A close up of a logo

Description automatically generated

**Master Studies**

**Project: Training Reinforcement Learning agent in simulated gaming environment “Flappy Bird”.**

Field of Study: Advanced Analytics-Big Data. Advanced Simulation Modelling

|  |  |
| --- | --- |
| **Student names:**  Oleksandr Bachynsky | **Album nr:**  83457 |
| Oleksandr Romanchenko | 83459 |
| Nazarii Melnyk | 85471 |

----------------------------------------

Instructor: Anna Szczurek

----------------------------------------

Warsaw, 2020

**Description of the task**

Our environment is a mobile game Flappy Bird where player (or the agent in our case) has to maintain the position of the bird and continue flying between the pipes as long as it can. Every time the bird gets into terminal state (hit the pipe, falls down or flies out of upper boundary) game starts from the beginning.

A close up of a device

Description automatically generated

To implement this, we’ve used a PLE library (PyGame Learning Environment) - extension to PyGame (python library for game development) for Reinforcement Learning tasks.

The only parameter that can be changed in environment is the pipe gap (the gap in pixels left between the top and bottom pipes, default: 100).

**PLE API**

* init() – initializes an environment after instance of object is created
* getActionSet() – returns [None, 119]. Corresponds to doing nothing (falling) or accelerating upward.

119 is ASCII number for “w” which is used in games for upward/forward movement

* act(action) – performs the action and returns the reward
* game\_over() – returns True if terminal state
* reset\_game() – goes back to start of an episode

**Theory of Reinforcement Learning**

The reinforcement learning problem is meant to be a straightforward framing of the problem of learning from interaction to achieve a goal. The learner and decision-maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. These interact continually, the agent selecting actions and the environment responding to those actions and presenting new situations to the agent. The environment also

A close up of a logo

Description automatically generated

*fig.1 - The agent-environment interaction in reinforcement learning*

gives rise to rewards, special numerical values that the agent tries to maximize over time. A complete specification of an environment defines a task, one instance of the reinforcement learning problem. More specifically, the agent and environment interact at each of a sequence of discrete time steps, t = 0, 1, 2, 3... At each time step t, the agent receives some representation of the environment's state, , where is the set of possible states, and on that basis selects an action, , where , is the set of actions available in state . One-time step later, in part as a consequence of its action, the agent receives a numerical reward, , and finds itself in a new state, . At each time step, the agent implements a mapping from states to probabilities of selecting each possible action. This mapping is called the agent's policy and is denoted , where , is the probability that if . Reinforcement learning methods specify how the agent changes its policy as a result of its experience. The agent's goal, roughly speaking, is to maximize the total amount of reward it receives over the long run.

The goal of Reinforcement Learning: given the current state we are in, choose the optimal action which will maximize the long-term expected reward provided by the environment.

In practice it’s solved using dynamic programming paradigm and calculates the final reward recursively. Bellman equation (or so-called value function) helps us to do that.

is called the state-value function for policy .

The value of state under a policy is

the value of any state

if the agent starts in state and then uses the policy to choose its actions for all time steps

Advantage of MDP framework: good old concept which has proven to be robust and can be applied to a many different tasks.

Disadvantage: uses a lot of computational power to calculate cumulative reward of all steps ahead.

To assess the state, our agent uses the following function provided by PLE.FlappyBird classA screenshot of a cell phone

Description automatically generated

Reward system for flappy bird:

For each pipe it passes through it gains a positive reward of +1. Each time a terminal state is reached it receives a negative reward of -1.

**Evolution Strategy theory**

There is another algorithm for solving RL tasks – evolution strategy.

ES in the context of RL:

* No complicated formulas!
* No MDP, no Bellman, no value functions etc.
* Environment doesn’t even have to fit the MDP framework
* If environment is more complex than an MDP – ES doesn’t care!
* Works well when reward is delayed or episode is very long, we only care about the number at the end
* No need for discounting “future” rewards
* Less hyperparameters – learning rate, population size, noise deviation

Let’s start with some motivation with biological evolution:

* DNA is a code that describes us (hair color, height, etc.)
* Assume we are dealing with single celled organisms that copy themselves to produce offspring (rather than 2 parent organisms which is more complicated)
* DNA copying isn’t perfect!
* Mutations can be good or bad:

you can be more athletic or more susceptible to being sick

* But overall the offspring is similar to a parent

Evolution strategy also may be described in the following way. Let’s use the optimization example of hill climbing:

A close up of a map

Description automatically generated

fig.14 – Illustration of algorithm

Usually, pseudocode for such type of problem looks like:

A screenshot of a cell phone

Description automatically generated

We start at some random point w0. We also going to get the value of the function we want to optimize at this point w0. Let’s assume it’s somewhere at the bottom of the hill. Then, in the loop, we are going to add some random noise to w0 and call it w. Next, we are going to evaluate our function at the value of w. It could be better or worse than our previous value because the noise we added was random (just like a random mutation in a DNA string). If it’s worse – we throw it away but if it’s better – we keep it and make it a new value of w0. As we do it iteratively, eventually we are going to get to the top of the hill. The only way we end up keeping a new value of w0 is if we find that it gives us an improvement on our function value. This already sounds a lot like natural selection discussed before but with 1 important point missing: in each generation we don’t just have one offspring, we have multiple! But how we update weight w if we have multiple offspring who will perform with the various degree of success? We might say: throw out the bad ones and keep good ones, but how can we combine multiple good offspring? What if 2 offspring are equally good and they have different weights? In fact, having multiple offspring could very useful in reinforcement learning since even with the same policy playing the episode multiple times will yield a different reward each time, therefore aggregating the results of multiple runs from multiple offspring! Possibly playing multiple episode per offspring could be useful in estimating the true expected reward.

We can present the algorithm with the following pseudocode:

A screenshot of a cell phone

Description automatically generated

fig.15 – Illustration of algorithm

First, we have to introduce a few more variables:

* Learning rate (a hyper-parameter that controls how much we are adjusting the weights of our network)
* noise standard deviation
* initial policy parameters (0) (theta zero)

Now, for predetermined number of epochs, here is what we do:

We are going to have the inner loop which generates the offspring. Inside this loop we generate some Gaussian noise from the standard normal distribution which represents the offspring. Then, we multiply it by sigma, so that it has a desired standard deviation and add it to our current weight which gives us . Next, we are going to evaluate function F using the current offspring parameters . Note: this function F can be an actual function like a quadratic or it can be the accuracy of supervised/unsupervised learning model or it can be a reward from some environment after playing one or more episodes. That’s a great thing about this black box optimization method, it’s very flexible, since F can be anything we want to optimize. Net, we update to get using the formula above (in green frame). We add a learning rate times 1 over N times times the sum of all the rewards multiplied by their corresponding noise vectors.

Now, since we have a strategy for implementation we just need to specify initial parameters.

We did 2 experiments with slightly different parameters. Learning rate is always = 0.03 and sigma = 0.1. But some parameters are different:

We tried few agents with different pipe\_gaps and training parameters. As output printed the following graphs showing how reward changes over the iterations:

* pipe\_gap = 125

1. population\_size = 50, num\_iters = 300
2. pipe\_gap = 125, population\_size = 50, num\_iters = 300
3. pipe\_gap = 110, population\_size = 75, num\_iters = 500
4. pipe\_gap = 110, population\_size = 30, num\_iters = 400

Also, since the library used to obtain environment for flappy bird (PLE) doesn’t provide convenient API for controlling reinforcement learning agent, we’ve put all necessary methods in the class of environment and neural network for computational purposes.

Sources used:

<https://pygame-learning-environment.readthedocs.io/en/latest/user/home.html>

<https://pygame-learning-environment.readthedocs.io/en/latest/user/games/flappybird.html>

<https://pygame-learning-environment.readthedocs.io/en/latest/_modules/ple/games/flappybird.html#FlappyBird.getGameState>

<https://github.com/Kulbear/deep-learning-nano-foundation/wiki/ReLU-and-Softmax-Activation-Functions>