскольку агент наблюдает только за изображениями текущего экрана [...], невозможно полностью понять текущую ситуацию только с текущего экрана 𝑥𝑡. Поэтому мы рассматриваем последовательности действий и наблюдений, 𝑠𝑡 = 𝑥1, 𝑎1, 𝑥2, ..., 𝑎𝑡 − 1, 𝑥𝑡, и изучаем стратегии игры, которые зависят от этих последовательностей. Предполагается, что все последовательности в эмуляторе заканчиваются за конечное число временных шагов. Этот формализм приводит к большому, но конечному марковскому процессу принятия решений (MDP), в котором каждая последовательность представляет собой отдельное состояние. (страница 2 Mnih et al. 2013)

Вы должны точно понимать, что такое марковский процесс принятия решений и марковская цепочка, они имеют основополагающее значение для обучения подкреплению.

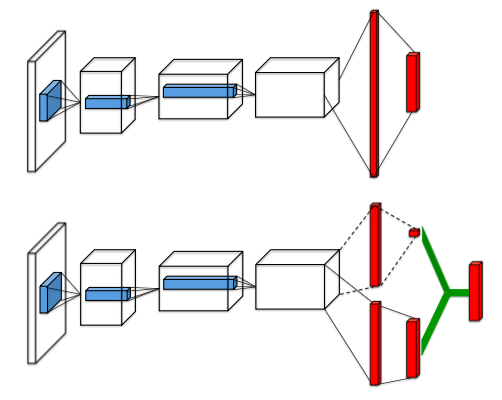
На странице 5 Mnih et al. В 2013 году авторы объясняют предварительную обработку кадров:

Работа непосредственно с необработанными кадрами Atari, представляющими собой изображения размером 210 × 160 пикселей с 128-цветовой палитрой, может потребовать вычислительных усилий, поэтому мы применяем базовый шаг предварительной обработки, направленный на уменьшение входной размерности. Необработанные кадры предварительно обрабатываются путем предварительного преобразования их представления RGB в оттенки серого и понижающей дискретизации его до изображения 110 × 84. Окончательное представление ввода получается путем обрезки области 84 × 84 изображения, которая приблизительно захватывает игровую область. Последний этап обрезки требуется только потому, что мы используем реализацию двумерных сверток на графическом процессоре из [...], которая предполагает квадратные входные данные. Для экспериментов, описанных в этой статье, функция *𝜙* [...] применяет эту предварительную обработку к последним 4 кадрам истории и суммирует их для получения входных данных для -функции.

Итак, давайте начнем с рассмотрения того, как может быть выполнена предварительная обработка. Я использовал тренажерный зал от OpenAi, чтобы обеспечить окружающую среду. Кадр, возвращаемый средой, имеет форму (210, 160, 3), где 3 обозначает цветовые каналы RGB. Такой кадр передается в FunctorFrameProcessor, который преобразует его в (84,84,1) кадр, где 1 указывает, что вместо трех каналов RGB существует один канал в градациях серого.

## 3. Дуэль сетей

Вместо сетевой архитектуры, описанной в Mnih et al. 2013 или Mnih et al. 2015 Я использовал дуэльную сетевую архитектуру, описанную в Wang et al. 2016.



И Mnih et al. 2015 и Wang et al. Дуэльная архитектура 2016 года имеет такую же низкоуровневую сверточную структуру:

Первый сверточный слой имеет 32 фильтра 8x8 с шагом 4, второй 64 фильтра 4x4 с шагом 2, а третий и последний сверточный слой состоит из 64 фильтров 3x3 с шагом 1.

В обычной архитектуре DQN (верхняя сеть на рисунке) последний скрытый уровень полностью связан и состоит из 512 выпрямительных блоков. Выходной слой представляет собой полностью связанный линейный слой с одним выходом для каждого действительного действия. (см. стр. 6 Mnih et al. 2015)

Эти результаты являются прогнозируемыми *𝑄*(*𝑠*,*𝑎*;*𝜃*)-значения для действия a в состоянии S.

Вместо прямого прогнозирования одного Q-значения для каждого действия, дуэльная архитектура разделяет конечный сверточный уровень на два потока, которые представляют функции значения и преимущества, которые предсказывают значение состояния 𝑉 (𝑠), которое зависит только от состояния, и преимущества действия 𝐴 (𝑠, 𝑎), которые зависят от состояния и соответствующего действия. На странице 2 Wang et al. В 2016 году авторы объясняют:

Интуитивно понятно, что дуэльная архитектура может узнать, какие состояния являются (или не являются) ценными, без необходимости изучать влияние каждого действия для каждого состояния. Это особенно полезно в тех штатах, где его действия никак не влияют на окружающую среду. В экспериментах мы демонстрируем, что дуэльная архитектура может быстрее определять правильное действие во время оценки политики, поскольку избыточные или подобные действия добавляются к проблеме обучения.

Значение состояния 𝑉 (𝑠) предсказывает, насколько хорошо быть в определенном состоянии 𝑠, а преимущество действия 𝐴 (𝑠, 𝑎) предсказывает, насколько хорошо выполнять действие 𝑎 в состоянии 𝑠, Я предлагаю вам взглянуть на рисунок 2 в Wang et al. 2016, чтобы лучше понять, на что научиться смотреть поток ценностей и преимуществ.

Далее мы должны объединить поток ценности и преимущества в -значения 𝑄 (𝑠, 𝑎). Это делается следующим образом (уравнение 9 в Wang et al. 2016):

*𝑄*(*𝑠*,*𝑎*)=*𝑉*(*𝑠*)+(*𝐴*(*𝑠*,*𝑎*)−1|**|∑*𝑎*′*𝐴*(*𝑠*,*𝑎*′))

Почему так сложно вместо простого добавления 𝑉 (𝑠) и 𝐴 (𝑠, 𝑎)? Предположим, что 𝑄 (𝑠, 𝑎) = 𝑉 (𝑠) + 𝐴 (𝑠, 𝑎):

Функция Q измеряет значение выбора конкретного действия в определенном состоянии. Функция значения 𝑉, которая является ожидаемым значением 𝑄 для всех возможных действий, 𝑉 = 𝐸 (𝑄), измеряет, насколько хорошо быть в этом конкретном состоянии.

Если вы объедините 𝐸 (𝑄) = 𝑉 и 𝑄 = 𝑉 + 𝐴, вы найдете 𝐸 (𝑄) = 𝐸 (𝑉) + 𝐸 (𝐴). Но 𝑉 не зависит от какого-либо действия, что означает 𝐸 (𝑉) = 𝑉, 𝐸 (𝑄) = 𝑉 + 𝐸 (𝐴) = 𝑉 и, следовательно, 𝐸 (𝐴) = 0. Ожидаемое значение преимущества 𝐴 (𝑠, ′ ′) над всеми возможными действиями 𝑎 ′ должно быть равно нулю. Ожидаемое значение выражения в скобках равно нулю, потому что мы вычитаем среднее из преимуществ из каждого преимущества:

*𝐸*(*𝐴*(*𝑠*,*𝑎*)−1|**|∑*𝑎*′*𝐴*(*𝑠*,*𝑎*′))=*𝐸*(*𝐴*−*𝐸*(*𝐴*))=*𝐸*(*𝐴*)−*𝐸*(*𝐸*(*𝐴*))=*𝐸*(*𝐴*)−*𝐸*(*𝐴*)=0

В ячейке ниже вы найдете код, который реализует эту архитектуру в тензорном потоке. Некоторые вещи, которые нужно иметь в виду: вы должны нормализовать значения входного пикселя до [0,1], разделив вход с 0xFF = 255. Причиной этого является то, что значения пикселей кадров, возвращаемых средой, являются uint8, которые могут хранить значения в диапазоне [0,255].

Кроме того, убедитесь, что вы правильно инициализировали веса! DQN использует функцию активации Relu, и правильным инициализатором является He et al. 2015 уравнение 10 (нажмите здесь для подробного объяснения). В тензорном потоке используйте tf.variance\_scaling\_initializer (документация) с масштабом = 2.

DeepMind использовал реализацию оптимизатора RMSProp, отличающуюся от оптимизатора tenorsflow (см. Стр. 23, пример 40 в Graves 2014). Прежде чем приступить к его реализации, я попробовал оптимизатор Adam, который дал многообещающие результаты без особого поиска гиперпараметров. Адам не был изобретен, когда Mnih et al. Был опубликован 2013 год, поэтому можно утверждать, что они могли бы использовать его вместо RMSProp, если бы он был изобретен ранее. С другой стороны, авторы этого блога сравнивают Momentum, RMSProp и Adam и утверждают:

Из вышеперечисленных трех вы можете найти импульс наиболее распространенным, несмотря на то, что Адам выглядит наиболее многообещающе на бумаге. Эмпирические результаты показали, что все эти алгоритмы могут сходиться к различным оптимальным локальным минимумам при одинаковых потерях. Однако SGD с импульсом, кажется, находит более плоские минимумы, чем Адам, в то время как адаптивные методы имеют тенденцию быстро сходиться к более резким минимумам. Плоские минимумы обобщаются лучше, чем острые.

Возможно, некоторые из этих различий могут быть смягчены использованием AdamW, которое я описал в этом посте в блоге (не имеет значения для DQN, поскольку не используется регуляризация). Сейчас я придерживаюсь Адама и, если найду время в будущем, я мог бы вернуться к этому, поскольку, возможно, стоит потратить некоторое время на игру с различными оптимизаторами и реализацию версии RMSProp, используемой DeepMind.

Редактировать: В более поздней статье DeepMind под названием «Радуга: объединение улучшений в изучении глубокого подкрепления» Хессель и соавт. RMSProp на 2017 год был заменен на Адама со скоростью обучения 0,0000625 (см. Таблицу 1). Этот уровень обучения близок к тому, что я нашел, работая хорошо для Breakout (0,00001), прежде чем читать Hessel et al. 2017.

Если вы сравните описанную выше дуэльную архитектуру с сетью, реализованной в следующей ячейке, вы обнаружите небольшую разницу. Вместо двух скрытых полностью связанных слоев с 512 выпрямительными блоками для каждого, значения и потока преимуществ, я закончил тем, что добавил четвертый сверточный слой с 1024 фильтрами. Выход имеет форму (1, 1, 1024) и затем разделяется на два потока с формами (1, 1, 512). Эта архитектура предлагается здесь и после выполнения некоторых тестов в среде Pong, которую сравнительно легко освоить для агента DQN, я обнаружил, что эта небольшая настройка позволяет вознаграждению увеличиваться немного раньше и выше.

class DQN(object):

"""Implements a Deep Q Network"""

# pylint: disable=too-many-instance-attributes

def \_\_init\_\_(self, n\_actions, hidden=1024, learning\_rate=0.00001,

frame\_height=84, frame\_width=84, agent\_history\_length=4):

"""

Args:

             n\_actions: целое число, количество возможных действий

             hidden: Integer, количество фильтров в конечном сверточном слое.

                     Это отличается от реализации DeepMind

             learning\_rate: Float, скорость обучения для оптимизатора Adam

             frame\_height: целое число, высота фрейма игры Atari

             frame\_width: целое число, ширина кадра игры Atari

             agent\_history\_length: Integer, количество кадров, собранных вместе для создания состояния

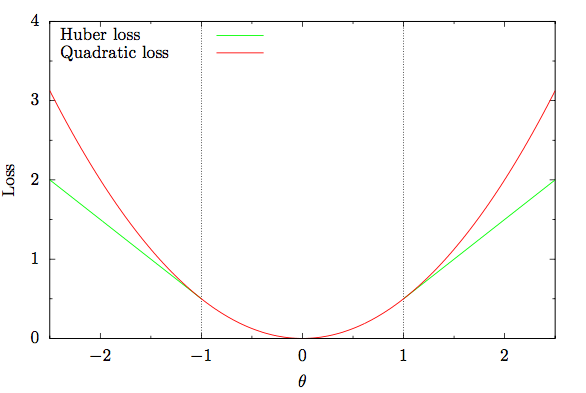
Еще один интересный момент: DeepMind использует функцию квадратичной стоимости с отсечением ошибок (см. Стр. 7 Mnih et al. 2015).

Мы также сочли полезным обрезать термин ошибки из обновления [...] в диапазоне от -1 до 1. Поскольку функция потери абсолютного значения | x | имеет производную от -1 для всех отрицательных значений x и производную от 1 для всех положительных значений x, ограничение квадрата ошибки до значения от -1 до 1 соответствует использованию функции потери абсолютного значения для ошибок за пределами (- 1,1) интервал. Эта форма отсечения ошибок дополнительно улучшила стабильность алгоритма.

Почему это улучшает стабильность алгоритма?

В глубоких сетях или периодических нейронных сетях градиенты ошибок могут накапливаться во время обновления и приводить к очень большим градиентам. Это, в свою очередь, приводит к большим обновлениям веса сети и, в свою очередь, к нестабильной сети. В крайнем случае значения весов могут стать настолько большими, что могут переполниться и привести к значениям NaN.

Эту так называемую проблему градиента взрыва можно в некоторой степени избежать, обрезая градиенты до определенного порогового значения, если они превышают его: \* Если истинный градиент больше критического значения 𝑥, просто предположите, что это 𝑥. \* Заметьте, что производная зеленой кривой не увеличивается (или не уменьшается) при 𝑥> 1 (или 𝑥 <−1). Отсечение ошибок может быть легко реализовано в тензорном потоке с помощью функции потерь Хьюбера tf.losses.huber\_loss.

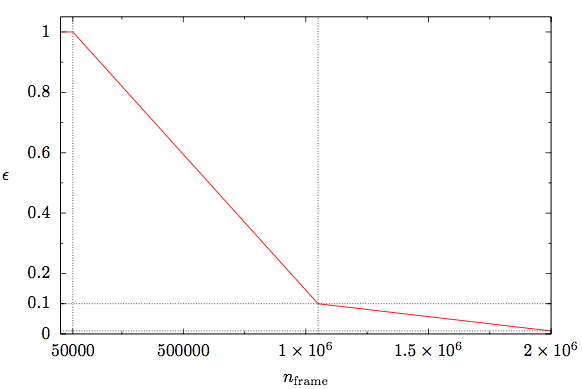


## Примечание: Если вы заинтересованы в создании высококачественных графиков для публикаций в формате eps (вектор) с использованием LaTeX для меток, аннотаций или даже формул, проверьте сценарии gnuplot в папке с изображениями.

## 4. Exploration-exploitation trade-off (Компромисс разведки и эксплуатации)

Если вы посмотрите на код в предыдущей ячейке, то обнаружите, что теперь мы можем прогнозировать действие, которое сеть считает наилучшим (self.best\_action), принимая аргумент максимума -стоимость. Но изначально агент не знает, как играть в игру. Если мы всегда используем и никогда не исследуем, всегда выбирая действие с самым высоким значением (жадным), агент будет придерживаться первой обнаруженной им стратегии, которая возвращает небольшое вознаграждение. Затем он может не продолжать изучать окружающую среду и не может продолжать учиться. -жадный алгоритм предлагает простое решение для этой проблемы: проще говоря, мы обычно выбираем действие, которое сети считают лучшим, но с вероятностью 𝜖 мы выбираем случайное действие. *𝜖* является функцией количества кадров, которые видел агент. Для первых 50000 кадров агент только исследует (𝜖 = 1). В течение следующих 1 миллиона кадров значение 𝜖 линейно уменьшается до 0,1, что означает, что агент начинает использовать все больше и больше. DeepMind затем сохраняет 𝜖 = 0.1, однако мы решили уменьшить его до 0.01 = 0.01

по оставшимся кадрам, как предложено в OpenAi Baselines for DQN (на графике максимальное количество кадров составляет 2 миллиона для демонстрационных целей).



Метод get\_action в ячейке ниже реализует это поведение: сначала он вычисляет 𝜖

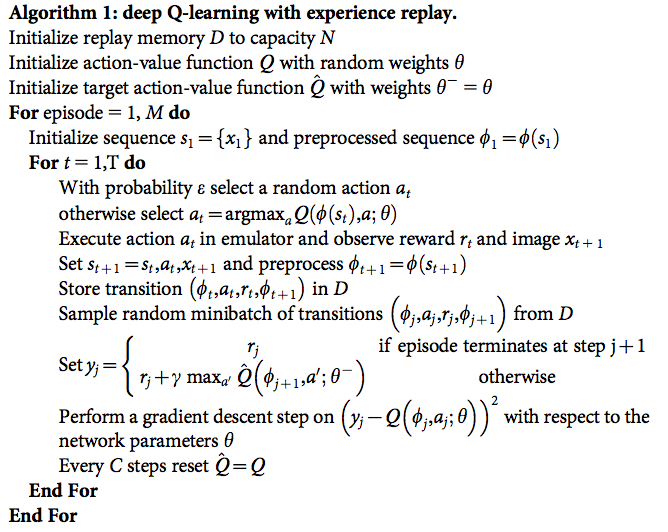
из номера текущего кадра, а затем либо возвращает случайное действие (с вероятностью 𝜖), либо действие, которое DQN считает лучшим. Переменные в конструкторе - это наклоны и точки пересечения для уменьшения 𝜖

показано на графике выше.

class ExplorationExploitationScheduler(object):

"" "Определяет действие в соответствии с жадной стратегией эпсилона с эпсилоном отжига" ""

Теперь мы знаем, как DQN предсказывает лучшие действия, и у нас есть простой ответ на дилемму разведки и эксплуатации. Итак, что еще нам нужно, чтобы это работало? Давайте посмотрим на алгоритм, представленный на странице 7 в Mnih et al. 2015.



## Давайте пройдемся по алгоритму шаг за шагом:

## • Мы еще не знаем, что такое повторная память D.

## • Действие-функция Q - это наша сеть DQN, которую мы уже внедрили.

## • Нам нужно обсудить, зачем нужна вторая Q-сеть, называемая целевой функцией-значением.

## • В начале каждого эпизода последовательность инициализируется. Это реализуется путем объединения четырех (в градациях серого) кадров вместе, как обсуждалось выше.

## • Мы обсудили, как выбирается действие ((𝜖-жадное).

## • Когда действие выполнено, среда возвращает следующий кадр и вознаграждение за это действие. gym дополнительно возвращает логическое значение с именем Terminal, которое указывает, закончилась ли игра, и словарь, содержащий количество жизней, оставленных агентом (ale.lives).

## • Мы еще не знаем, что означает хранить переход в памяти воспроизведения D. Список [state, action, reward, terminal, new\_state] называется переходом. Состояние - это четыре кадра, сложенных вместе. new\_state создается путем укладки наблюдаемого кадра (после выполнения действия) в состояние и удаления самого старого кадра. Вы увидите реализацию позже.

## • Мы должны обсудить, как мини-пакет восстанавливается из памяти воспроизведения и как выполняется шаг спуска градиента.

## • Наконец, мы должны посмотреть, почему и как целевая сеть Q сбрасывается в основную сеть Q.

## Давайте продолжим с памятью воспроизведения

## 5. Replay memory (Память воспроизведения)

Во-вторых, обучение непосредственно из последовательных выборок неэффективно из-за сильной корреляции между выборками; рандомизация выборок нарушает эти корреляции и, следовательно, уменьшает дисперсию обновлений. В-третьих, при изучении на-полисе текущие параметры определяют следующую выборку данных, на которой обучаются параметры. Например, если максимизирующим действием является перемещение влево, то в обучающих выборках будут преобладать выборки с левой стороны; если максимизирующее действие затем переключается вправо, то распределение обучения также переключается. Легко увидеть, как могут возникать нежелательные петли обратной связи, и параметры могут застрять в плохом локальном минимуме или даже катастрофически расходиться. (страница 5 Mnih et al. 2013)

Это означает, что когда мы выбираем действие и выполняем шаг для получения вознаграждения, сеть не учится на этом последнем шаге, а добавляет переход в память воспроизведения. Затем он извлекает случайную мини-партию из памяти воспроизведения для выполнения шага градиентного спуска.

В памяти воспроизведения хранится последний миллион переходов. Давайте вспомним, что переход - это [state, action, reward, terminal, new\_state]. Поэтому нам нужно сохранить последний миллион state, action, reward, terminal, new\_state. Если вы помните, что state и new\_state составляют четыре кадра каждый, это будет восемь миллионов кадров. Однако, поскольку new\_state создается путем укладки самого нового кадра поверх состояния и удаления самого старого кадра, new\_state и state совместно используют три кадра. Кроме того, new\_state перехода i будет состоянием перехода i + 1. Это означает, что достаточно сохранить последний миллион кадров (84 \* 84 пикселей) в виде тензора (1 миллион, 84, 84), а затем вырезать четыре кадра из этого тензора, когда нам понадобится состояние или new\_state.

С миллионами кадров 84 на 84 пикселя, которые должны быть сохранены в памяти вашего компьютера, мы должны рассмотреть, в каком типе данных мы их храним. Среда возвращает кадры со значениями пикселей, сохраненными как uint8, которые могут иметь значения в диапазоне от 0 до 255. Uint8 требуется 8 битов. Сеть ожидает ввод tf.float32 со значениями пикселей от 0 до 1 (который занимает в четыре раза больше места, чем uint8). Поскольку мы хотим уменьшить требования к памяти, мы храним кадры в uint8 и делим их на 255, прежде чем передавать их в сеть.

При реализации этой версии памяти воспроизведения мы рассмотрели этот код и в итоге реализовали память воспроизведения с некоторыми изменениями, которые делают код более понятным.

Давайте посмотрим на класс ReplayMemory ниже. В конструкторе мы предварительно выделяем память для фреймов, действий, наград, состояний терминала, а также для состояний и новых состояний мини-пакета.

В методе add\_experience кадры и т. Д. Записываются в self.frames с индексом self.current, который затем увеличивается на 1. Когда self.current достигает размера памяти воспроизведения (один миллион), он сбрасывается в ноль, чтобы перезаписать самые старые кадры. Метод \_get\_state вырезает четыре кадра из self.frames и возвращает их как состояние.

Чтобы понять, что делает метод \_get\_valid\_indices, нам нужно понять, что такое недопустимый индекс. Мы храним все кадры, которые агент видит в self.frames. Когда игра заканчивается (terminal = True) с индексом i, кадр с индексом i принадлежит другому эпизоду, чем кадр с i + 1. Мы хотим избежать создания состояния с кадрами из двух разных эпизодов. То же самое может произойти с индексом self.current.

Наконец, нам нужно убедиться, что индекс не меньше, чем количество кадров, собранных вместе, чтобы создать состояние (self.agent\_history\_length = 4), чтобы можно было вырезать состояние и new\_state из массива.

Метод \_get\_valid\_indices находит 32 (размер мини-пакета) допустимых индексов. Метод get\_minibatch возвращает переходы для этих индексов. Обратите внимание, что нам нужно транспонировать self.states и self.new\_states перед их возвратом: DQN ожидает ввод измерения [Нет, 84,84,4], тогда как \_get\_state возвращает состояние измерения [4,84,84]

Теперь мы знаем 1) почему память воспроизведения значительно улучшает стабильность алгоритма, 2) как сохранить переход в памяти воспроизведения и 3) как возвращается мини-пакет.

class ReplayMemory(object):

"""Replay Memory that stores the last size=1,000,000 transitions"""

## 6. Целевая сеть и обновление параметров

Зачем нам нужны две сети: функция «значение действия» и целевая функция «значение действия»?

Помните, что перед обновлением параметров сети мы рисуем мини-пакет с 32 переходами. Для простоты рассмотрим один переход сейчас. Он состоит из состояния, действия, которое было выполнено в состоянии, полученного вознаграждения, new\_state и bool, сообщающего, закончился ли эпизод.

Мы выполняем шаг градиентного спуска: основная сеть просматривает состояние и оценивает значения 𝑄prediction, которые говорят о том, насколько хорошо каждое действие. Однако мы хотим, чтобы -значения следовали приведенному выше уравнению Беллмана. Поэтому мы вычисляем значения 𝑄target в соответствии с уравнением Беллмана (как нам хотелось бы, чтобы значения 𝑄) были, а затем сравниваем оценки 𝑄prediction с целями 𝑄target. Для простоты рассмотрим функцию квадратичных потерь вместо функции потерь Хьюбера:

*𝐿*=1/2(*𝑄*prediction−*𝑄*target)2

Это гарантирует, что мы регрессируем текущие значения 𝑄prediction для состояния к значениям Qtarget, заданным уравнением Беллмана.

Qprediction рассчитывается в классе DQN (self.q\_values). Qprediction зависит от текущего состояния в нарисованном нами мини-пакете и от параметров 𝜃 сети, которая его оценивает.

Значение 𝑄target рассчитывается в соответствии с уравнением Беллмана. Это сумма немедленного вознаграждения receivedr, полученного за выполнение действия 𝑎 в состоянии (s (действие и состояние из мини-пакета), и максимального значения 𝑄 по всем возможным действиям 𝑎′a ′ в s′s ′ (new\_state из мини-пакета) :

*𝑄*target(*𝑠*,*𝑎*)=*𝑟*+*𝛾*max(*𝑄*(*𝑠*′,*𝑎*′))

Это делается не в классе DQN, а в методе обучения ниже. Затем вычисленное значение передается заполнителю с именем self.target\_q в классе DQN. Там определяется функция потерь и выполняется шаг градиентного спуска.

## Итак, теперь, когда мы понимаем, как обновляются параметры, зачем использовать две сети?

## Проблема заключается в том, что оба параметра Qprediction и 𝑄target зависят от одних и тех же параметров θ, если используется только одна сеть. Это может привести к нестабильности при регрессии 𝑄prediction в направлении 𝑄target, поскольку «цель движется». Мы обеспечиваем «фиксированную цель», вводя вторую сеть с фиксированными и только иногда обновляемыми параметрами, которые оценивают целевые значения 𝑄.

## На странице 1 Mnih et al. 2015 авторы объясняют:

## Известно, что обучение подкреплению нестабильно или даже расходится, когда для представления функции-значения (также известной как Q) используется аппроксиматор нелинейной функции, такой как нейронная сеть. Эта нестабильность имеет несколько причин: корреляции, присутствующие в последовательности наблюдений, тот факт, что небольшие обновления Q могут значительно изменить политику и, следовательно, изменить распределение данных, а также корреляции между значениями действия [...] и целью ценности [...]. Мы решаем эти проблемы с помощью нового варианта Q-обучения, в котором используются две ключевые идеи. Во-первых, мы использовали биологически вдохновленный механизм, называемый повторением опыта, который рандомизирует данные, тем самым устраняя корреляции в последовательности наблюдений и сглаживая изменения в распределении данных [...]. Во-вторых, мы использовали итеративное обновление, которое корректирует значения действия (Q) в соответствии с целевыми значениями, которые обновляются только периодически, тем самым уменьшая корреляции с целью.

## Поэтому они использовали одну сеть для прогнозирования значения Qprediction, а другую фиксированную сеть - для прогнозирования значения -Qtarget. Основная сеть оптимизируется во время шага градиентного спуска, и каждые 10000 шагов параметры основной сети копируются в целевую сеть. Имейте в виду, что частота обновления сети измеряется количеством выбранных выбранных действий / кадров (код DeepMind), а не количеством обновлений параметров, которые происходят каждые четыре кадра (Mnih et al. 2015).

## Есть еще одно очень мощное улучшение, которое называется Double Q-Learning.

## 7. Двойное Q-Learning

Наблюдалось, что DQN оценивает нереально высокие значения 𝑄. Причина этого заключается в том, что уравнение Беллмана включает шаг максимизации по оценочным значениям действия, который имеет тенденцию предпочитать завышенные значения заниженным значениям (см. Van Hasselt et al. 2016, стр. 1).

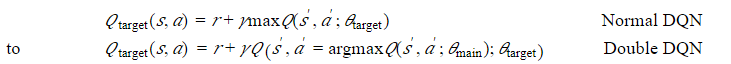
Авторы объясняют:

Если бы все значения были одинаково выше, то относительные предпочтения действий сохранялись, и мы не ожидали бы, что итоговая политика будет хуже. [...] Если, однако, завышенные оценки не являются единообразными и не сконцентрированы в тех штатах, о которых мы хотим узнать больше, то они могут негативно повлиять на качество итоговой политики. [...] Затем мы показываем, что этот алгоритм не только дает более точные оценки стоимости, но и приводит к гораздо более высоким показателям в нескольких играх. Это демонстрирует, что переоценка DQN действительно приводила к более слабым политикам и что было бы полезно их уменьшить

Оцененные значения 𝑄 являются зашумленными. Предположим, что истинное -значение равно 0 для всех действий. Но из-за шумной оценки некоторые -значения могут быть слегка положительными, другие - слегка отрицательными. Операция max в уравнении Беллмана, тем не менее, всегда выбирает небольшие положительные значения, несмотря на то, что эти действия на самом деле не лучше. Таким образом, оценка -значений смещается в сторону больших значений. Как мы это исправим? Вместо того, чтобы оценивать -значения в следующем состоянии 𝑄 (𝑠 ′, 𝑎 ′) Q (s ′, a ′) только с целевой сетью, мы используем основную сеть, чтобы оценить, какое действие является наилучшим, и затем запрашиваем цель сеть, как высоко значение 𝑄 для этого действия. Таким образом, основная сеть будет по-прежнему предпочитать действие с небольшим положительным значением 𝑄, но из-за шумной оценки целевая сеть будет прогнозировать небольшое положительное или небольшое отрицательное значение 𝑄 для этого действия и в среднем прогнозируемое значение --Q - значения будут ближе к 0.

Математически причина переоценки состоит в том, что ожидание максимума больше или равно максимуму ожидания Ван Хасселта 2013, теорема 1.

Уравнение Беллмана изменяется от



The main network estimates which action 𝑎′

(in the next state 𝑠′) is best (that is the argmax𝑄(𝑠′,𝑎′;𝜃main) part). The target network then estimates what the 𝑄-value for that action is. This 𝑄-value has to be discounted with 𝛾 and is then added to the reward 𝑟 the agent got for action 𝑎 (not 𝑎′

).

I know that this equation might look discouraging. So let's describe it again in words:

Normal DQN: Ask the target network for the highest 𝑄

-Value. If the noisy 𝑄-values are for example (0.1,−0.1) for actions with index 0 and 1 respectively, the target 𝑄-network will answer 0.1

.

Основная сеть оценивает, какое действие a′ (в следующем состоянии 𝑠′) является наилучшим (то есть часть argmax𝑄 (𝑠 ′, 𝑎 ′; 𝜃main)). Затем целевая сеть оценивает значение Q для этого действия. Это -значение должно быть дисконтировано с 𝛾, а затем добавлено к награде 𝑟, которую агент получил за действие 𝑎 (не 𝑎′a ′).

Я знаю, что это уравнение может выглядеть обескураживающим. Итак, давайте опишем это снова словами:

Обычный DQN: запросите у целевой сети самое высокое значение 𝑄. Если зашумленные значения 𝑄 равны, например, (0.1, -0.1 для действий с индексами 0 и 1 соответственно, целевая 𝑄 -сеть ответит 0.1

Двойной DQN: спросите основную сеть, какое действие имеет наибольшее значение 𝑄. Если зашумленные значения 𝑄 равны, например, (0.1, -0.1) для действий с индексами 0 и 1 соответственно, основная сеть ответит, что действие с индексом 0 имеет наибольшее значение 𝑄. Затем мы спрашиваем целевую сеть, которая имеет другой шум, каково значение 𝑄 для действия с выбранным индексом (в этом примере 0). Давайте предположим, что оценки целевой сети с шумом (-0,05,0,3) будут отвечать -0,05-0,05.

Это решает проблему завышенных 𝑄Q-значений, потому что две сети имеют разные шумы, и отклонение в сторону немного больших шумных -значений отменяется.

Еще одна вещь: если игра окончена (терминал = Истина) из-за проигранной или выигранной повестки дня, следующего состояния не существует, а значение 𝑄target является просто наградой 𝑟r.

Посмотрите на реализацию в ячейке ниже.

def learn(session, replay\_memory, main\_dqn, target\_dqn, batch\_size, gamma):

"""

Args:

session: A tensorflow sesson object

replay\_memory: A ReplayMemory object

main\_dqn: A DQN object

target\_dqn: A DQN object

batch\_size: Integer, Batch size

gamma: Float, discount factor for the Bellman equation

Returns:

loss: The loss of the minibatch, for tensorboard

Draws a minibatch from the replay memory, calculates the

target Q-value that the prediction Q-value is regressed to.

Then a parameter update is performed on the main DQN.

"""

Как я уже упоминал, параметры основной сети периодически копируются каждые 10 000 шагов в целевую сеть. Это реализовано в ячейке ниже.

Class TergetNetworkUpdater(object):

Функция в ячейке ниже создает GIF из последовательности переданных ей кадров.

def generate\_gif(frame\_number, frames\_for\_gif, reward, path):

Среда обучения обеспечивается тренажерным залом OpenAi. Очень важно, чтобы у вас была правильная версия окружения. Например, BreakoutDeterministic-v3 имеет шесть действий, тогда как BreakoutDeterministic-v4 имеет минимальный набор из четырех действий, которые DeepMind использовал в xitari. Дополнительные действия усложняют задачу обучения для агента, что может существенно изменить оценку. Если вы хотите узнать количество действий и их значение, введите env.action\_space.n и env.unwrapped.get\_action\_meanings ().

Есть две дополнительные небольшие корректировки, которые мы должны обсудить:

Когда жизнь потеряна, мы сохраняем Terminal\_life\_lost = True в памяти воспроизведения. Создайте новую записную книжку, создайте среду Breakout, в цикле повторите случайные действия или не выполняйте никаких действий и напечатайте вознаграждение и количество жизней, которые имеет агент.

frame = env.reset()

for i in range(1000):

new\_frame, reward, terminal, info = env.step(0)

print(reward, terminal, info['ale.lives'])

Вы увидите, что нет никакого наказания (награда равна 0), когда жизнь потеряна. Это очень помогает агенту избежать потери жизни, если вы считаете потерю жизни концом эпизода. Тем не менее, мы делаем это только в памяти воспроизведения, поскольку не хотим сбрасывать игру после потери первой жизни. Следовательно, необходимы два состояния терминала: Terminal и Terminal\_life\_lost: один для сброса игры, другой для памяти воспроизведения. Эта корректировка помогла агенту улучшить среднее вознаграждение чуть выше 50 до примерно 140 в Breakout!

Давайте обернем среду гимнастики в классе Atari, который заботится о наложении кадров друг на друга, чтобы создать состояния, сбрасывая среду после окончания эпизода и проверяя, была ли потеряна жизнь после того, как был сделан какой-либо шаг. Вы найдете реализацию в ячейке ниже.

Во время оценки в начале каждого эпизода действие 1 («ОГОНЬ») повторяется для случайного числа шагов от 1 до no\_op\_steps = 10. Это гарантирует, что агент каждый раз запускается в другой ситуации и поэтому не может просто выучить фиксированную последовательность действий. Мних и др. В 2015 году используется случайное число от 1 до 30 действий NOOP (см. Стр. 10, таблица 1). Тем не менее, в Breakout ничего не происходит, если вы не стреляете первыми. Если в игре есть мяч, «ОГОНЬ» ничего не делает. Поэтому я начал со случайного числа «ОГОННЫХ» действий. Кроме того, я ограничил случайное число начальных действий «ОГОНЬ» до 10. При экспериментировании с большими числами я обнаружил, что первая жизнь обычно уже теряется, когда агент начинает двигаться. Вы можете изменить это, если хотите поэкспериментировать с другой средой.

class Atari(object):

"""Wrapper for the environment provided by gym"""

634/5000

Мних и др. 2013 и Mnih et al. 2015 год обрезал награды, объяснив, что «Поскольку шкала оценок сильно варьируется от игры к игре, мы установили все положительные награды равными 1, а все отрицательные награды - равными -1, оставив нулевые награды без изменений. Отсечение вознаграждений таким образом ограничивает масштаб из производных ошибок и облегчает использование одной и той же скорости обучения в нескольких играх. В то же время это может повлиять на производительность нашего агента, поскольку он не может различать награды разной величины ". Ограничение наград делает тренировку сходящейся быстрее и повышает производительность на примере Breakout.

def clip\_reward(reward):

if reward > 0:

return 1

elif reward == 0:

return 0

else:

return -1

В ячейке ниже я объявляю некоторые константы, которые определяют обучающее поведение агента:tf.reset\_default\_graph()

​

# Control parameters

MAX\_EPISODE\_LENGTH = 18000 # Equivalent of 5 minutes of gameplay at 60 frames per second

EVAL\_FREQUENCY = 200000 # Number of frames the agent sees between evaluations

EVAL\_STEPS = 10000 # Number of frames for one evaluation

NETW\_UPDATE\_FREQ = 10000 # Number of chosen actions between updating the target network.

# According to Mnih et al. 2015 this is measured in the number of

# parameter updates (every four actions), however, in the

# DeepMind code, it is clearly measured in the number

# of actions the agent choses

DISCOUNT\_FACTOR = 0.99 # gamma in the Bellman equation

REPLAY\_MEMORY\_START\_SIZE = 50000 # Number of completely random actions,

# before the agent starts learning

MAX\_FRAMES = 30000000 # Total number of frames the agent sees

MEMORY\_SIZE = 1000000 # Number of transitions stored in the replay memory

NO\_OP\_STEPS = 10 # Number of 'NOOP' or 'FIRE' actions at the beginning of an

# evaluation episode

UPDATE\_FREQ = 4 # Every four actions a gradient descend step is performed

HIDDEN = 1024 # Number of filters in the final convolutional layer. The output

# has the shape (1,1,1024) which is split into two streams. Both

# the advantage stream and value stream have the shape

# (1,1,512). This is slightly different from the original

# implementation but tests I did with the environment Pong

# have shown that this way the score increases more quickly

LEARNING\_RATE = 0.00001 # Set to 0.00025 in Pong for quicker results.

# Hessel et al. 2017 used 0.0000625

BS = 32 # Batch size

​

PATH = "output/" # Gifs and checkpoints will be saved here

SUMMARIES = "summaries" # logdir for tensorboard

RUNID = 'run\_1'

os.makedirs(PATH, exist\_ok=True)

os.makedirs(os.path.join(SUMMARIES, RUNID), exist\_ok=True)

SUMM\_WRITER = tf.summary.FileWriter(os.path.join(SUMMARIES, RUNID))

​

atari = Atari(ENV\_NAME, NO\_OP\_STEPS)

​

print("The environment has the following {} actions: {}".format(atari.env.action\_space.n,

atari.env.unwrapped.get\_action\_meanings()))

Let's create the networks:

# main DQN and target DQN networks:

with tf.variable\_scope('mainDQN'):

MAIN\_DQN = DQN(atari.env.action\_space.n, HIDDEN, LEARNING\_RATE) # (★★)

with tf.variable\_scope('targetDQN'):

TARGET\_DQN = DQN(atari.env.action\_space.n, HIDDEN) # (★★)

​

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

​

MAIN\_DQN\_VARS = tf.trainable\_variables(scope='mainDQN')

TARGET\_DQN\_VARS = tf.trainable\_variables(scope='targetDQN')

### Tensorboard

Настройка тензорной сводки для потери, среднего вознаграждения, оценки и сетевых параметров для наблюдения за процессом обучения:

LAYER\_IDS = ["conv1", "conv2", "conv3", "conv4", "denseAdvantage",

"denseAdvantageBias", "denseValue", "denseValueBias"]

​

# Scalar summaries for tensorboard: loss, average reward and evaluation score

with tf.name\_scope('Performance'):

LOSS\_PH = tf.placeholder(tf.float32, shape=None, name='loss\_summary')

LOSS\_SUMMARY = tf.summary.scalar('loss', LOSS\_PH)

REWARD\_PH = tf.placeholder(tf.float32, shape=None, name='reward\_summary')

REWARD\_SUMMARY = tf.summary.scalar('reward', REWARD\_PH)

EVAL\_SCORE\_PH = tf.placeholder(tf.float32, shape=None, name='evaluation\_summary')

EVAL\_SCORE\_SUMMARY = tf.summary.scalar('evaluation\_score', EVAL\_SCORE\_PH)

​

PERFORMANCE\_SUMMARIES = tf.summary.merge([LOSS\_SUMMARY, REWARD\_SUMMARY])

​

# Histogramm summaries for tensorboard: parameters

with tf.name\_scope('Parameters'):

ALL\_PARAM\_SUMMARIES = []

for i, Id in enumerate(LAYER\_IDS):

with tf.name\_scope('mainDQN/'):

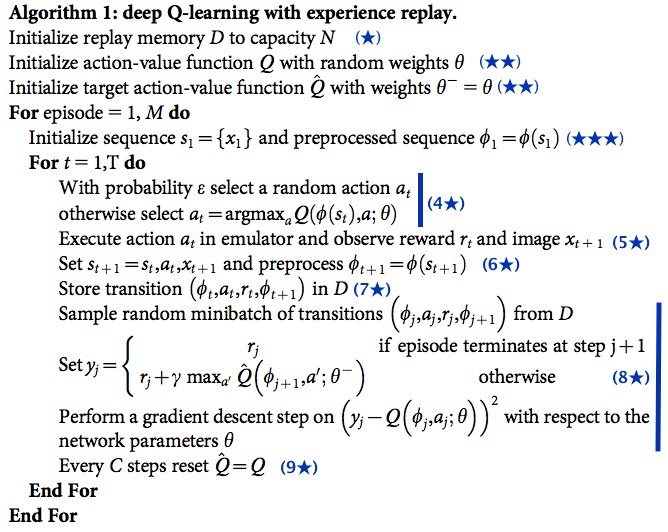
MAIN\_DQN\_KERNEL = tf.summary.histogram(Id, tf.reshape(MAIN\_DQN\_VARS[i], shape=[-1]))

ALL\_PARAM\_SUMMARIES.extend([MAIN\_DQN\_KERNEL])

PARAM\_SUMMARIES = tf.summary.merge(ALL\_PARAM\_SUMMARIES)

## 8. Собираем все вместе: DQN

Теперь у нас есть все, что нужно, чтобы наконец реализовать Алгоритм 1 от Mnih et al. 2015. Звезды помогут вам определить, где реализован каждый шаг.



Пройдите реализацию в ячейке ниже. Вы обнаружите, что после каждой эпохи обучения в 200 000 кадров (50 000 обновлений параметров) агент оценивается в течение 10 000 кадров. Конспекты записываются в логдир для тензорной доски.

Я описал эксперименты, которые я провел, используя этот ноутбук здесь. Не стесняйтесь играть с ноутбуком, вносить изменения и экспериментировать с различными средами. Если вы можете улучшить его или получить опыт работы с другой игрой Atari, пожалуйста, оставьте комментарий здесь, я хотел бы узнать об этом.

## 10. Тренируй сам

Если вы хотите обучить сеть самостоятельно, установите TRAIN на True в первой ячейке этого ноутбука.

Подумайте о том, чтобы сделать ваш компьютер доступным удаленно, как я описал в этом сообщении в блоге: Удаленный доступ к вашей станции Deep Learning и настройка wake on lan

Вы можете преобразовать блокнот в скрипт на python, используя jupyter-nbconvert --to script DQN.ipynb, а затем запустить его в сеансе tmux, который я описал здесь: jupyter и тензорная доска в tmux. Преимущество этого заключается в том, что вы можете отсоединиться от сеанса tmux и подключиться к нему удаленно (например, с ноутбука), чтобы проверить ход работы агентов.

Если вы хотите использовать тензорную доску для мониторинга улучшений в сети, введите тензорную доску --logdir = summary в терминале, в котором активирована соответствующая виртуальная среда. Откройте браузер и перейдите по адресу http: // localhost: 6006. Это работает и удаленно.

if TRAIN:

train()

## 9. Использование тренированной сети

save\_files\_dict = {

'BreakoutDeterministic-v4':("trained/breakout/", "my\_model-15845555.meta"),

'PongDeterministic-v4':("trained/pong/", "my\_model-3217770.meta")

}

if not TRAIN:

Ссылки:

<http://ai.lector.ru/?go=lection07>