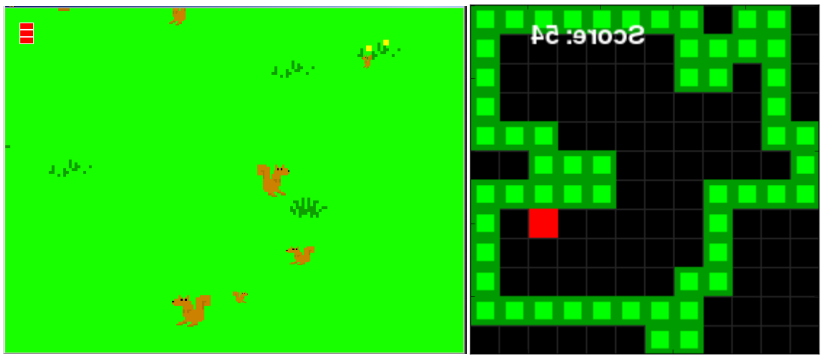
Technion
Israel Institute of Technology

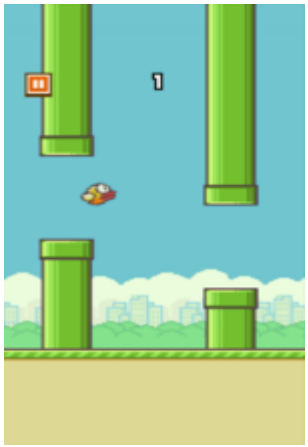
Using Deep Q-Network To Learn Fast How To Play Games

Submitters: **Roee Mazor**

Ilana Feldman

Supervisor: **Roman Kaplan**





Semester: **Spring 2016**

Date of Submission: **March 2017**

Table of Contents

Table of Figures

Introduction

1. Abstract

   a. Project Definition

      This project demonstrates how to use the Deep-Q Learning algorithm with Keras together to play several games.

   b. Introduction

      Imagine playing a new game whose rules you don't know; after a hundred or so moves, your opponent announces, "You lose"[1]

      Learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long-standing challenges of reinforcement learning (RL). Most successful RL applications that operate on these domains have relied on hand-crafted features combined with linear value functions or policy representations. Clearly, the performance of such systems heavily relies on the quality of the feature representation.
      Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in computer vision and speech recognition.
      These methods utilize a range of neural network architectures, including convolutional networks, multilayer perceptron's, restricted Boltzmann machines and recurrent neural networks, and have exploited both supervised and unsupervised learning.
      It seems natural to ask whether similar techniques could also be beneficial for RL with sensory data.
      However, reinforcement learning presents several challenges from a deep learning perspective.
      Firstly, most successful deep learning applications to date have required large amounts of hand labelled training data. RL algorithms, on the other hand, must be able to learn from a scalar reward signal that is frequently sparse, noisy and delayed. The delay between actions and resulting rewards, which can be thousands of time steps long, seems particularly daunting when compared to the direct association between inputs and targets found in supervised learning.
      Another issue is that most deep learning algorithms assume the data samples to be independent, while in reinforcement learning one typically encounters sequences of highly correlated states. Furthermore, in RL the data distribution changes as the algorithm learns new behaviors, which can be problematic for deep learning methods that assume a fixed underlying distribution.
      This paper demonstrates that a convolutional neural network can overcome these challenges to learn successful control policies from raw video data in complex RL

[1] [1] Russell and Norvig - Introduction to Artificial Intelligence

environments. The network is trained with a variant of the Q-learning algorithm, with stochastic gradient descent to update the weights.
We apply our approach to a range of games implemented in python with a high dimensional visual input and a diverse and interesting set of tasks that were designed to be difficult for humans players. Our goal is to create a single neural network agent that is able to successfully learn to play as many of the games as possible. The network was not provided
with any game-specific information or hand-designed visual features, and was not privy to the internal state of the emulator; it learned from nothing but the image input, the reward and terminal signals, and the set of possible actions—just as a human player would.
Furthermore the network architecture and all hyperparameters used for training were kept constant across the games.

This project is the result of an Academic Course, at the NSSL Lab, Technion, Israel.

c. Development Environments:

i. Python 2.7

ii. Keras

Keras ($\kappa \acute{\varepsilon} \rho \ \alpha \varsigma$) means horn in greek. It is a reference to a literary image from ancient Greek and Latin literature:
Two divided dream spirits;

- Ivory, those who deceive men with false visions
- Horn, those who announce a future that will come to pass

Keras is written in python and capable of running on top of either TensorFlow or Theano.

Keras guiding principles are:

- Modularity
  A model is understood as a sequence or a graph of standalone, fully-configurable modules that can be plugged together with as little restrictions as possible. In particular, neural layers, cost functions, optimizers, initialization schemes, activation functions, regularization schemes are all standalone modules that you can combine to create new models.

- Minimalism
  Each module should be kept short and simple. Every piece of code should be transparent upon first reading. No black magic: it hurts iteration speed and ability to innovate.

- Easy extensibility
  New modules are dead simple to add (as new classes and functions), and existing modules provide ample examples. To be able to easily create new

modules allows for total expressiveness, making Keras suitable for advanced research.

- Work with Python.
  No separate models configuration files in a declarative format. Models are described in Python code, which is compact, easier to debug, and allows for ease of extensibility.

iii. Theano

Theano is named after the Greek mathematician, who may have been Pythagoras' wife.

Theano was written at the LISA lab to support rapid development of efficient machine learning algorithms.

Theano is a Python library that lets you to define, optimize, and evaluate mathematical expressions, especially ones with multi-dimensional arrays (numpy.ndarray).

Using Theano it is possible to attain speeds rivaling hand-crafted C implementations for problems involving large amounts of data. It can also surpass C on a CPU by many orders of magnitude by taking advantage of recent GPUs.

Theano combines aspects of a computer algebra system (CAS) with aspects of an optimizing compiler. It can also generate customized C code for many mathematical operations. This combination of CAS with optimizing compilation is particularly useful for tasks in which complicated mathematical expressions are evaluated repeatedly and evaluation speed is critical. For situations where many different expressions are each evaluated once Theano can minimize the amount of compilation/analysis overhead, but still provide symbolic features such as automatic differentiation.

Theano's compiler applies many optimizations of varying complexity to these symbolic expressions. These optimizations include, but are not limited to:

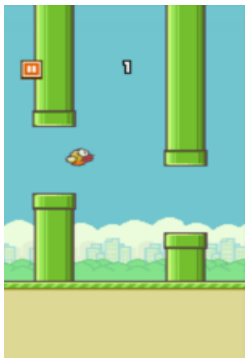
- use of GPU for computations
- constant folding
- merging of similar subgraphs, to avoid redundant calculation
- arithmetic simplification (e.g. x*y/x -> y, --x -> x)
- inserting efficient BLAS operations (e.g. GEMM) in a variety of contexts
- using memory aliasing to avoid calculation
- using inplace operations wherever it does not interfere with aliasing
- loop fusion for elementwise sub-expressions
- improvements to numerical stability (e.g. $\log(1+\exp(x))$ and $\log(\sum_i \exp(x[i]))$)

2. The games

a. Flappy bird

Flappy Bird is a 2013 mobile game developed by Vietnamese artist and programmer Dong Nguyen, under his game development company dotGEARS. The game is a side-scroller where the player controls a bird, attempting to fly between rows of green pipes without hitting them

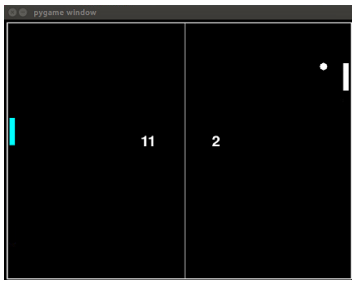
Figure 1 - Flappy Bird



b. Pong

Pong is one of the earliest arcade video games and the very first sports arcade video game. While other arcade video games such as Computer Space came before it, Pong was one of the first video games to reach mainstream popularity. The game was originally manufactured by Atari, which released it in 1972.
Pong is a two-dimensional sports game that simulates table tennis. The player controls an in-game paddle by moving it vertically across the left side of the screen, and can compete against either a computer-controlled opponent or another player controlling a second paddle on the opposing side. Players use the paddles to hit a ball back and forth. The aim is for each player to reach eleven points before the opponent; points are earned when one fails to return the ball to the other.

Figure 2 - Pong

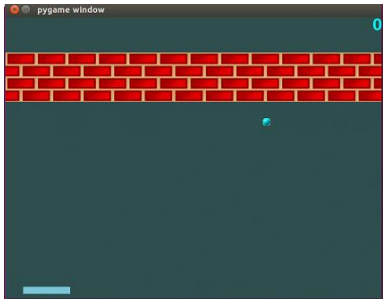


c. Breakout

Breakout is an arcade game developed and published by Atari, Inc. influenced by the 1972 Atari arcade game Pong.
The game was ported to multiple platforms and upgraded to video games such as Super Breakout. In addition, Breakout was the basis and inspiration for certain aspects of the Apple II personal computer.

In the game, a layer of bricks lines the top third of the screen. A ball travels across the screen, bouncing off the top and sidewalls of the screen. When a brick is hit, the ball bounces away and the brick is destroyed. The player loses a turn when the ball touches the bottom of the screen. To prevent this from happening, the player has a movable paddle to bounce the ball upward, keeping it in play.
Fan fact about breakout: it was built by Steve Wozniak aided by Steve Jobs.

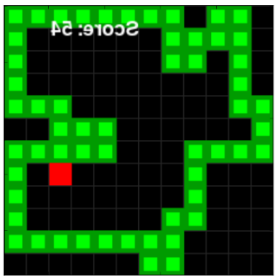
Figure 3 - Breakout



d. Snake

Snake is the common name for a videogame concept where the player maneuvers a line which grows in length, with the line itself being a primary obstacle. The concept originated in the 1976 arcade game Blockade, and the ease of implementing Snake has led to hundreds of versions (some of which have the word snake or worm in the title) for many platforms. There are over 300 for iOS alone.

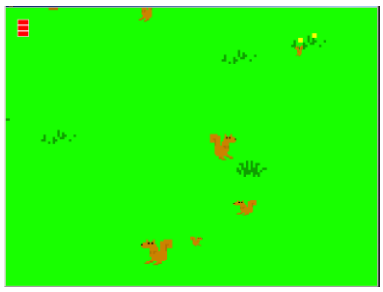
Figure 4 - Snake



e. Squirrel eat Squirrel

Squirrel Eat Squirrel is a 2D Katamari clone. You start as a small squirrel that must chase down and eat smaller squirrels, while avoiding all the squirrels that are bigger than you. The more squirrels you eat, the larger you become.

Figure 5 - Squirrel eat Squirrel

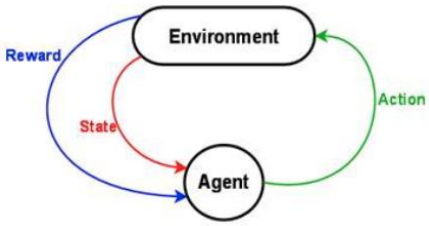


3. Background

Suppose you want to teach a neural network to play a game. Input to your network would be screen images, and output would the actions. It would make sense to treat it as a classification problem – for each game screen you have to decide which action to take. We need occasional feedback that we did the right thing and can then figure out everything else ourselves.

This is the task reinforcement learning tries to solve. Reinforcement learning lies somewhere in between supervised and unsupervised learning. Whereas in supervised learning one has a target label for each training example and in unsupervised learning one has no labels at all, in reinforcement learning one has sparse and time-delayed labels – the rewards. Based only on those rewards the agent has to learn to behave in the environment.
While the idea is quite intuitive, in practice there are numerous challenges. For example when you hit a brick and score a reward in the Breakout game, it often has nothing to do with the actions (paddle movements) you did just before getting the reward. All the hard work was already done, when you positioned the paddle correctly and bounced the ball back. This is called the credit assignment problem – i.e., which of the preceding actions was responsible for getting the reward and to what extent.
Reinforcement learning is an important model of how we (and all animals in general) learn. Praise from our parents, grades in school, salary at work – these are all examples of rewards. Credit assignment problems come up every day both in business and in relationships. That is why it is important to study this problem, and games form a wonderful sandbox for trying out new approaches.

Figure 6 - : Reinforcement learning problem

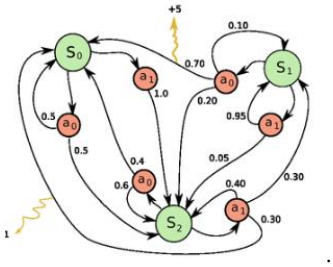


a. Markov Decision Process

The question is, how do you formalize a reinforcement-learning problem, so that you can reason about it? The most common method is to represent it as a Markov decision process.
Suppose you are an agent, situated in an environment (the game). The environment is in a certain state (e.g. location of the paddle, location and direction of the ball, existence of every brick and so on). The agent can perform certain actions in the environment (e.g. move the paddle to the left or to the right). These actions sometimes result in a reward (e.g. increase in score). Actions transform the environment and lead to a new state, where the agent can perform another action, and so on. The rules for how you choose those actions are called policy. The environment in general is stochastic, which means the next state may be

somewhat random (e.g. when you lose a ball and launch a new one, it goes towards a random direction).

Figure 7 - : Markov decision process



The set of states and actions, together with rules for transitioning from one state to another, make up a Markov decision process. One episode of this process (e.g. one game) forms a finite sequence of states, actions and rewards: $s_0, a_0, r_1, s_1, a_1, r_2, ..., s_{n-1}, a_{n-1}, r_n, s_n$

Here s_i represents the state, a_i is the action and r_{i+1} s the reward after performing the action. The episode ends with terminal state s_n (e.g. “game over” screen). A Markov decision process relies on the Markov assumption, that the probability of the next state s_{i+1} depends only on current state s_i and action a_i, but not on preceding states or actions.

b. Discounted Future Reward

To perform well in the long-term, we need to take into account not only the immediate rewards, but also the future rewards we are going to get. How should we go about that?

Given one run of the Markov decision process, we can easily calculate the total reward for one episode:

$$R_t = r_1 + r_2 + ... + r_n$$

Given that, the total future reward from time point *t* o reward can be expressed as:

$$R_t = r_t + r_{t+1} + ... + r_n$$

However, because our environment is stochastic, we can never be sure, if we will get the same rewards the next time we perform the same actions. The more into the future we go, the more it may diverge. For that reason, it is common to use discounted future reward instead:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \cdots + \gamma^{n-t} r_n$$

Here γ is the discount factor between 0 and 1 – the more into the future the reward is, the less we take it into consideration. It is easy to see, that discounted future reward at time step *t* can be expressed in terms of the same thing at time step *t+1*:

$$R_t = r_t + \gamma\left(r_{t+1} + \gamma\left(r_{t+2} + ...\right)\right) = r_t + \gamma R_{t+1}$$

If we set the discount factor $\gamma = 0$ then our strategy will be short-sighted and we rely only on the immediate rewards. If we want to balance between immediate and future rewards, we should set discount factor to something like γ=0.9. If our environment is deterministic and the same actions always result in same rewards, then we can set discount factor γ=1.
A good strategy for an agent would be to always choose an action that maximizes the (discounted) future reward.

c. Q-learning - Bellman equation

In Q-learning we define a function $Q(s,a)$ representing the maximum discounted future reward when we perform action a in state s, and continue optimally from that point on.

$$Q(s_t, a_t) = \max R_{t+1}$$

The way to think about $Q(s,a)$ is that it is "the best possible score at the end of the game after performing action a in state s". It is called Q-function, because it represents the "quality" of a certain action in a given state.
This may sound like quite a puzzling definition. How can we estimate the score at the end of game, if we know just the current state and action, and not the actions and rewards coming after that? We really can't. But as a theoretical construct we can assume existence of such a function.
The meaning of such function is if you are in state and wondering whether you should take action *a* or *b*. You want to select the action that results in the highest score at the end of game. Once you have the magical Q-function, the answer becomes really simple – pick the action with the highest Q-value!

$$\pi(s) = \arg\max_a Q(s,a)$$

Here π represents the policy, the rule how we choose an action in each state.
We can express the Q-value of state *s* and action *a* in terms of the Q-value of the next state *s'*.

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

This is called the Bellman equation – maximum future reward for this state and action is the immediate reward plus maximum future reward for the next state.
The main idea in Q-learning is that we can iteratively approximate the Q-function using the Bellman equation. In the simplest case the Q-function is implemented as a table, with states as rows and actions as columns. The Q-learning algorithm is as simple as the following:

Figure 8 - Q-learn algorithm

```
initialize Q[num_states,num_actions] arbitrarily
observe initial state s
repeat
      select and carry out an action a
      observe reward r and new state s'
      Q[s,a] = Q[s,a] + α(r + γ max_a' Q[s',a'] - Q[s,a])
      s = s'
until terminated
```

α in the algorithm is the learning rate that controls how much of the difference between previous Q-value and newly proposed Q-value is taken into account. In particular, when $\alpha = 1$, then the two $Q(s,a)$ cancel each other and the update is the same as the Bellman equation.

The $\max_{a'} Q(s',a')$ that we use to update $Q(s,a)$ is only an approximation and in early stages of learning it may be completely wrong. However, the approximation get more and more accurate with every iteration and if we perform this update enough times, then the Q-function will converge and represent the true Q-value.

d. Deep Q Network

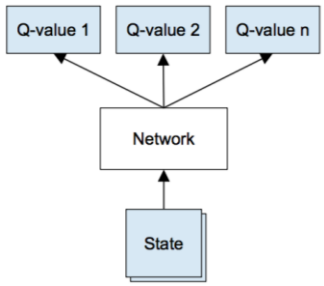
We use screen pixels as definition of current state – they implicitly contain all of the relevant information about the game situation, except for the speed and direction, so we use four consecutive screens for this.

If we apply the same preprocessing to game screens as in the DeepMind paper – take the four last screen images, resize them to $84 \cdot 84$ and convert to grayscale with 256 gray levels – we would have $256^{\underset{grey\_levels}{84 \cdot 84 \cdot 4}}_{image\_size\ images} \approx 10^{67970}$ possible game states.

This means 10^{67970} rows in our Q-table. One could argue that many pixel combinations (and therefore states) never occur – we could possibly represent it as a sparse table containing only visited states. Even so, most of the states are rarely visited and it would take a lifetime for the Q-table to converge. Ideally, we would also like to have a good guess for Q-values for states we have never seen before.

This is the point where deep learning steps in. Neural networks are exceptionally good at coming up with good features for highly structured data. We could represent our Q-function with a neural network, that takes the state (four game screens) and action as input and outputs the corresponding Q-value. Alternatively, we could take only game screens as input and output the Q-value for each possible action. This approach has the advantage that if we want to perform a Q-value update or pick the action with the highest Q-value, we only have to do one forward pass through the network and have all Q-values for all actions available immediately.

Figure 9 - Architecture of deep Q-network, used in DeepMind paper.



The network architecture that DeepMind used is as follows:

Figure 10 - DeepMind network architecture

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

This is a classical convolutional neural network with three convolutional layers, followed by two fully connected layers. Input to the network are 4 $84 \cdot 84$ grayscale game screens. Outputs of the network are Q-values for each possible action (18 in Atari). Q-values can be any real values, which makes it a regression task that can be optimized with simple squared error loss.

$$L = \frac{1}{2}\left(\underbrace{r + \max_{a'} Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}\right)^2$$

Given a transition $\langle s, a, r, s' \rangle$, the Q-table update rule in the previous algorithm must be replaced with the following:

1. Do a feedforward pass for the current state *s* to get predicted Q-values for all actions.
2. Do a feedforward pass for the next state *s'* and calculate maximum overall network outputs $\max a' Q(s', a')$.
3. Set Q-value target for action to $r + \gamma m \max_{a'} Q(s', a')$ (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
4. Update the weights using backpropagation

.

e. Experience Replay

By now, we have an idea how to estimate the future reward in each state using Q-learning and approximate the Q-function using a convolutional neural network. However, it turns out that approximation of Q-values using non-linear functions is

not very stable. There is a whole bag of tricks that you have to use to actually make it converge. Moreover, it takes a long time, almost a week on a single GPU.
The most important trick is experience replay. During gameplay all the experiences $\langle s, a, r, s' \rangle$ are stored in a replay memory. When training the network, random minibatches from the replay memory are used instead of the most recent transition. This breaks the similarity of subsequent training samples, which otherwise might drive the network into a local minimum. In addition, experience replay makes the training task more similar to usual supervised learning, which simplifies debugging and testing the algorithm. One could actually collect all those experiences from human gameplay and then train network on these.

f. Exploration-Exploitation

Q-learning attempts to solve the credit assignment problem – it propagates rewards back in time, until it reaches the crucial decision point which was the actual cause for the obtained reward. Firstly observe, that when a Q-table or Q-network is initialized randomly, then its predictions are initially random as well. If we pick an action with the highest Q-value, the action will be random and the agent performs crude "exploration". As a Q-function converges, it returns Q-values that are more consistent and the amount of exploration decreases. So one could say that Q-learning incorporates the exploration as part of the algorithm. However, this exploration is "greedy", it settles with the first effective strategy it finds.
A simple and effective fix for the above problem is ε-greedy exploration – with probability ε choose a random action, otherwise go with the "greedy" action with the highest Q-value. DeepMind in their work actually decrease ε over time from 1 to 0.1 – in the beginning the system makes completely random moves to explore the state space maximally, and then it settles down to a fixed exploration rate.

g. Deep Q-learning Algorithm

This is the final deep Q-learning algorithm with experience replay:

Figure 11- Q-learn final algorithem

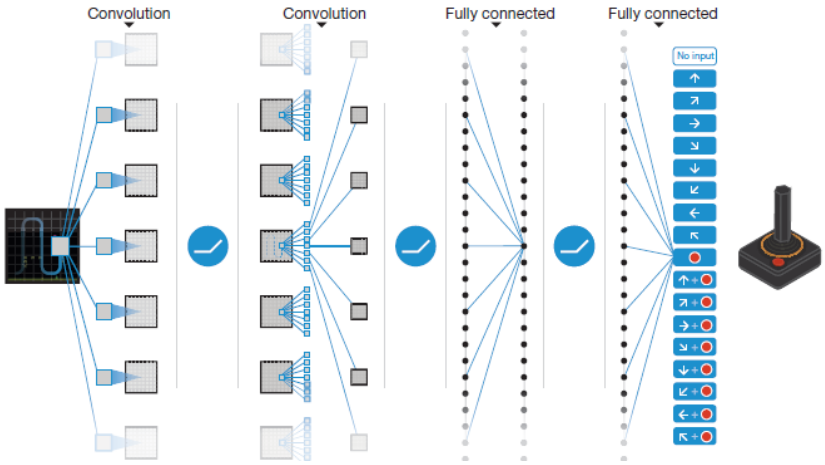
```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
      select an action a
            with probability ε select a random action
            otherwise select a = argmax_a' Q(s,a')
      carry out action a
      observe reward r and new state s'
      store experience <s, a, r, s'> in replay memory D

      sample random transitions <ss, aa, rr, ss'> from replay memory D
      calculate target for each minibatch transition
            if ss' is terminal state then tt = rr
            otherwise tt = rr + γmax_a' Q(ss', aa')
      train the Q network using (tt - Q(ss, aa))^2 as loss

      s = s'
until terminated
```

4. Related Work

a. DeepMind – Atari 2600

Perhaps the best-known success story of reinforcement learning.
They introduced a new deep learning model for reinforcement learning, and demonstrated its ability to master difficult control policies for Atari 2600 computer games, using only raw pixels as input. They also presented a variant of online Q-learning that combines stochastic minibatch updates with experience replay memory to ease the training of deep networks for RL. Their approach gave state-of-the-art results in six of the seven games it was tested on, with no adjustment of the architecture or hyper parameters.
Their algorithm only stores the last N experience tuples in the replay memory, and samples uniformly at random from D when performing updates. They worked directly with raw Atari frames, which are 210X160 pixel images with a 128-color palette, converted to gray-scale and down sampling it to a 110X84 image. The final input representation is obtained by cropping an 84X84 region of the image that roughly captures the playing area. The final cropping stage is required because the used the GPU implementation of 2D convolutions which expects square inputs. They used an architecture in which there is a separate output unit for each possible action, and only the state representation is an input to the neural network. The outputs correspond to the predicted Q-values of the individual action for the input state. The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network. They used the exact architecture for all seven Atari games.

Figure 12- DeepMind network architecture



b. DeepMind – AlphaGo

In this work DeepMind have developed a Go program, based on a combination of deep neural networks and tree search, that plays at the level of the strongest human players, thereby achieving one of artificial
Intelligence's "grand challenges". They have developed, for the first time, effective move selection and position evaluation functions for Go, based on deep neural networks that are trained by a novel combination of supervised and reinforcement

learning. We have introduced a new search algorithm that successfully combines neural network evaluations with Monte-Carlo rollouts. Their program AlphaGo integrates these components together, at scale, in a high-performance tree search engine.
Deep Mind passed the board position as a 19X19 image and used convolutional layers to construct a representation of the position. They use these neural networks to reduce the effective depth and breadth of the search tree: evaluating positions using a value network, and sampling actions using a policy network.
We train the neural networks using a pipeline consisting of several stages of machine learning.
They begin by training a supervised learning (SL) policy network, directly from expert human moves. This provides fast, efficient learning updates with immediate feedback and high quality gradients. They also train a fast policy that can rapidly sample actions during rollouts. Next, they train a reinforcement learning (RL) policy network, that improves the SL policy network by optimizing the final outcome of games of self-play. This adjusts the policy towards the correct goal of winning games, rather than maximizing predictive accuracy. Finally, they train a value network that predicts the winner of games played by the RL policy network against itself.
Their program AlphaGo efficiently combines the policy and value networks with MCTS.
They evaluated the distributed version of AlphaGo against Fan Hui, a professional 2 dan, and the winner of the 2013, 2014 and 2015 European Go championships. On 5–9th October 2015 AlphaGo and Fan Hui competed in a formal five game match. AlphaGo won the match 5 games to 0. This is the first time that a computer Go program has defeated a human professional player, without handicap, in the full game of Go; a feat that was previously believed to be at least a decade away.

5. Classic Q-learn

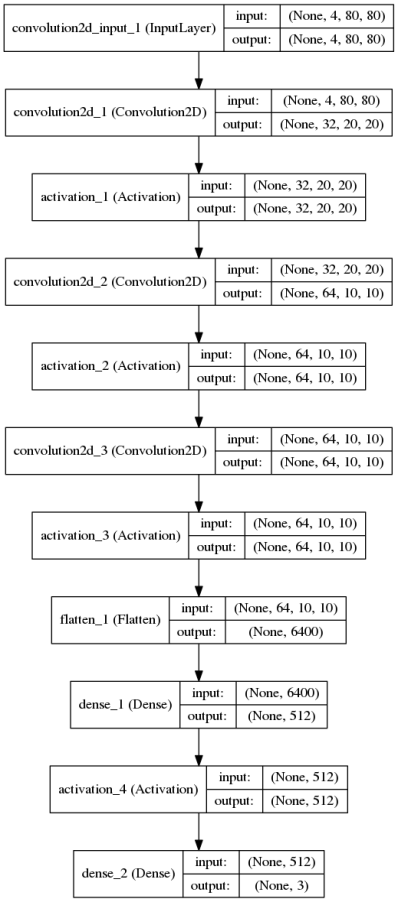
a. Network Architecture

We used similar network architecture as DeepMind did.
Our network architecture is: The input to the neural network consists of a 4x80x80 images. The first hidden layer convolves 32 filters of 8x8 with stride 4 and applies ReLU activation function. The second layer convolves a 64 filters of 4x4 with stride 2 and applies ReLU activation function. The third layer convolves a 64 filters of 3x3 with stride 1 and applies ReLU activation function. The final hidden layer is fully-connected consisted of 512 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action.

Figure 13- Building the model - code

```
def buildmodel():
    print("Now we build the model")
    model = Sequential()
    model.add(Convolution2D(32, 8, 8, subsample=(4,4),init=lambda shape, name: normal(shape, scale=0.01, name=name), border_mode='same',input_shape=(img_channels,img_rows,img_cols)))
    model.add(Activation('relu'))
    model.add(Convolution2D(64, 4, 4, subsample=(2,2),init=lambda shape, name: normal(shape, scale=0.01, name=name), border_mode='same'))
    model.add(Activation('relu'))
    model.add(Convolution2D(64, 3, 3, subsample=(1,1),init=lambda shape, name: normal(shape, scale=0.01, name=name), border_mode='same'))
    model.add(Activation('relu'))
    model.add(Flatten())
    model.add(Dense(512, init=lambda shape, name: normal(shape, scale=0.01, name=name)))
    model.add(Activation('relu'))
    model.add(Dense(ACTIONS,init=lambda shape, name: normal(shape, scale=0.01, name=name)))
```

Figure 14 - Network architecture

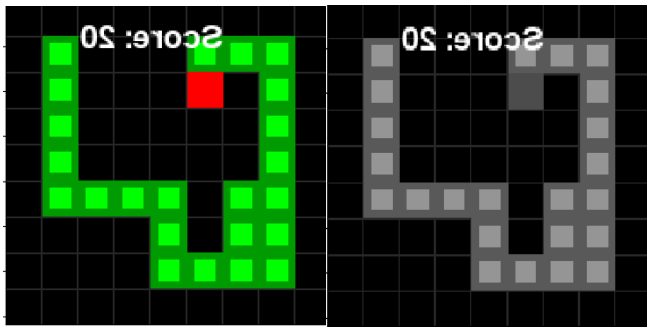


b. Image pre-processing

In order to make the code train faster, it is vital to do some image processing. We used similar image preprocessing method as DeepMind did.

1. Convert the color image into grayscale
2. Crop down the image size into 80x80 pixel
3. Stack 4 frames together before feed into neural network (This is one way for the model to be able to infer the velocity and direction information.

Figure 15 - Image pre-processing



- left image is original image and the right one is grey-scale image

Figure 16 - Image pre-processing code

```
x_t1 = skimage.color.rgb2gray(x_t1_colored)
x_t1 = skimage.transform.resize(x_t1,(80,80))
x_t1 = skimage.exposure.rescale_intensity(x_t1, out_range=(0, 255))

x_t1 = x_t1.reshape(1, 1, x_t1.shape[0], x_t1.shape[1])
s_t1 = np.append(x_t1, s_t[:, :3, :, :], axis=1)
```

c. Reward function

The reward function corresponds to game score.

- Each time the score rises the reward is +1.
- If you lose the reward is -1
- In some cases we used survivor reward - -0.1 (we want the agent to try to win the game as fast as possible).

The game return each step the reward for each taken action.

d. Experience Replay

As discussed in the theory chapter the most common trick to solve the instability from approximation of Q-value using non-linear functions like neural network is experience replay. During the gameplay all the episode are stored in replay memory D. (we used Python function deque() to store it). When training the network, random mini-batches from the replay memory are used instead of most the recent transition, which will greatly improve the stability.

Figure 17- Experience replay code

```
#only train if done observing
if t > OBSERVE:
    #sample a minibatch to train on
    minibatch = random.sample(D, BATCH)

    inputs = np.zeros((BATCH, s_t.shape[1], s_t.shape[2], s_t.shape[3]))   #32, 80, 80, 4
    targets = np.zeros((inputs.shape[0], ACTIONS))                         #32, 2

    #Now we do the experience replay
    for i in range(0, len(minibatch)):
        state_t = minibatch[i][0]
        action_t = minibatch[i][1]   #This is action index
        reward_t = minibatch[i][2]
        state_t1 = minibatch[i][3]
        terminal = minibatch[i][4]
        # if terminated, only equals reward
```

e. Exploration vs. Exploitation

As discussed in the theory chapter another issue in the reinforcement-learning algorithm is Exploration vs. Exploitation. How much of an agent's time should be spent exploiting its existing known-good policy, and how much time should be focused on exploring new, possibility better, actions? In order to maximize future reward, the agent need to balance the amount of time that they follow their current policy (this is called being "greedy"), and the time they spend exploring new possibilities that might be better. We used ϵ greedy approach. Under this approach, the policy tells the agent to try a random action some percentage of the time, as defined by the variable ϵ, which is a number between 0 and 1. The strategy will help the RL agent to occasionally try something new and see if we can achieve ultimate strategy. As time passes ϵ decreases, so less random actions are taken.

Figure 18- ϵ greedy code

```
elif random.random() <= epsilon:
    #print("----------Random Action----------")
    #raw_input("Press Enter to continue...")
    action_index = random.randrange(ACTIONS)
    a_t[action_index] = 1
```

f. Results

We tried to learn to play our five games. Except Squirrel eat Squirrel we succeeded in all of them.
Each game was tested by four parameters:

a. Score per epoch[2]
b. Reward per epoch
c. Loss function – we define the following loss function (error function):

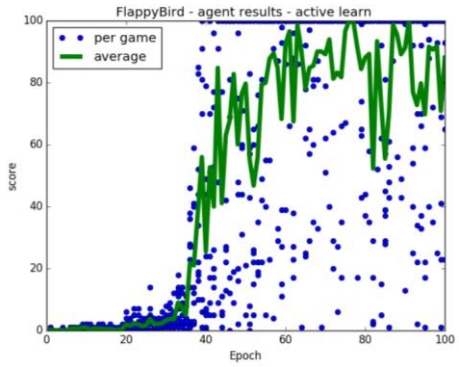
$$L = \left(r + \max_{a'} Q(s', a') - Q(s, a)\right)^2$$

d. Qmax

Each test consisted of experiment of ten games per epoch.

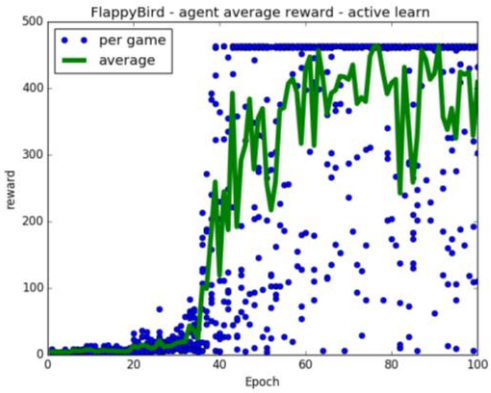
1. Flappy bird
   a. Score per epoch

Figure 19 - FlappyBird - Qleearn – scores



   b. Reward per epoch

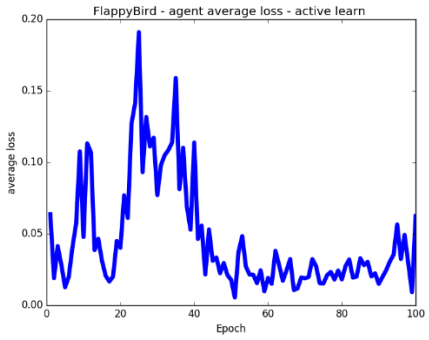
Figure 20- FlappyBird - Qleearn – reward



[2] Epoch is time period of 30 minutes

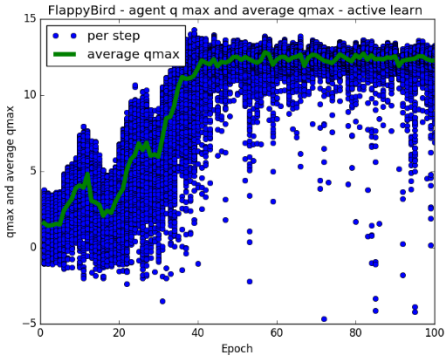
c. Loss function

Figure 21- FlappyBird - Qleearn - loss



d. Qmax

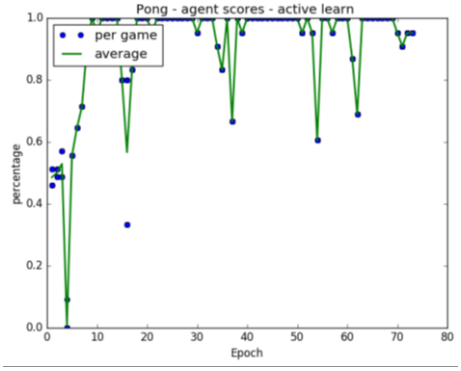
Figure 22- FlappyBird - Qleearn - qmax



2. <u>Pong</u>

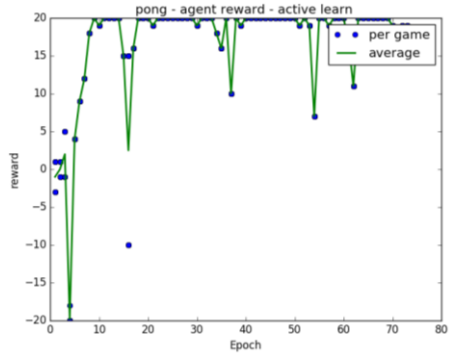
a. Score per epoch

Figure 23 - Pong - Qleearn - scores



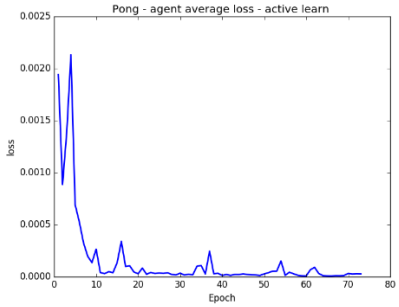
b. Reward per epoch

Figure 24 - Pong - Qleearn - reward



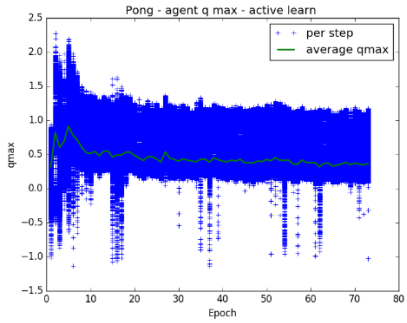
c. Loss function

Figure 25 - Pong - Qleearn - loss



d. Qmax

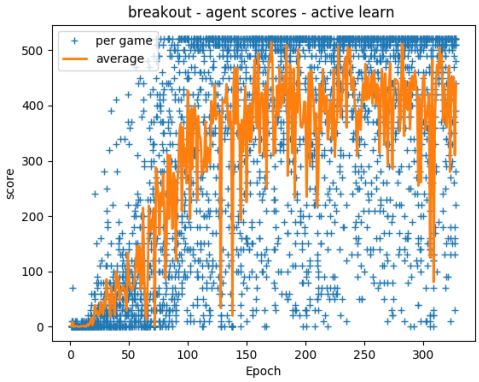
Figure 26 - Pong - Qleearn - Qmax



3. Breakout

a. Score per epoch

Figure 27 - Breakout - Qleearn - scores

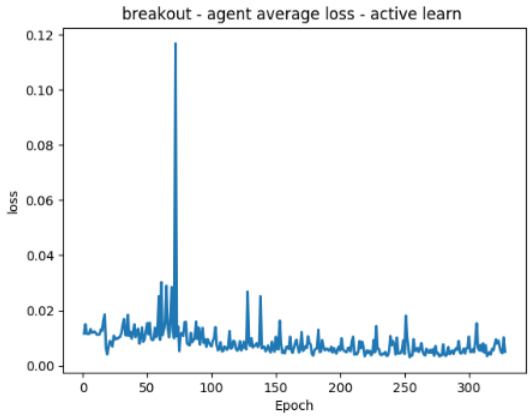


b. Reward per epoch

Figure 28 - Breakout – Qleearn - reward

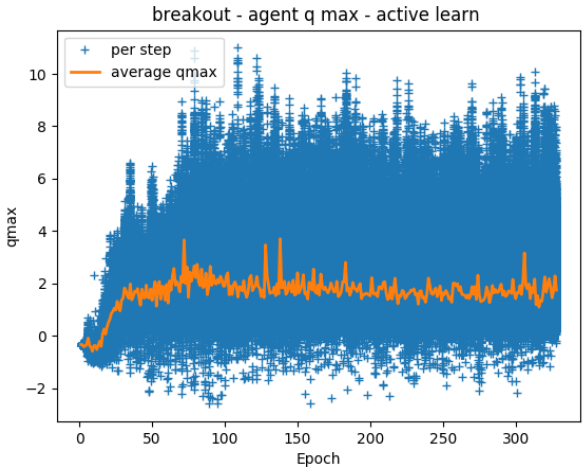
c. Loss function

Figure 29 - Breakout - Qleearn – loss



d. Qmax

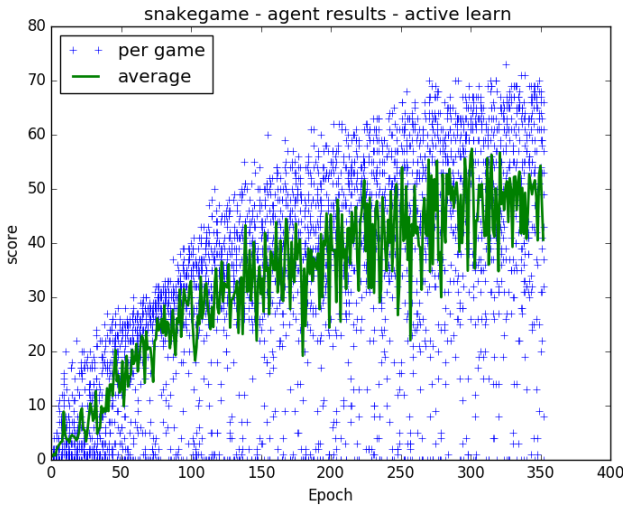
Figure 30 - Breakout - Qleearn – Qmax



4. Snake

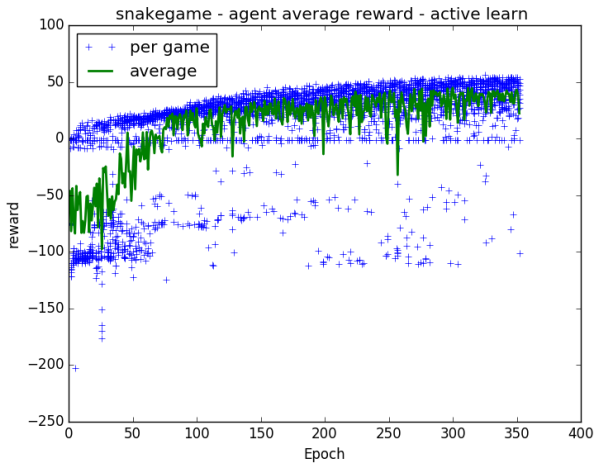
a. Score per epoch

Figure 31 - Snake - Qleearn - scores



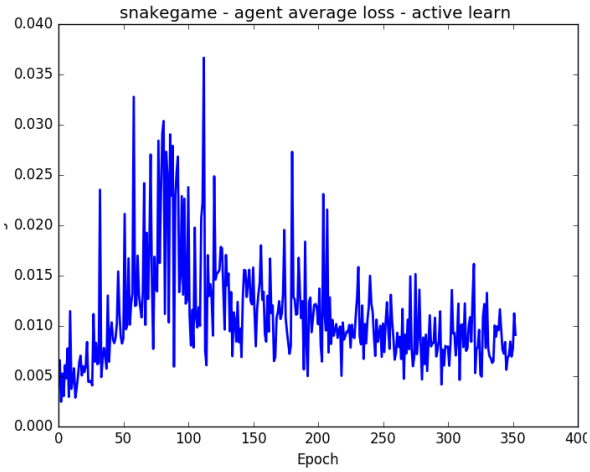
b. Reward per epoch

Figure 32 - Snake - Qleearn - reward



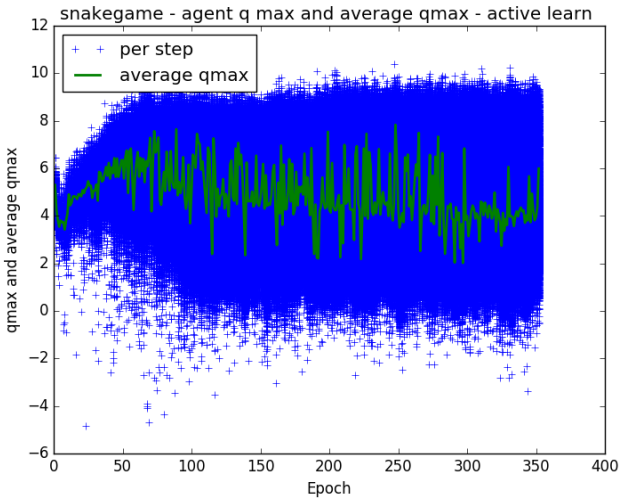
c. Loss function

Figure 33 - Snake - Qleearn – loss



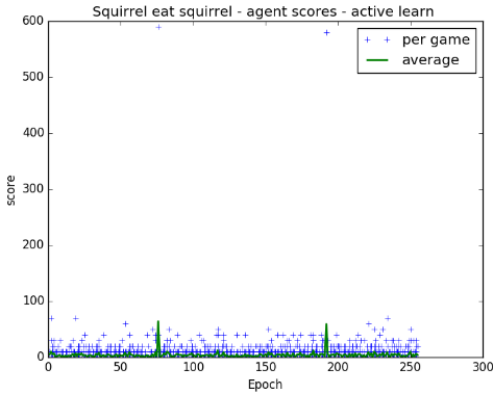
d. Qmax

Figure 34 - Snake - Qleearn – Qmax



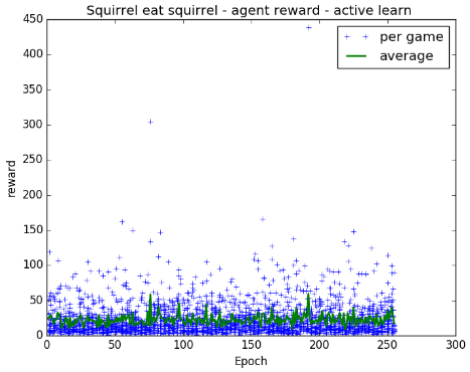
5. Squirrel eat Squirrel
   a. Score per epoch

Figure 35 – Squirrel eat Squirrel- Qleearn - scores



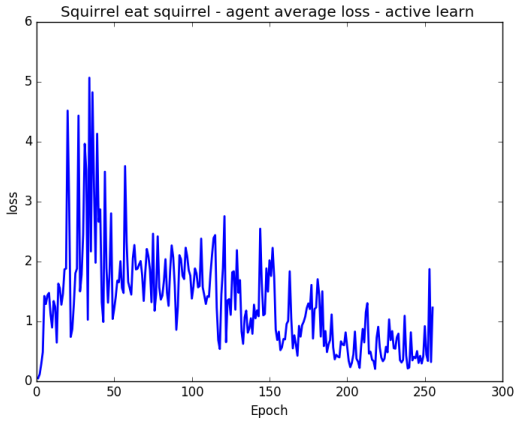
   b. Reward per epoch

Figure 36 - Squirrel eat Squirrel - Qleearn - reward



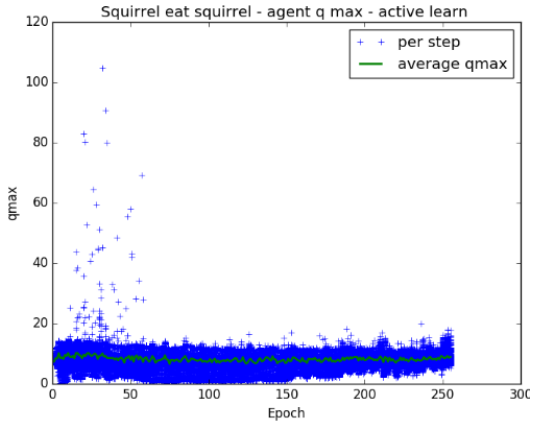
   c. Loss function

Figure 37 - Squirrel eat Squirrel - Qleearn – loss



d. Qmax

Figure 38 - Squirrel eat Squirrel - Qleearn – Qmax



g. Discussion

We succeeded creating an agent that plays successfully four games out of five.
We failed in the game Squirrel eat Squirrel probably because "our" squirrel looks exactly as the other squirrels so the network can't know which one is "our" squirrel. One possible solution is to remember the last four or more actions. If you know your last actions, you can understand which of the squirrels our player is.

In all four other games, we succeeded in creating an agent that can play the game like expert.
We can see that in all games the agent managed to get average score to be maximum possible score.
In supervised learning, one can easily track the performance of a model during training by evaluating it on the training and validation sets. In reinforcement learning, however, accurately evaluating the progress of an agent during training can be challenging. Since our evaluation metric, is the total reward the agent collects in an episode or game averaged over a number of games. The average total reward metric tends to be very noisy because small changes to the weights of a policy can lead to large changes in the distribution of states the policy visits. The score and reward figures show how the average total reward evolves on the games. All averaged reward plots are indeed quite noisy, giving one the impression that the learning algorithm is not making steady progress.
Another, more stable, metric is the policy's estimated action-value function Q, which provides an estimate of how much discounted reward the agent can obtain by following its policy from any given state. We collect a fixed set of states by running a random policy before training starts and track the average of the maximum predicted Q for these states. The qmax figures show that average predicted Q increases much more smoothly than the average total reward obtained by the agent. In addition to seeing relatively smooth improvement to predicted Q during training, we did not experience any divergence issues in any of our experiments. This suggests that, despite lacking any theoretical convergence guarantees, our method is able to train large neural networks using a reinforcement-learning signal and stochastic gradient descent in a stable manner.

We also can see that our lose (error) getting smaller over the learning process. In some cases we have small error in the beguiling which increases later. This is due to weight initialization (the initialization is through random function).
Moreover, as expected there is a big correlation between reward graph and score graph (we choose the reward corresponding to score so it is expected behavior).

The biggest problem we had with this method was that it is very slow. It took us more than seven days to make the agent play breakout or snake.

6. Passive learn vs Active learn

Due to q-learn being very slow, we tried to find a way to make it study faster.
There are two methods of studding:

- Passive Learning
  With passive learning, agents require a model of the environment. This model tells the agent what moves are legal and what the results of actions will be. This is particularly useful because it allows the agent to look ahead and make better choices about what actions should be taken. However, passive learning agents have a fixed policy and this limits their ability to adapt to or operate in unknown environments.
  In passive learning the action is chosen by outside source such as recording of games, human source etc. (DeepMind used it with alphaGo)

- Active Learning
  Unlike passive learning agents, active learning agents do not have a fixed policy. Active learning agents must learn a complete model of the environment.
  This means that the agent must determine what actions are possible at any given state since it is building a model of its environment and does not yet have an optimal policy. This allows active learning agents to learn how to effectively operate in environments that are initially unknown. However, the lack of a fixed policy slows the rate at which the agent learns the optimal behaviors in its environment.

Classic q-learn is active learning. We tried to make the process of studding much faster with passive learn.

7. Imitating player – passive learn

Our network is the same as we used in active learning. The only thing we changed is the reward function

a. Reward function

In this case, the reward is per each action and decided by the correlation between agent's action and source action.

- If the agent and the source make the same decision :reward is +1
- Different decision : reward is -1

b. Results

We tried this method on three games: Pong, Breakout and snake.
In pong we used as the source an agent which never losses, in breakout we used very good player –see figure 27, and in snake we used a beginner player – see figure 28.

Figure 39 - Breakout agent's scores – 500 first games

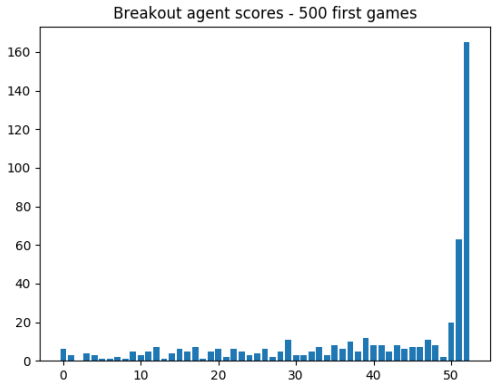
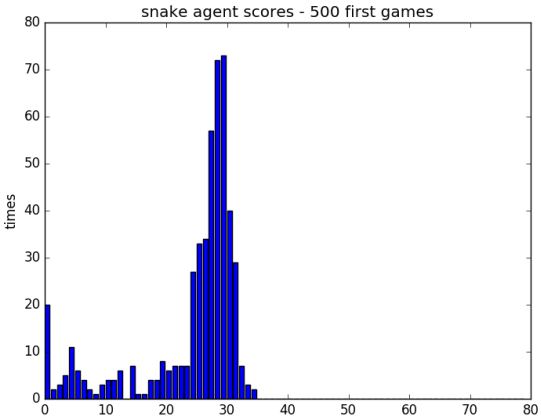


Figure 40 – Snake agent's scores - – 500 first games

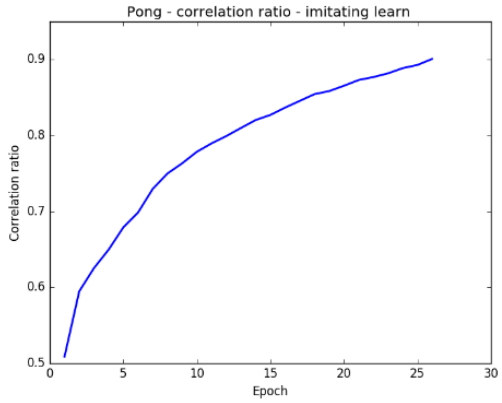


Each game was tested by five parameters:

a. Correlation ratio between agent action and source action
b. Score per epoch
c. Reward per epoch
d. Loss function
e. Qmax

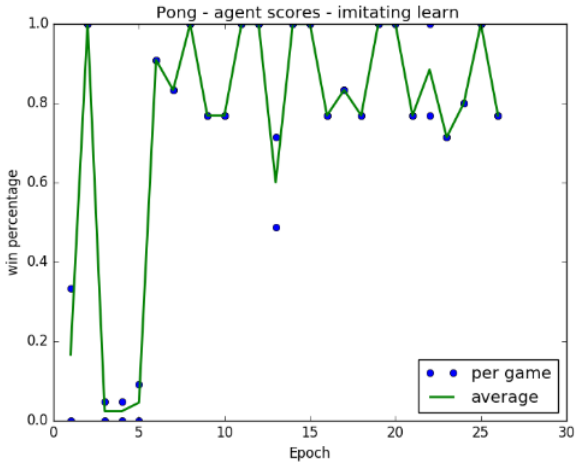
1. Pong
   a. Correlation ratio between agent action and source action

Figure 41 - Pong - Imitating player - correlation ratio



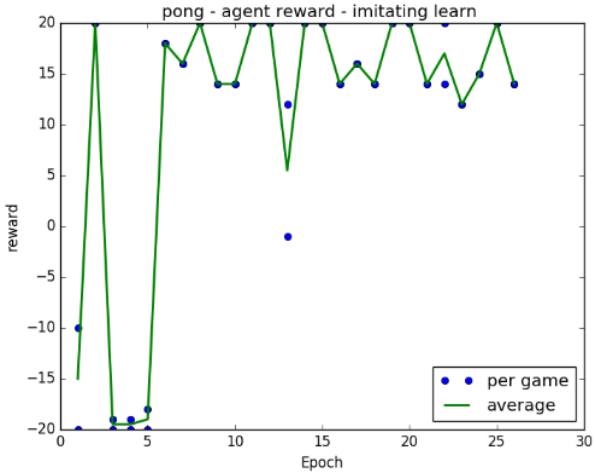
   b. Score per epoch

Figure 42 - Pong - Imitating player - scores



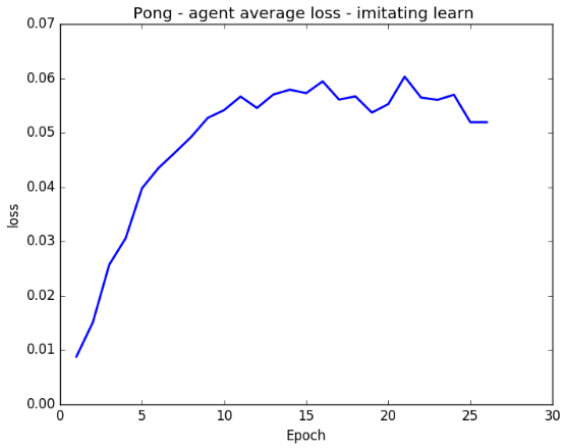
   c. Reward per epoch

Figure 43 - Pong - Imitating player - reward



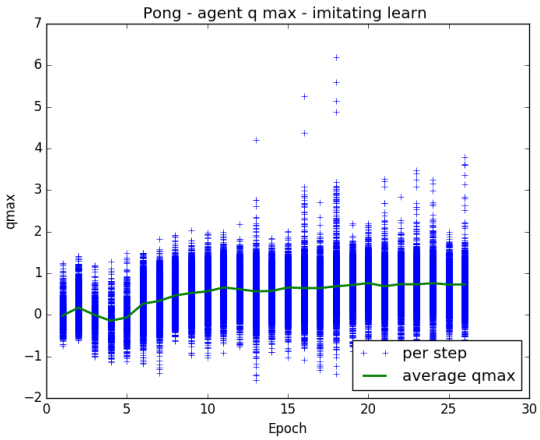
d. Loss function

Figure 44 - Pong - Imitating player - loss



e. Qmax

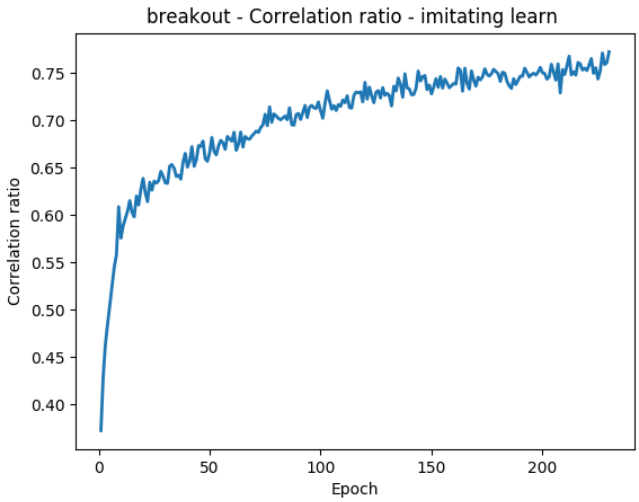
Figure 45 - Pong - Imitating player - Qmax



2. <u>Breakout</u>

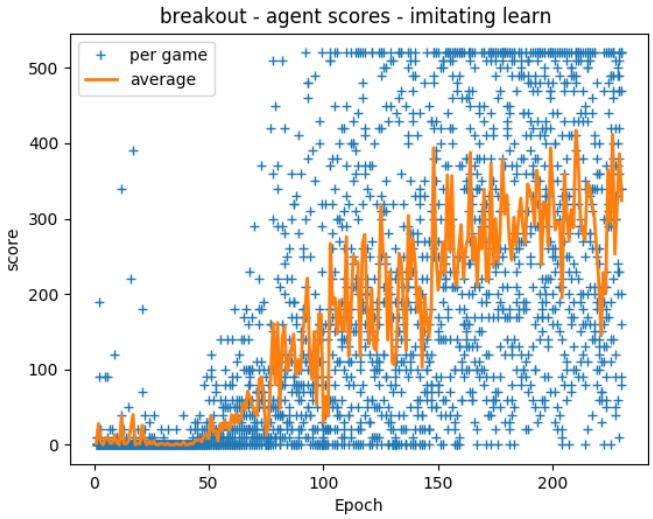
a. Correlation ratio between agent action and source action

Figure 46 - Breakout - Imitating player – Corerelation Ratio



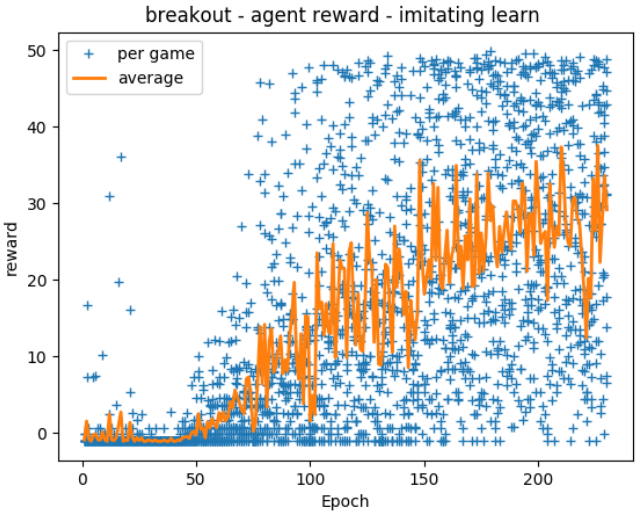
b. Score per epoch

Figure 47 - Breakout - Imitating player - Score



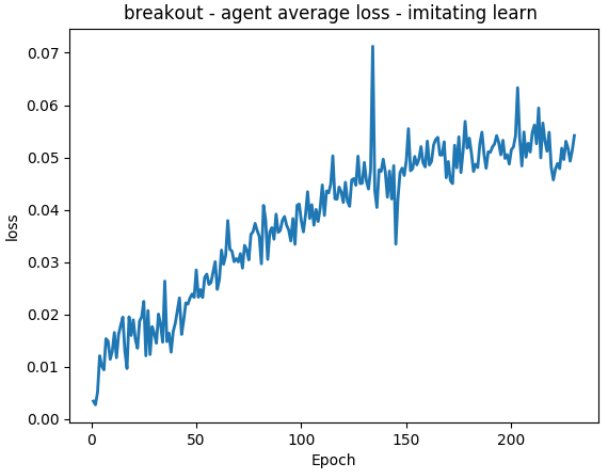
c. Reward per epoch

Figure 48 - Breakout - Imitating player - Reward



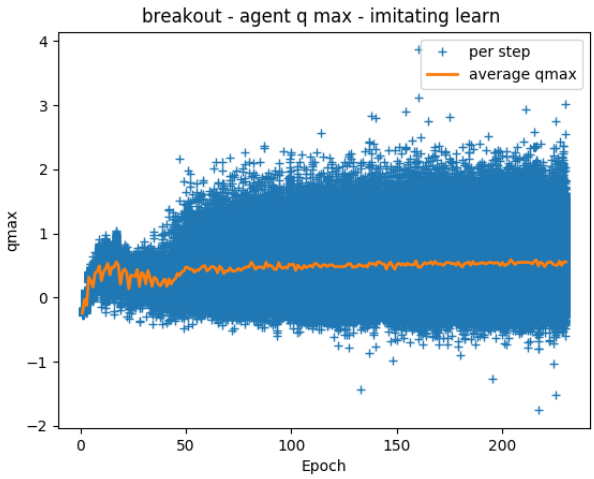
d. Loss function

Figure 49 - Breakout - Imitating player - Loss



e. Qmax

Figure 50 - Breakout - Imitating player - Qmax

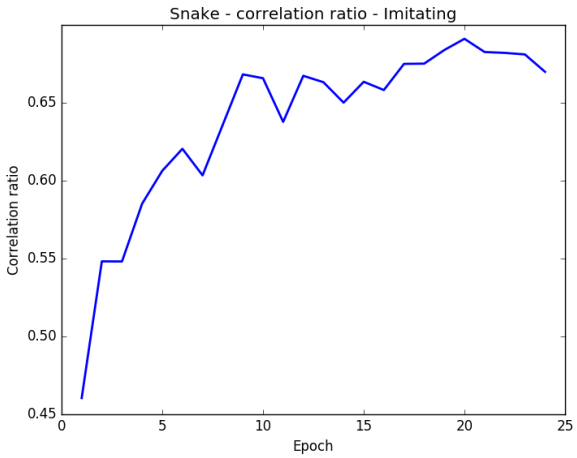


3. Snake

In this game we switched to classic q-learn after 24 epoches.

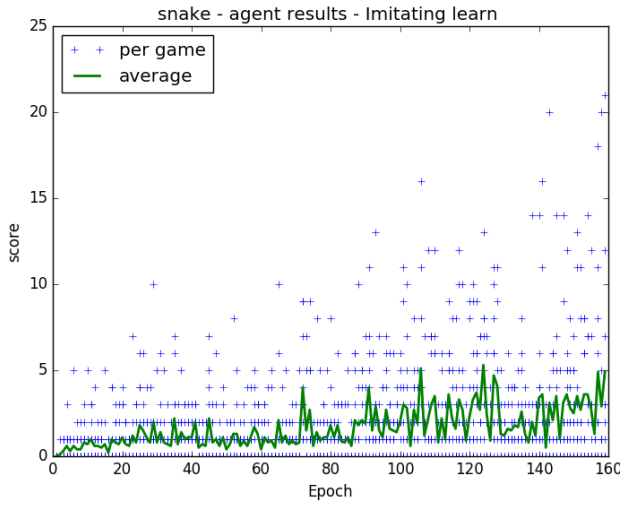
a. Correlation ratio between agent action and source action

Figure 51 - Snake - Imitating player – Correlation Ratio



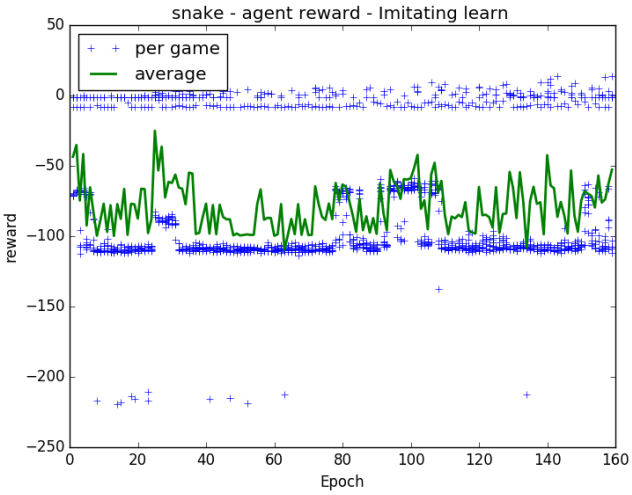
b. Score per epoch

Figure 52 - Snake - Imitating player - Score



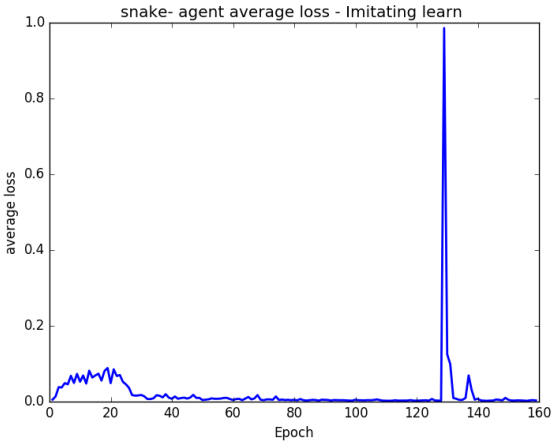
c. Reward per epoch

Figure 53 - Snake - Imitating player – Reward



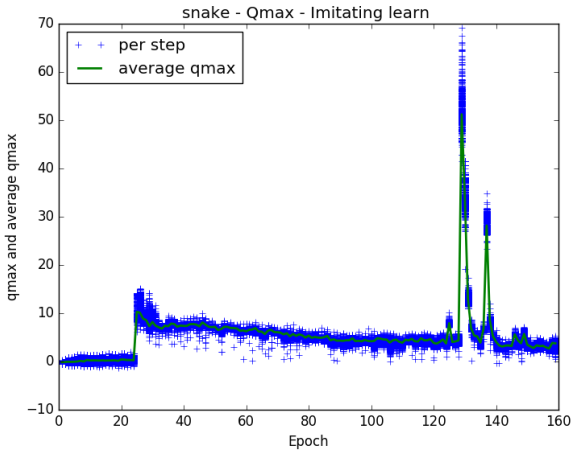
d. Loss function

Figure 54 - Snake - Imitating player - Loss



e. Qmax

Figure 55 - Snake - Imitating player - Qmax



c. Discussion

In this method, you create an agent that imitates the player. He never will be better than the source player.
In pong, we have source that never loses. By this method we created successfully very fast an agent that never losses.
We can see that the error increases over time. This is due to the reward function in this method.

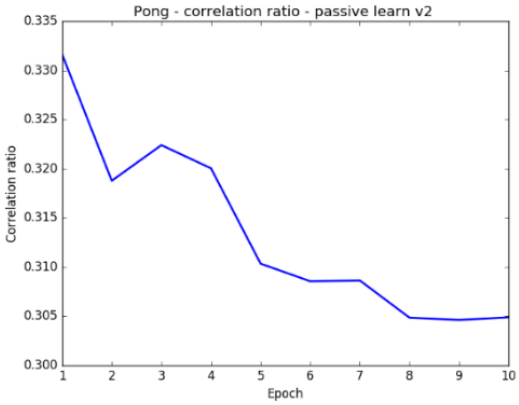
8. passive learn – v2

Our network is the same as we used in active learning. The only difference between this method and classic q-learn is that the action is decided by source player and not by the agent itself.

a. Results

We tried this method on Pong game only.
We got the following graph for Correlation ratio

Figure 56 - Pong - Passive learn - v2 - corrolation



b. Discussion

We can see that the correlation ratio decreases instead of increasing. This is due to there is no learning for highest initial weight so it will stay the preferred agent's action. Let us try to upgrade this method

9. Hybrid Q-Learn

In this method each 15 minutes we change from active learn to passive learn. In this method, you alternate between active and passive learn.
Our network is the same as we used in active learning.

a. Reward function

In this case, the reward is per each action and decided by the correlation between agent's action and source action.

- If the agent and the source make the same decision :reward is +1
- Different decision : reward is -1

b. Results

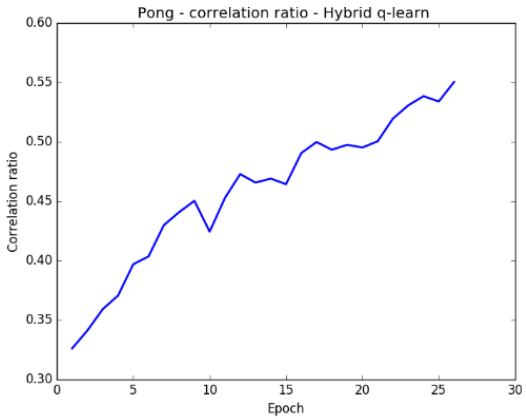
We tried this method on three games: Pong, Breakout and snake.
In pong we used as the source an agent which never losses, in breakout we used very good player –see figure 27, and in snake we used a beginner player – see figure 28.

Each game was tested by five parameters:

a. Correlation ratio between agent action and source action
b. Score per epoch
c. Reward per epoch
d. Loss function
e. Qmax

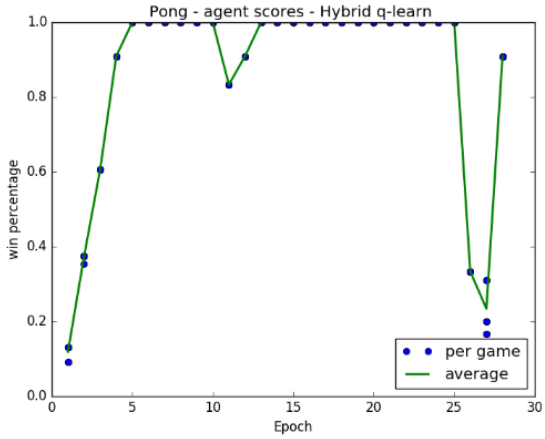
1. Pong
   a. Correlation ratio between agent action and source action

Figure 57 - Pong - Hybrid Q-learn - correlation ratio



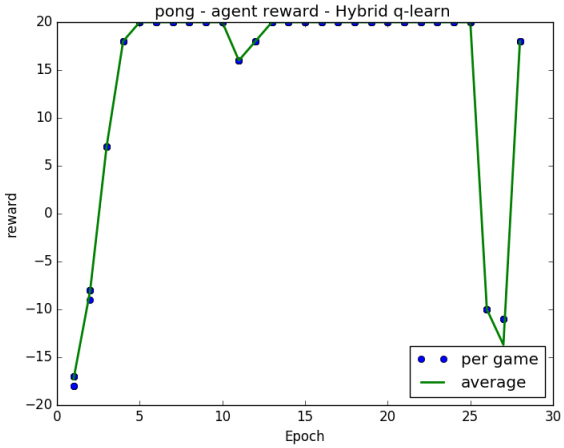
   b. Score per epoch

Figure 58- Pong - Hybrid Q-learn - scores



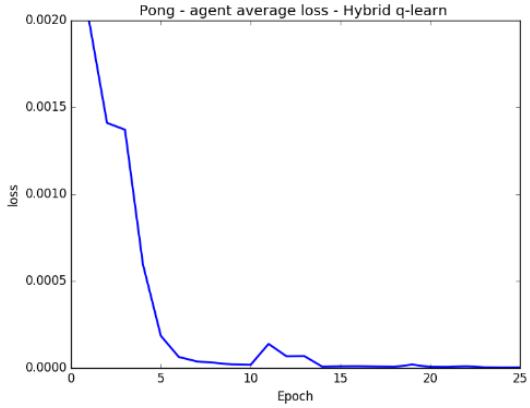
c. Reward per epoch

Figure 59- Pong - Hybrid Q-learn - reward



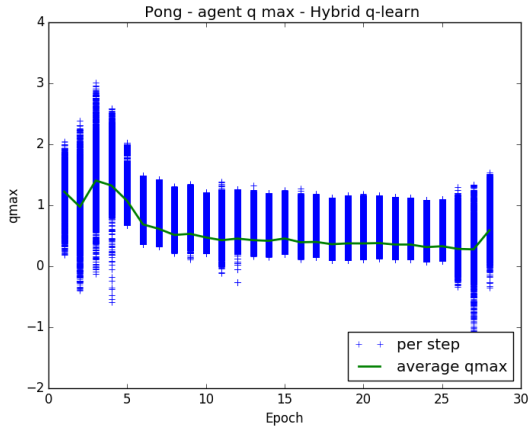
d. Loss function

Figure 60- Pong - Hybrid Q-learn - loss



e. Qmax

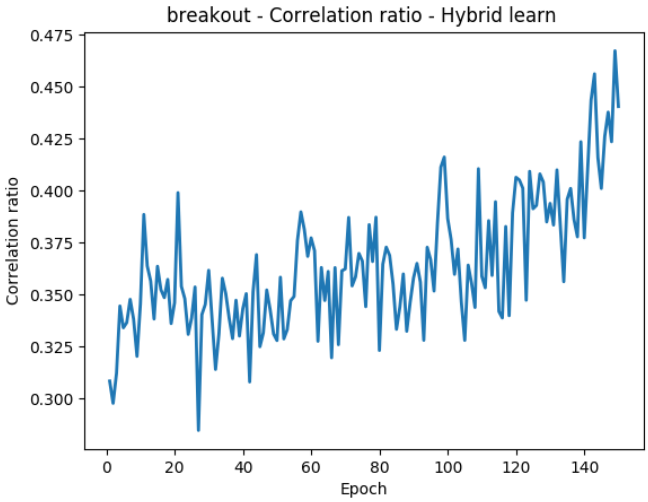
Figure 61- Pong - Hybrid Q-learn - Qmax



2. Breakout

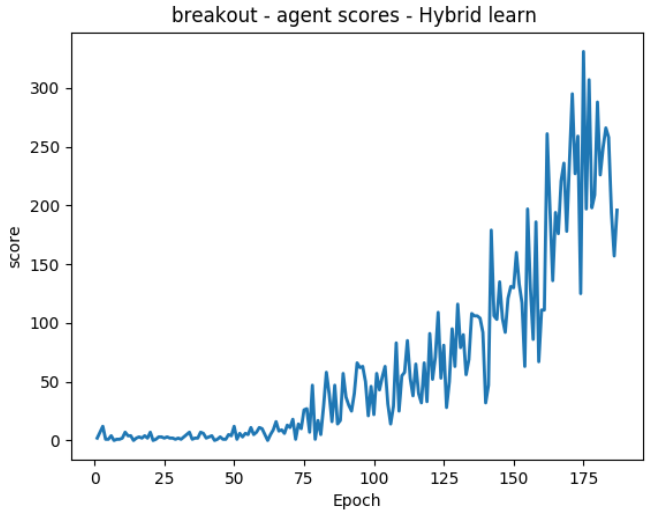
a. Correlation ratio between agent action and source action

Figure 62- Breakout - Hybrid Q-learn – Correlation Ratio



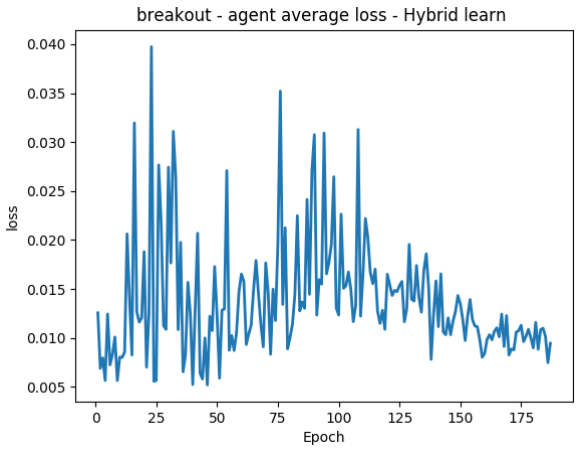
b. Score per epoch

Figure 63- Breakout - Hybrid Q-learn - Score



c. Loss function

Figure 64- Breakout - Hybrid Q-learn - Loss



c. Discussion

We succeeded creating an agent that plays successfully all three games. In this method there is active learning involved so the agent gets better scores then the source (in games were the source is not perfect player).
In all other aspects we get results very similar to classic (active) q-learn but a little bit faster

10. Comparison between methods

We will compare between the three methods we applied based on five parameters:

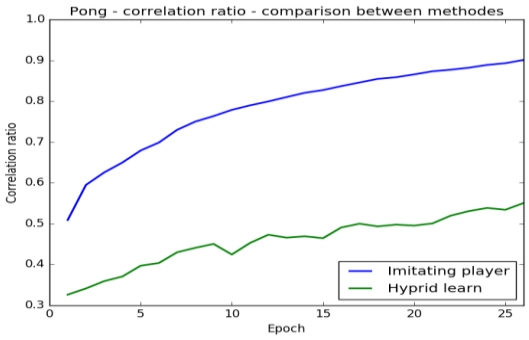
- Correlation ratio
- Score per epoch
- Reward per epoch
- Loss function
- Qmax

a. Pong game

In this game we used as a source a player that never losses.

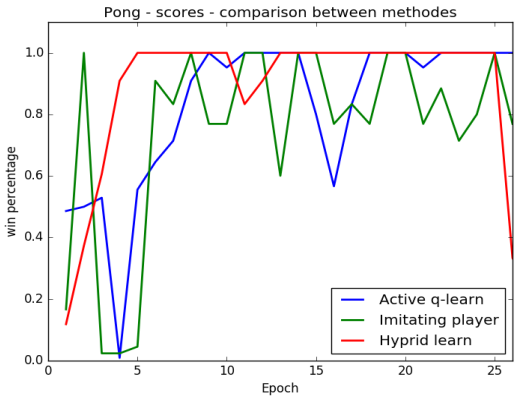
a. Correlation ratio

Figure 65 - Pong - correlation ratio - comparison between methodes



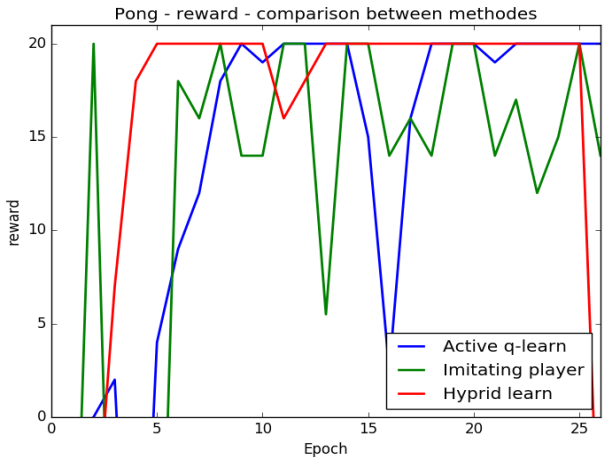
b. Score per epoch

Figure 66 - Pong - scores - comparison between methodes



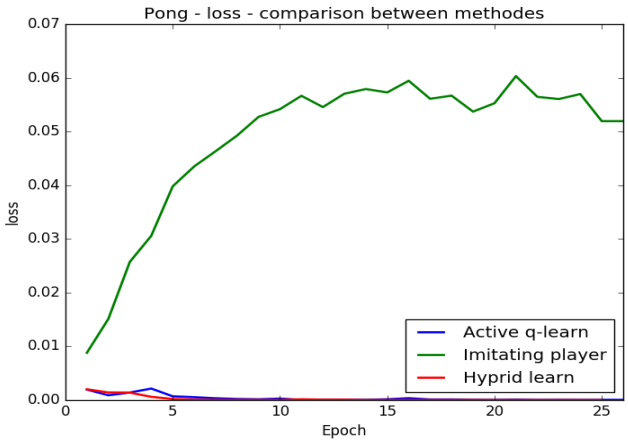
c. Reward per epoch

Figure 67 - Pong - reward - comparison between methodes



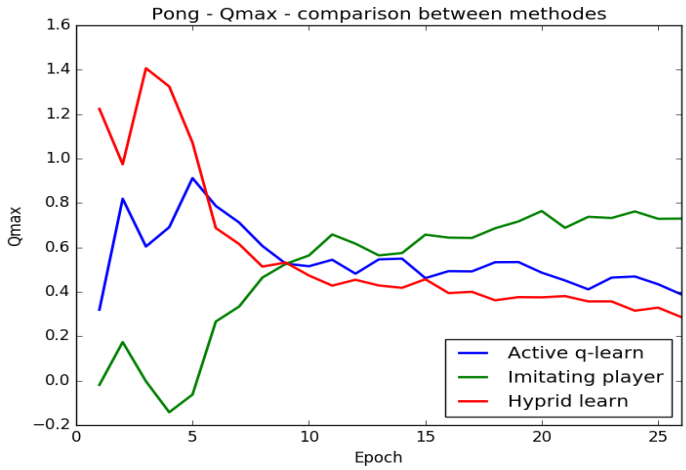
d. Loss function

Figure 68 - Pong - loss - comparison between methodes



e. Qmax

Figure 69 - Pong - Qmax - comparison between methodes

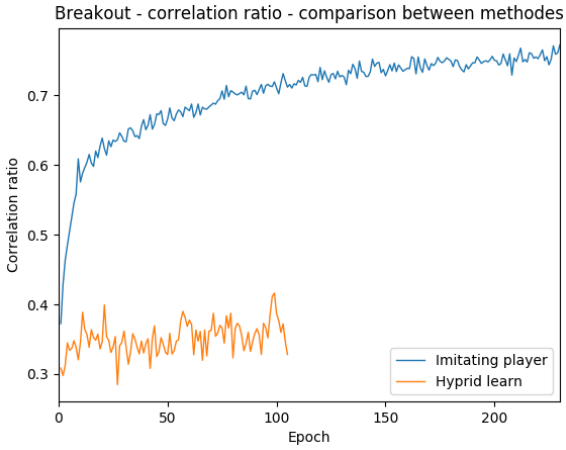


b. Breakout game

In this game we used as an source a player that never losses.

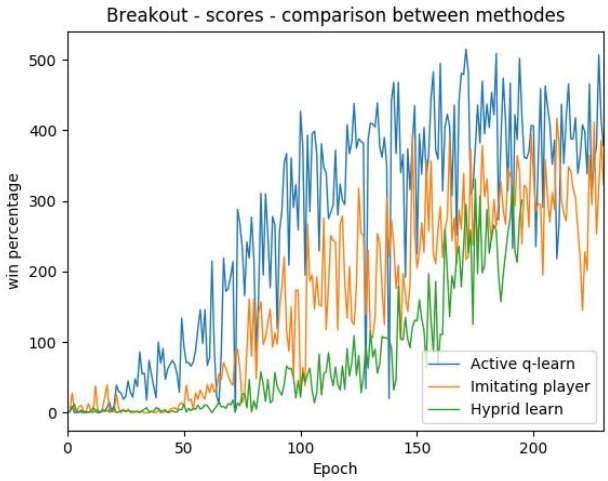
a. Correlation ratio

Figure 70 - Breakout - correlation ratio - comparison between methodes



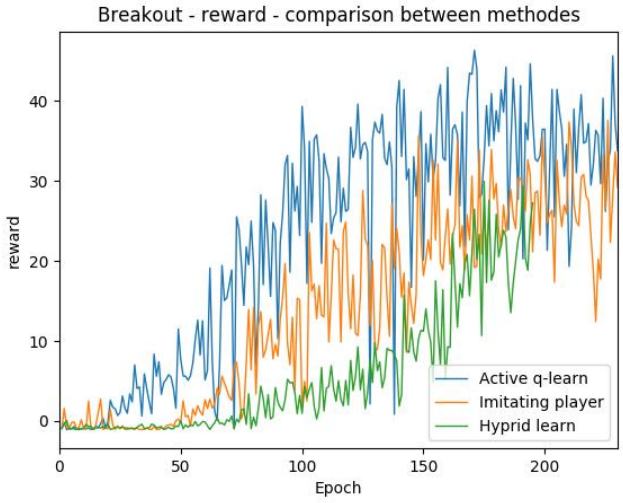
b. Score per epoch

Figure 71 Breakout - scores - comparison between methodes



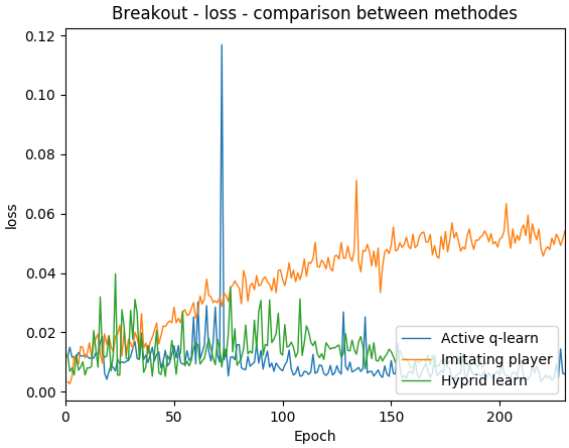
c. Reward per epoch

Figure 72 - Breakout - reward - comparison between methodes



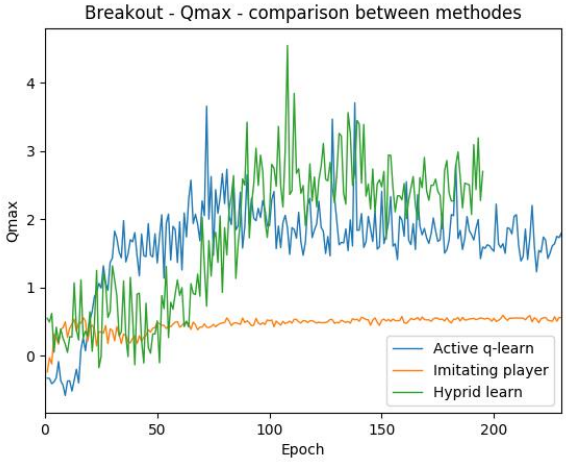
d. Loss function

Figure 73 - Breakout - loss - comparison between methodes



e. Qmax

Figure 74 - Breakout - Qmax - comparison between methodes

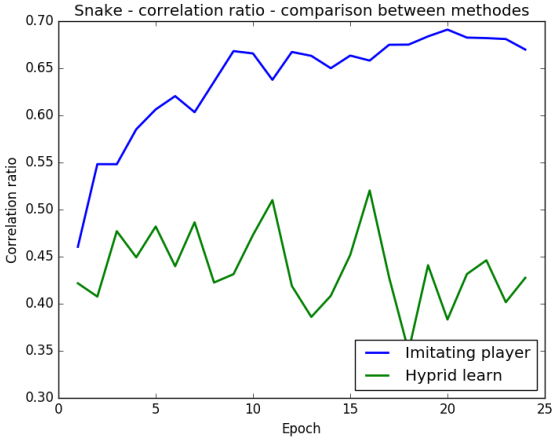


c. Snake game

In this game we used as a source a player that never losses.
After 66 epochs we took the weights file and run classic active q-learn.

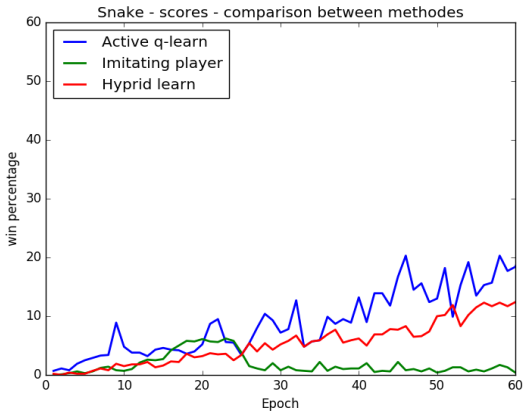
a. Correlation ratio

Figure 75 - Snake - correlation ratio - comparison between methodes



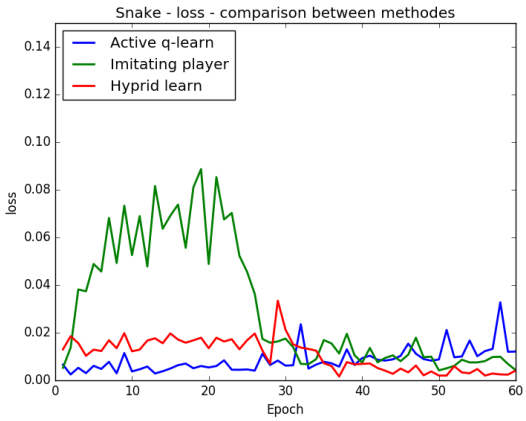
b. Score per epoch

Figure 76 Snake - scores - comparison between methodes



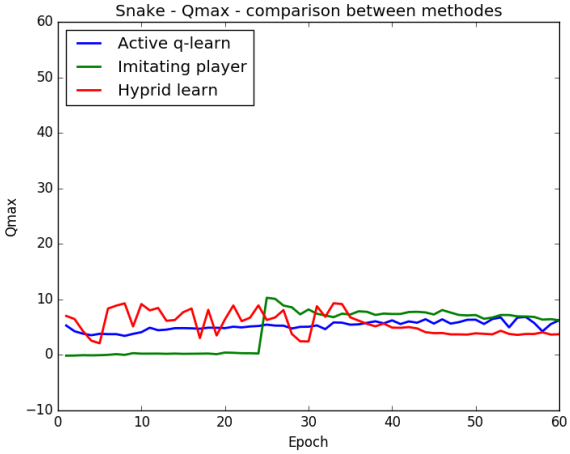
c. Loss function

Figure 77 - Snake - loss - comparison between methodes



d. Qmax

Figure 78 - Snake - Qmax - comparison between methodes



d. <u>Discussion</u>

We can see that imitating player is much faster. If you have a perfect source you can use imitating method and you will get pretty fast an agent that plays the

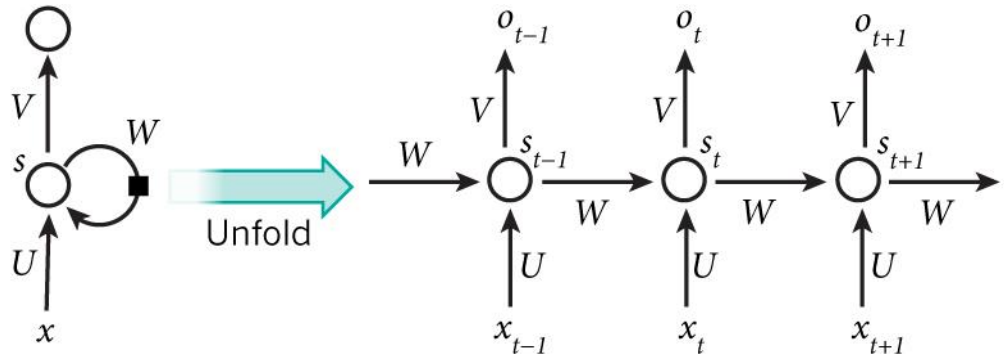
game as expert. The biggest problem with this method is that the agent is as good as the source. It can never be better or exceed human ability.
Hybrid Q-learn is slower then imitating but still faster than classic. The biggest plus in this method is that it can exceed the source ability because there is active learn involved. In addition, you can take the weights and keep studying from there while in imitating it starts from the beginning.

11. Future work

a. RNN

The idea behind RNNs is to make use of sequential information. In a traditional neural network we assume that all inputs (and outputs) are independent of each other. But for many tasks that's a very bad idea. If you want to predict the next word in a sentence you better know which words came before it. RNNs are called *recurrent* because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Another way to think about RNNs is that they have a "memory" which captures information about what has been calculated so far. In theory RNNs can make use of information in arbitrarily long sequences, but in practice they are limited to looking back only a few steps (more on this later). Here is what a typical RNN looks like:

Figure 79 - RNN



The above diagram shows a RNN being *unrolled* (or unfolded) into a full network. By unrolling we simply mean that we write out the network for the complete sequence. For example, if the sequence we care about is a sentence of 5 words, the network would be unrolled into a 5-layer neural network, one layer for each word. The formulas that govern the computation happening in a RNN are as follows:

- x_t is the input at time step t. For example, x_1 could be a one-hot vector corresponding to the second word of a sentence.
- s_t is the hidden state at time step t. It's the "memory" of the network. $s_t$is calculated based on the previous hidden state and the input at the current step: $s_t = f(Ux_t + Ws_{t-1})$. The function $f$usually is a nonlinearity such as tanh or ReLU. s_{-1}, which is required to calculate the first hidden state, is typically initialized to all zeroes.
- o_t is the output at step t. For example, if we wanted to predict the next word in a sentence it would be a vector of probabilities across our vocabulary. $o_t = \mathrm{softmax}(Vs_t)$.

There are a few things to note here:

- You can think of the hidden state $s_t$as the memory of the network. $s_t$captures information about what happened in all the previous time steps. The output at step $o_t$is calculated solely based on the memory at time t. As briefly mentioned above, it's a bit more complicated in practice because $s_t$typically can't capture information from too many time steps ago.
- Unlike a traditional deep neural network, which uses different parameters at each layer, a RNN shares the same parameters (U, V, W above) across all steps. This reflects the fact that we are performing the same task at each step, just with different inputs. This greatly reduces the total number of parameters we need to learn.
- The above diagram has outputs at each time step, but depending on the task this may not be necessary. For example, when predicting the sentiment of a sentence we may only care about the final output, not the sentiment after each word. Similarly, we may not need inputs at each time step. The main feature of an RNN is its hidden state, which captures some information about a sequence.

b. DCGAN (Deep Convolutional Generative Adversarial Network)

To build a DCGAN, we create two deep neural networks. Then we make them fight against each other, endlessly attempting to out-do one another. In the process, they both become stronger.

References

- http://keras.io/
- http://deeplearning.net/software/theano/
- Demystifying Deep Reinforcement Learning, Tambet Matiise, 2015
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller from DeepMind Technologies, **Playing Atari with Deep Reinforcement Learning**, arXiv:1312.5602, 19 Dec 2013
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou,Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg & Demis Hassabis, **Human-level control through deep reinforcement learning**. Nature, 2015
- David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, Demis Hassabis, **Mastering the Game of Go with Deep Neural Networks and Tree Search**
- Yann LeCun, , Yoshua Bengio, and Geoffrey Hinton, **Deep learning,** Nature, 521, 28 May 2015