1. Инициализировать Q(s, a)

2. Повторять для каждой игры

3. Инициализировать s

4. Повторять для каждого шага

5. Выбрать a по s (ε-жадную)

6. Выполнить a, найти r, s’

7. Q(s, a) = Q(s, a) + α[r + γmaxa’Q(s’,a’)-Q(s,a)]

8. s = s’

9. Пока s не станет финальным

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

# 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*

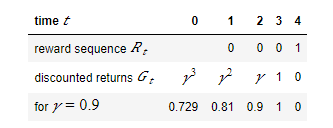
Авторы также объясняют:

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.

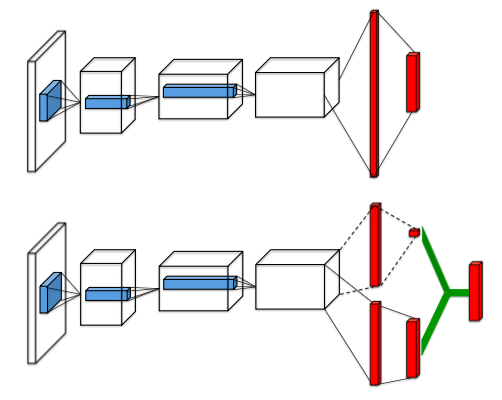
На странице 5 Mnih et al. В 2013 году авторы объясняют предварительную обработку кадров:

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

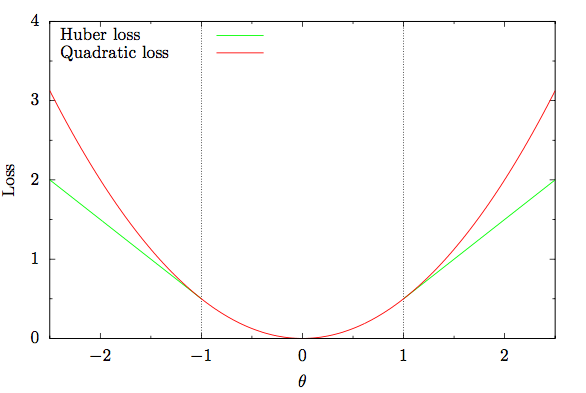
Еще один интересный момент: DeepMind использует функцию квадратичной стоимости с отсечением ошибок (см. Стр. 7 Mnih et al. 2015).

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.

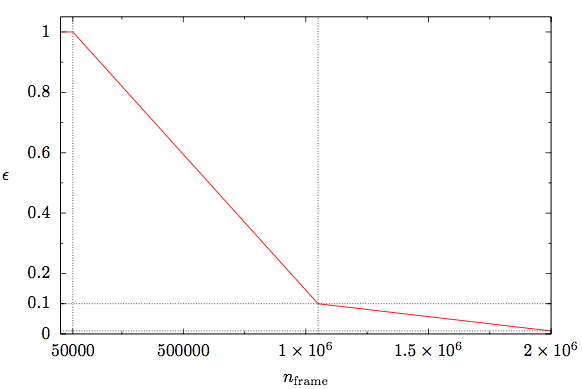


## Примечание: Если вы заинтересованы в создании высококачественных графиков для публикаций в формате eps (вектор) с использованием LaTeX для меток, аннотаций или даже формул, проверьте сценарии gnuplot в папке с изображениями.

## 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).



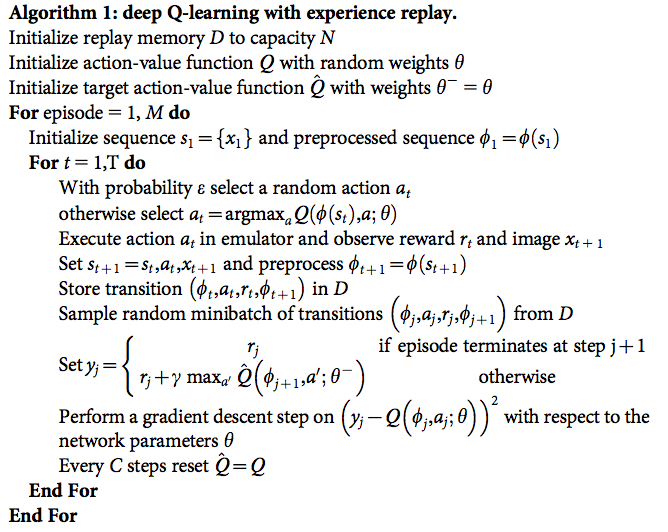
Метод get\_action в ячейке ниже реализует это поведение: сначала он вычисляет 𝜖

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)

Second, learning directly from consecutive samples is inefficient, due to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates. Third, when learning on-policy the current parameters determine the next data sample that the parameters are trained on. For example, if the maximizing action is to move left then the training samples will be dominated by samples from the left-hand side; if the maximizing action then switches to the right then the training distribution will also switch. It is easy to see how unwanted feedback loops may arise and the parameters could get stuck in a poor local minimum, or even diverge catastrophically. ([page 5 of Mnih et al. 2013](https://arxiv.org/abs/1312.5602))

This means that when we choose an action and perform a step to receive a reward, the network does not learn from this last step but rather adds the transition to the replay memory. It then draws a random minibatch from the replay memory to perform a gradient descent step.

The replay memory stores the last one million transitions. Let's recall that a transition is [state, action, reward, terminal, new\_state]. We therefore need to store the last one million state, action, reward, terminal and new\_state. If you remember that state and new\_state are four frames each, that would be eight million frames. However, since new\_state is created by stacking the newest frame on top of state and deleting the oldest frame, new\_state and state share three frames. Furthermore, new\_state of transition i will be state of transition i+1. This means that it is sufficient to store the last one million frames (84\*84 pixels) as a (1 million, 84, 84) tensor and then slicing four frames out of this tensor when we need a state or new\_state.

With one million frames of 84 by 84 pixels that need to be stored in your computers memory, we need to consider in what datatype we store them. The environment returns frames with pixel values stored as uint8 which can have values ranging from 0 to 255. A uint8 needs 8 bits. The network expects a tf.float32 input with pixel values between 0 and 1 (which takes four times more space than a uint8). Since we want to reduce the memory requirements, we store the frames in uint8 and divide them by 255 before passing them to the network.

When implementing this version of replay memory, we looked at this [code](https://github.com/tambetm/simple_dqn/blob/master/src/replay_memory.py) and ended up implementing the replay memory with some adjustments that make the code more understandable.

Let's look at the ReplayMemory class below. In the constructor, we pre-allocate the memory for the frames, the actions, the rewards, the terminal states and also for the states and new states of the minibatch.

In the add\_experience method the frames etc. are written into self.frames at index self.current which is then increased by 1. When self.current reaches the size of the replay memory (one million), it is reset to zero to overwrite the oldest frames. The method \_get\_state slices four frames out of self.frames and returns them as a state.

To understand what the method \_get\_valid\_indices does, we need to understand what an invalid index is. We store all frames the agent sees in self.frames. When a game terminates (terminal=True) at index i, frame at index i belongs to a different episode than the frame at i+1. We want to avoid creating a state with frames from two different episodes. The same thing can happen at the index self.current.

Finally we need to make sure that an index is not smaller than the number of frames stacked toghether to create a state (self.agent\_history\_length=4), so that a state and new\_state can be sliced out of the array.

The method \_get\_valid\_indices finds 32 (size of minibatch) valid indices. The method get\_minibatch returns the transitions for those indices. Pay attention that we need to transpose self.states and self.new\_states before returning them: the DQN expects an input of the dimension [None,84,84,4] whereas \_get\_state returns a state of the dimension [4,84,84]

We now know 1) why a replay memory greatly improves the stability of the algorithm, 2) how to store a transition in the replay memory and 3) how a minibatch is returned.

class ReplayMemory(object):

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

def \_\_init\_\_(self, size=1000000, frame\_height=84, frame\_width=84,

agent\_history\_length=4, batch\_size=32):

"""

Args:

size: Integer, Number of stored transitions

frame\_height: Integer, Height of a frame of an Atari game

frame\_width: Integer, Width of a frame of an Atari game

agent\_history\_length: Integer, Number of frames stacked together to create a state

batch\_size: Integer, Number if transitions returned in a minibatch

"""

self.size = size

self.frame\_height = frame\_height

self.frame\_width = frame\_width

self.agent\_history\_length = agent\_history\_length

self.batch\_size = batch\_size

self.count = 0

self.current = 0

# Pre-allocate memory

self.actions = np.empty(self.size, dtype=np.int32)

self.rewards = np.empty(self.size, dtype=np.float32)

self.frames = np.empty((self.size, self.frame\_height, self.frame\_width), dtype=np.uint8)

self.terminal\_flags = np.empty(self.size, dtype=np.bool)

# Pre-allocate memory for the states and new\_states in a minibatch

self.states = np.empty((self.batch\_size, self.agent\_history\_length,

self.frame\_height, self.frame\_width), dtype=np.uint8)

self.new\_states = np.empty((self.batch\_size, self.agent\_history\_length,

self.frame\_height, self.frame\_width), dtype=np.uint8)

self.indices = np.empty(self.batch\_size, dtype=np.int32)

def add\_experience(self, action, frame, reward, terminal):

"""

Args:

action: An integer between 0 and env.action\_space.n - 1

determining the action the agent perfomed

frame: A (84, 84, 1) frame of an Atari game in grayscale

reward: A float determining the reward the agend received for performing an action

terminal: A bool stating whether the episode terminated

"""

if frame.shape != (self.frame\_height, self.frame\_width):

raise ValueError('Dimension of frame is wrong!')

self.actions[self.current] = action

self.frames[self.current, ...] = frame

self.rewards[self.current] = reward

self.terminal\_flags[self.current] = terminal

self.count = max(self.count, self.current+1)

self.current = (self.current + 1) % self.size

def \_get\_state(self, index):

if self.count is 0:

raise ValueError("The replay memory is empty!")

if index < self.agent\_history\_length - 1:

raise ValueError("Index must be min 3")

return self.frames[index-self.agent\_history\_length+1:index+1, ...]

def \_get\_valid\_indices(self):

for i in range(self.batch\_size):

while True:

index = random.randint(self.agent\_history\_length, self.count - 1)

if index < self.agent\_history\_length:

continue

if index >= self.current and index - self.agent\_history\_length <= self.current:

continue

if self.terminal\_flags[index - self.agent\_history\_length:index].any():

continue

break

self.indices[i] = index

def get\_minibatch(self):

"""

Returns a minibatch of self.batch\_size = 32 transitions

"""

if self.count < self.agent\_history\_length:

raise ValueError('Not enough memories to get a minibatch')

self.\_get\_valid\_indices()

for i, idx in enumerate(self.indices):

self.states[i] = self.\_get\_state(idx - 1)

self.new\_states[i] = self.\_get\_state(idx)

return np.transpose(self.states, axes=(0, 2, 3, 1)), self.actions[self.indices], self.rewards[self.indices], np.transpose(self.new\_states, axes=(0, 2, 3, 1)), self.terminal\_flags[self.indices]

## 6. Target network and parameter update

Why do we need two networks, the action-value function and the target action-value function?

Remember that prior to updating the network's parameters, we draw a minibatch with 32 transitions. For simplicity we consider one transition now. It consists of a state, an action that was performed in the state, the received reward, the new\_state and a bool saying whether the episode is over.

We perform a gradient descent step: The main network looks at state and estimates the 𝑄prediction

-values that say how good each action is. However, we want the 𝑄-values to follow the Bellman equation we introduced above. Therefore we calculate the 𝑄target-values according to the Bellman equation (how we would like the 𝑄-values to be) and then compare the estimates 𝑄prediction to the targets 𝑄target

. Let's consider the quadratic loss function instead of the Huber loss function for simplicity:

𝐿=12(𝑄prediction−𝑄target)2

This ensures that we regress the current 𝑄prediction

-values for state towards the 𝑄target

-values given by the Bellman equation.

𝑄prediction

is calculated in the DQN class (self.q\_values). 𝑄prediction depends on the current state in the minibatch we drew and on the parameters 𝜃

of the network that estimates it.

The 𝑄target

value is calculated according to the Bellman equation. It is the sum of the immediate reward 𝑟 received for performing action 𝑎 in state 𝑠 (action and state from the minibatch) and the maximum 𝑄-value over all possible actions 𝑎′ in 𝑠′

(new\_state from the minibatch):

𝑄target(𝑠,𝑎)=𝑟+𝛾max(𝑄(𝑠′,𝑎′))

This is not done in the DQN class but in the learn method below. The calculated value is then passed to the placeholder called self.target\_q in the DQN class. There, the loss function is defined and the gradient descent step is performed.

So, now that we understand how the parameters are updated, why use two networks?

The problem is that both 𝑄prediction

and 𝑄target depend on the same parameters 𝜃 if only one network is used. This can lead to instability when regressing 𝑄prediction towards 𝑄target because the "target is moving". We ensure a "fixed target" by introducing a second network with fixed and only occasionally updated parameters that estimates the target 𝑄

-values.

On page 1 of [Mnih et al. 2015](https://www.nature.com/articles/nature14236/) the authors explain:

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q) function. This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values [...] and the target values [...]. We address these instabilities with a novel variant of Q-learning, which uses two key ideas. First, we used a biologically inspired mechanism termed experience replay that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution [...]. Second, we used an iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target.

Therefore they used one network to predict the 𝑄prediction

-value and the other fixed network to predict the 𝑄target

-value. The main network is optimized during the gradient descend step and every 10000 steps the main network's parameters are copied to the target network. Be aware that the network update frequency is measured in the number of chosen actions/frames seen (DeepMind code) and not the number of parameter updates which occur every four frames ([Mnih et al. 2015](https://www.nature.com/articles/nature14236/)).

There is one additional very powerful improvement called Double Q-Learning.

## 7. Double Q-Learning

DQN has been observed to estimate unrealistically high 𝑄

-values. The reason for this is, that the Bellman equation includes a maximization step over estimated action values, which tends to prefer overestimated to underestimated values (see [van Hasselt et al. 2016, page 1](http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/download/12389/11847)).

The authors explain:

If all values would be uniformly higher then the relative action preferences are preserved and we would not expect the resulting policy to be any worse. [...] If, however, the overestimations are not uniform and not concentrated at states about which we wish to learn more, then they might negatively affect the quality of the resulting policy. [...] We then show that this algorithm not only yields more accurate value estimates, but leads to much higher scores on several games. This demonstrates that the overestimations of DQN were indeed leading to poorer policies and that it is beneficial to reduce them

The estimated 𝑄

-values are noisy. Assume that the true 𝑄-value is 0 for all actions. But because of the noisy estimation, some 𝑄-values might be slightly positive, others slightly negative. The max operation in the Bellman equation will however always chose the small positive values, despite the fact, that those actions are not truly better. The estimatation of 𝑄-values is thus biased towards larger values. How do we fix this? Instead of estimating the 𝑄-values in the next state (𝑠′,𝑎′) with only the target network, we use the main network to estimate which action is the best and then ask the target network how high the 𝑄-value is for that action. This way, the main network will still prefer the action with the small positive 𝑄-value but because of the noisy estimation, the target network will predict a small positive or small negative 𝑄-value for that action and on average, the predicted 𝑄

-values will be closer to 0.

Mathematically, the reason for the overestimation is, that the expectation of a maximum is greater than or equal to the maximum of an expectation [van Hasselt 2013, Theorem 1](https://arxiv.org/abs/1302.7175).

The Bellman equation changes from

𝑄target(𝑠,𝑎)to𝑄target(𝑠,𝑎)=𝑟+𝛾max𝑄(𝑠′,𝑎′;𝜃target)=𝑟+𝛾𝑄(𝑠′,𝑎′=argmax𝑄(𝑠′,𝑎′;𝜃main);𝜃target)Normal DQNDouble DQN

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

.

Double DQN: Ask the main network which action has 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 main network will answer that action with index 0 has the highest 𝑄-value. Then we ask the target network, which has a different noise, what the 𝑄-value for the action with the chosen index (0 in this example) is. Let's assume the target network's noisy estimates are (−0.05,0.3) it will answer −0.05

.

This solves the problem of overestimated 𝑄

-values because the two networks have different noise and the bias towards slightly larger noisy 𝑄

-values cancels.

One more thing: If the game is over (terminal=True) because the agend lost or won, there is no next state and the 𝑄target

-value is simply the reward 𝑟

.

Look at the implementation in the cell below.

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.

"""

# Draw a minibatch from the replay memory

states, actions, rewards, new\_states, terminal\_flags = replay\_memory.get\_minibatch()

# The main network estimates which action is best (in the next

# state s', new\_states is passed!)

# for every transition in the minibatch

arg\_q\_max = session.run(main\_dqn.best\_action, feed\_dict={main\_dqn.input:new\_states})

# The target network estimates the Q-values (in the next state s', new\_states is passed!)

# for every transition in the minibatch

q\_vals = session.run(target\_dqn.q\_values, feed\_dict={target\_dqn.input:new\_states})

double\_q = q\_vals[range(batch\_size), arg\_q\_max]

# Bellman equation. Multiplication with (1-terminal\_flags) makes sure that

# if the game is over, targetQ=rewards

target\_q = rewards + (gamma\*double\_q \* (1-terminal\_flags))

# Gradient descend step to update the parameters of the main network

loss, \_ = session.run([main\_dqn.loss, main\_dqn.update],

feed\_dict={main\_dqn.input:states,

main\_dqn.target\_q:target\_q,

main\_dqn.action:actions})

return loss

As I already mentioned, the parameters of the main network are periodically copied every 10,000 steps to the target network. This is implemented in the cell below.

class TargetNetworkUpdater(object):

"""Copies the parameters of the main DQN to the target DQN"""

def \_\_init\_\_(self, main\_dqn\_vars, target\_dqn\_vars):

"""

Args:

main\_dqn\_vars: A list of tensorflow variables belonging to the main DQN network

target\_dqn\_vars: A list of tensorflow variables belonging to the target DQN network

"""

self.main\_dqn\_vars = main\_dqn\_vars

self.target\_dqn\_vars = target\_dqn\_vars

​

def \_update\_target\_vars(self):

update\_ops = []

for i, var in enumerate(self.main\_dqn\_vars):

copy\_op = self.target\_dqn\_vars[i].assign(var.value())

update\_ops.append(copy\_op)

return update\_ops

def \_\_call\_\_(self, sess):

"""

Args:

sess: A Tensorflow session object

Assigns the values of the parameters of the main network to the

parameters of the target network

"""

update\_ops = self.\_update\_target\_vars()

for copy\_op in update\_ops:

sess.run(copy\_op)

The function in the cell below creates a gif from a sequence of frames passed to it.

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

"""

Args:

frame\_number: Integer, determining the number of the current frame

frames\_for\_gif: A sequence of (210, 160, 3) frames of an Atari game in RGB

reward: Integer, Total reward of the episode that es ouputted as a gif

path: String, path where gif is saved

"""

for idx, frame\_idx in enumerate(frames\_for\_gif):

frames\_for\_gif[idx] = resize(frame\_idx, (420, 320, 3),

preserve\_range=True, order=0).astype(np.uint8)

imageio.mimsave(f'{path}{"ATARI\_frame\_{0}\_reward\_{1}.gif".format(frame\_number, reward)}',

frames\_for\_gif, duration=1/30)

​

The learning environment is provided by OpenAi's gym. It is very important that you have the right version of the environments. BreakoutDeterministic-v3 for example has six actions whereas BreakoutDeterministic-v4 has a minimal set of four actions, which is what DeepMind used in [xitari](https://github.com/deepmind/xitari/blob/master/games/supported/Breakout.cpp#_blank). Additional actions make the learning task harder for the agent which can alter the evaluation score significantly. If you want to find out the number of actions and their meaning, type env.action\_space.n and env.unwrapped.get\_action\_meanings().

There are two additional small adjustments we need to discuss:

When a life is lost, we save terminal\_life\_lost = True in the replay memory. Create a new notebook, make a Breakout environment, in a loop repeat random or no actions and print the reward and the number of lives the agent has.

frame = env.reset() for i in range(1000): new\_frame, reward, terminal, info = env.step(0) print(reward, terminal, info['ale.lives'])

You will see, that there is no punishment (reward is 0) when a life is lost. It helps the agent tremendously avoiding losing a life if you consider loss of life as end of episode. However, we only do this in the replay memory as we do not want to reset the game once the first life is lost. Therefore two terminal states terminal and terminal\_life\_lost are needed, one to reset the game, the other for the replay memory. This adjustment helped the agent improve from an average reward slightly above 50 to approximately 140 in Breakout!

Let's wrap the gym environment in an Atari class which takes care of stacking frames ontop of each other to create states, resetting the environment when an episode ended and checking if a life was lost after a step was taken. You find the implementation in the cell below.

During evaluation, at the beginning of each episode, action 1 ('FIRE') is repeated for a random number of steps between 1 and no\_op\_steps=10. This ensures, that the agent starts in a different situation every time and thus cannot simply learn a fixed sequence of actions. [Mnih et al. 2015](https://www.nature.com/articles/nature14236/) use a random number between 1 and 30 of 'NOOP'-actions (see page 10, Table 1). However, in Breakout, nothing happens if you don't fire first. Once there is a ball in the game, 'FIRE' does nothing. Therefore I started with a random number of 'FIRE'-actions. Furthermore, I limited the random number of initial 'FIRE' actions to 10. When experimenting with larger numbers, I found, that the first life was usually already lost when the agent starting moving. You might want to change this, in case you want to experiment with another environment.

class Atari(object):

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

def \_\_init\_\_(self, envName, no\_op\_steps=10, agent\_history\_length=4):

self.env = gym.make(envName)

self.process\_frame = FrameProcessor()

self.state = None

self.last\_lives = 0

self.no\_op\_steps = no\_op\_steps

self.agent\_history\_length = agent\_history\_length

​

def reset(self, sess, evaluation=False):

"""

Args:

sess: A Tensorflow session object

evaluation: A boolean saying whether the agent is evaluating or training

Resets the environment and stacks four frames ontop of each other to

create the first state

"""

frame = self.env.reset()

self.last\_lives = 0

terminal\_life\_lost = True # Set to true so that the agent starts

# with a 'FIRE' action when evaluating

if evaluation:

for \_ in range(random.randint(1, self.no\_op\_steps)):

frame, \_, \_, \_ = self.env.step(1) # Action 'Fire'

processed\_frame = self.process\_frame(sess, frame) # (★★★)

self.state = np.repeat(processed\_frame, self.agent\_history\_length, axis=2)

return terminal\_life\_lost

​

def step(self, sess, action):

"""

Args:

sess: A Tensorflow session object

action: Integer, action the agent performs

Performs an action and observes the reward and terminal state from the environment

"""

new\_frame, reward, terminal, info = self.env.step(action) # (5★)

if info['ale.lives'] < self.last\_lives:

terminal\_life\_lost = True

else:

terminal\_life\_lost = terminal

self.last\_lives = info['ale.lives']

processed\_new\_frame = self.process\_frame(sess, new\_frame) # (6★)

new\_state = np.append(self.state[:, :, 1:], processed\_new\_frame, axis=2) # (6★)

self.state = new\_state

return processed\_new\_frame, reward, terminal, terminal\_life\_lost, new\_frame

Mnih et al. 2013 and Mnih et al. 2015 clipped the rewards explaining that "Since the scale of scores varies greatly from game to game, we fixed all positive rewards to be 1 and all negative rewards to be -1, leaving 0 rewards unchanged. Clipping the rewards in this manner limits the scale of the error derivatives and makes it easier to use the same learning rate across multiple games. At the same time, it could affect the performance of our agent since it cannot differentiate between rewards of different magnitude." Clipping the rewards makes the training converge faster and improves the performance in the example of Breakout.

def clip\_reward(reward):

if reward > 0:

return 1

elif reward == 0:

return 0

else:

return -1

In the cell below I declare some constants that define the learning behaviour of the agent:

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

Setting up tensorboard summaries for the loss, the average reward, the evaluation score and the network parameters to observe the learning process:

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. Putting everything together: DQN

Now, we have everything we need to finally implement Algorithm 1 from [Mnih et al. 2015](https://www.nature.com/articles/nature14236/). The stars will help you identify where each step is implemented.

Go through the implementation in the cell below. You will find, that following every learning epoch of 200,000 frames (50,000 parameter updates), the agent is evaluated during 10,000 frames. Summaries are written to the logdir for tensorboard.

I described the experiments I ran using this notebook [here](https://medium.com/@fabiograetz/tutorial-double-deep-q-learning-with-dueling-network-architectures-4c1b3fb7f756). Feel free to play with the notebook, implement changes and experiment with different environments. If you are able to improve it or gained experience with another Atari game, please leave a commet [here](https://medium.com/@fabiograetz/tutorial-double-deep-q-learning-with-dueling-network-architectures-4c1b3fb7f756), I'd love to know about it.

def train():

"""Contains the training and evaluation loops"""

my\_replay\_memory = ReplayMemory(size=MEMORY\_SIZE, batch\_size=BS) # (★)

update\_networks = TargetNetworkUpdater(MAIN\_DQN\_VARS, TARGET\_DQN\_VARS)

explore\_exploit\_sched = ExplorationExploitationScheduler(

MAIN\_DQN, atari.env.action\_space.n,

replay\_memory\_start\_size=REPLAY\_MEMORY\_START\_SIZE,

max\_frames=MAX\_FRAMES)

​

with tf.Session() as sess:

sess.run(init)

frame\_number = 0

rewards = []

loss\_list = []

while frame\_number < MAX\_FRAMES:

########################

####### Training #######

########################

epoch\_frame = 0

while epoch\_frame < EVAL\_FREQUENCY:

terminal\_life\_lost = atari.reset(sess)

episode\_reward\_sum = 0

for \_ in range(MAX\_EPISODE\_LENGTH):

# (4★)

action = explore\_exploit\_sched.get\_action(sess, frame\_number, atari.state)

# (5★)

processed\_new\_frame, reward, terminal, terminal\_life\_lost, \_ = atari.step(sess, action)

frame\_number += 1

epoch\_frame += 1

episode\_reward\_sum += reward

# Clip the reward

clipped\_reward = clip\_reward(reward)

# (7★) Store transition in the replay memory

my\_replay\_memory.add\_experience(action=action,

frame=processed\_new\_frame[:, :, 0],

reward=clipped\_reward,

terminal=terminal\_life\_lost)

if frame\_number % UPDATE\_FREQ == 0 and frame\_number > REPLAY\_MEMORY\_START\_SIZE:

loss = learn(sess, my\_replay\_memory, MAIN\_DQN, TARGET\_DQN,

BS, gamma = DISCOUNT\_FACTOR) # (8★)

loss\_list.append(loss)

if frame\_number % NETW\_UPDATE\_FREQ == 0 and frame\_number > REPLAY\_MEMORY\_START\_SIZE:

update\_networks(sess) # (9★)

if terminal:

terminal = False

break

​

rewards.append(episode\_reward\_sum)

# Output the progress:

if len(rewards) % 10 == 0:

# Scalar summaries for tensorboard

if frame\_number > REPLAY\_MEMORY\_START\_SIZE:

summ = sess.run(PERFORMANCE\_SUMMARIES,

feed\_dict={LOSS\_PH:np.mean(loss\_list),

REWARD\_PH:np.mean(rewards[-100:])})

SUMM\_WRITER.add\_summary(summ, frame\_number)

loss\_list = []

# Histogramm summaries for tensorboard

summ\_param = sess.run(PARAM\_SUMMARIES)

SUMM\_WRITER.add\_summary(summ\_param, frame\_number)

print(len(rewards), frame\_number, np.mean(rewards[-100:]))

with open('rewards.dat', 'a') as reward\_file:

print(len(rewards), frame\_number,

np.mean(rewards[-100:]), file=reward\_file)

########################

###### Evaluation ######

########################

terminal = True

gif = True

frames\_for\_gif = []

eval\_rewards = []

evaluate\_frame\_number = 0

for \_ in range(EVAL\_STEPS):

if terminal:

terminal\_life\_lost = atari.reset(sess, evaluation=True)

episode\_reward\_sum = 0

terminal = False

# Fire (action 1), when a life was lost or the game just started,

# so that the agent does not stand around doing nothing. When playing

# with other environments, you might want to change this...

action = 1 if terminal\_life\_lost else explore\_exploit\_sched.get\_action(sess, frame\_number,

atari.state,

evaluation=True)

processed\_new\_frame, reward, terminal, terminal\_life\_lost, new\_frame = atari.step(sess, action)

evaluate\_frame\_number += 1

episode\_reward\_sum += reward

​

if gif:

frames\_for\_gif.append(new\_frame)

if terminal:

eval\_rewards.append(episode\_reward\_sum)

gif = False # Save only the first game of the evaluation as a gif

print("Evaluation score:\n", np.mean(eval\_rewards))

try:

generate\_gif(frame\_number, frames\_for\_gif, eval\_rewards[0], PATH)

except IndexError:

print("No evaluation game finished")

#Save the network parameters

saver.save(sess, PATH+'/my\_model', global\_step=frame\_number)

frames\_for\_gif = []

# Show the evaluation score in tensorboard

summ = sess.run(EVAL\_SCORE\_SUMMARY, feed\_dict={EVAL\_SCORE\_PH:np.mean(eval\_rewards)})

SUMM\_WRITER.add\_summary(summ, frame\_number)

with open('rewardsEval.dat', 'a') as eval\_reward\_file:

print(frame\_number, np.mean(eval\_rewards), file=eval\_reward\_file)

## 10. Train it yourself

If you want to train the network yourself, set TRAIN to True in the first cell of this notebook.

Consider making your computer accessible remotely which I described in this blog post: [Accessing your Deep Learning station remotely and setting up wake on lan](https://medium.com/@fabiograetz/accessing-your-deep-learning-station-remotely-and-setting-up-wake-on-lan-1e708c50fdd8)

You can convert the notebook to a python script using jupyter-nbconvert --to script DQN.ipynb and then run it in a tmux session which I described here: [jupyter and tensorboard in tmux](https://medium.com/@fabiograetz/jupyter-and-tensorboard-in-tmux-5e5d202a4fb6). This has the advantage that you can detach from the tmux session and reattach to it remotely (from a laptop for example) in order to check on the agents progress.

If you want to use tensorboard to monitor the networks improvements type tensorboard --logdir=summaries in a terminal in which the respective virtual environment is activated. Open a browser and go to [http://localhost:6006](http://localhost:6006/). This works remotely as well.

if TRAIN:

train()

## 9. Use a trained network

save\_files\_dict = {

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

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

}

if not TRAIN:

gif\_path = "GIF/"

os.makedirs(gif\_path, exist\_ok=True)

​

trained\_path, save\_file = save\_files\_dict[ENV\_NAME]

​

explore\_exploit\_sched = ExplorationExploitationScheduler(

MAIN\_DQN, atari.env.action\_space.n,

replay\_memory\_start\_size=REPLAY\_MEMORY\_START\_SIZE,

max\_frames=MAX\_FRAMES)

with tf.Session() as sess:

saver = tf.train.import\_meta\_graph(trained\_path+save\_file)

saver.restore(sess,tf.train.latest\_checkpoint(trained\_path))

frames\_for\_gif = []

terminal\_life\_lost = atari.reset(sess, evaluation = True)

episode\_reward\_sum = 0

while True:

atari.env.render()

action = 1 if terminal\_life\_lost else explore\_exploit\_sched.get\_action(sess, 0, atari.state,

evaluation = True)

processed\_new\_frame, reward, terminal, terminal\_life\_lost, new\_frame = atari.step(sess, action)

episode\_reward\_sum += reward

frames\_for\_gif.append(new\_frame)

if terminal == True:

break

atari.env.close()

print("The total reward is {}".format(episode\_reward\_sum))

print("Creating gif...")

generate\_gif(0, frames\_for\_gif, episode\_reward\_sum, gif\_path)

print("Gif created, check the folder {}".format(gif\_path))