УДК 004.896

LEARNING SYSTEM DESIGN FOR GAME APPLICATIONS

Yegoshyna G.A.,Voronoy S.M., Ovdieichuk A.A.

O. S. Popov Odessa National Academy of Telecommunications,
1 Kuznechna St., Odessa, 65029, Ukraine.
yegoshyna@onat.edu.ua, voronoy@onat.edu.ua

ПРОЕКТУВАННЯ СИСТЕМИ НАВЧАННЯ ДЛЯ ІГРОВИХ ДОДАТКІВ

Єгошина Г.А., Вороной С.М., Овдейчук А.А.

Одеська національна академія зв'язку ім. О.С. Попова,
65029, Україна, м. Одеса, вул. Кузнечна, 1.
yegoshyna@onat.edu.ua, voronoy@onat.edu.ua

ПРОЕКТИРОВАНИЕ СИСТЕМЫ ОБУЧЕНИЯ ДЛЯ ИГРОВЫХ ПРИЛОЖЕНИЙ

Егошина А.А., Вороной С.М., Овдейчук А.А.

Одесская национальная академия связи им. А.С. Попова,
65029, Украина, г. Одесса, ул. Кузнечная, 1.
yegoshyna@onat.edu.ua, voronoy@onat.edu.ua

Abstract. The presented paper investigates the problem of designing a learning system for agents in intelligent game applications based on Unity Game Engine and reinforcement machine learning algorithms. Modern trends in the game applications development are characterized by the active using of the concept of an intelligent agent as a behavior model of an active element in various situations with applying various strategies for interactions with other active elements and the environment. In recent years, there have been a significant number of advances in this area, such as DeepMind and the Deep Q learning architecture, the winning of the Go Game Champion with AlphaGo, OpenAI and PPO. Unity developers have implemented a support for machine learning and, in particular, for deep reinforcement learning in order to create a deep reinforcement learning the SDK (Software Development Kit) for game and simulation developers. With Unity and ML-Agents toolkits we can create physically, visually, and cognitively rich environments, including ones for evaluating new algorithms and strategies. However, learning system design for agents in Unity ML-Agents is possible only by using the Python API. The possibility of a learning system design for agents in the *Flappy Bird* game application based on the Unity Game Engine with using its own environment is discussed in this paper. Separately, the paper highlights typical features of the *Flappy Bird* gaming application environment. The environment can be implemented as a fully observable environment or a partially observable environment. The fully observable environment is suggested to be used due to all environment states in this case are seen in the playfield. Thus, the problem of strategy formation is considered as a Markov decision-making process and the agent directly observes the current state of the environment. Temporal Difference Learning is used as a learning method; it involves the assessment of a reward at each stage. Two separate environments, deterministic and stochastic, have been implemented, that allows to conduct further research and evaluation of strategy formation algorithms.

Key words: reinforcement learning, game application, Unity Game Engine, *Flappy Bird*, agent, environment, action, strategy, decision making, Temporal Difference Learning, model-free.

Анотація. У статті розглядається задача проектування системи навчання агентів в інтелектуальних ігрових додатках на основі Unity Game Engine і алгоритмів машинного навчання з підкріпленням. Сучасні тенденції розробки ігрових додатків характеризуються активним використанням концепції інтелектуального агента як моделі поведінки активного елементу в різних ситуаціях із застосуванням різноманітних стратегій взаємодії з іншими активними елементами і середовищем. В останні роки можна спостерігати значне число досягнень в цій області, такі як DeepMind and the Deep Q learning architecture, перемога чемпіона гри Go з AlphaGo, OpenAI і PPO. Розробники продуктів Unity впровадили підтримку машинного навчання і, зокрема, глибинного навчання з підкріпленням заради створення SDK, глибинного навчання з підкріпленням для розробників ігор і симуляцій. Використовуючи Unity й інструментарій ML-Agents можна створювати фізично, візуально і когнітивно багаті середовища оточення, в тому числі і для оцінки нових алгоритмів і стратегій. Проте проектування системи навчання агентів в Unity ML-Agents можливо тільки з використанням Python API. У даній статті вивчається можливість проектування системи навчання агентів в ігровому додатку *Flappy Bird* на основі Unity Game Engine з можливістю створення власного середовища оточення. Окремо в роботі виділені особливості, характерні для ігрового середовища *Flappy Bird*. Оточення може бути реалізовано як середовище, що повністю або частково спостерігається. У даній статті пропонується використання повністю спостережуваного оточення, оскільки в цьому випадку всі стани середовища видно на ігровому полі. Таким чином, проблема формування стратегії розглядається як марковський процес прийняття рішень і агент безпосередньо спостерігає за поточним станом навколишнього середовища. Як спосіб навчання був використаний Temporal Difference Learning, що передбачає оцінку винагороди на кожному етапі. Розроблено два окремих середовища оточення, детерміноване і стохастичне, що дозволяє проводити подальші дослідження й оцінки алгоритмів формування стратегій.

Ключові слова: навчання з підкріпленням, ігровий додаток, Unity Game Engine, *Flappy Bird*, агент, оточення, дія, стан, стратегія, прийняття рішень, Temporal Difference Learning, model-free.

Аннотация. В статье рассматривается задача проектирования системы обучения агентов в интеллектуальных игровых приложениях на основе Unity Game Engine и алгоритмов машинного обучения с подкреплением. Современные тенденции разработки игровых приложений характеризуются активным использованием концепции интеллектуального агента в качестве модели поведения активного элемента в различных ситуациях с применением разнообразных стратегий взаимодействия с другими активными элементами и средой. В последние годы можно наблюдать значительное число достижений в этой области, такие как DeepMind and the Deep Q learning architecture, победа чемпиона игры Go с AlphaGo, OpenAI и PPO. Разработчики продуктов Unity внедрили поддержку машинного обучения и, в частности, глубинного обучения с подкреплением ради создания SDK глубинного обучения с подкреплением для разработчиков игр и симуляций. Используя Unity и инструментарий ML-Agents можно создавать физически, визуально и когнитивно богатые среды окружения, в том числе и для оценки новых алгоритмов и стратегий. Однако проектирование системы обучения агентов в Unity ML-Agents возможно только с использованием Python API. В данной статье изучается возможность проектирования системы обучения агентов в игровом приложении *Flappy Bird* на основе Unity Game Engine с возможностью создания собственной среды окружения. Отдельно в работе выделены особенности, характерные для среды игрового приложения *Flappy Bird*. Окружение может быть реализовано как полностью наблюдаемая или частично наблюдаемая среда. В данной статье предлагается использование полностью наблюдаемого окружения, поскольку в этом случае все состояния среды видны на игровом поле. Таким образом, проблема формирования стратегии рассматривается как марковский процесс принятия решений и агент непосредственно наблюдает за текущим состоянием окружающей среды. В качестве способа обучения был использован Temporal Difference Learning, предполагающий оценку вознаграждения на каждом этапе. Разработаны две отдельные среды окружения, детерминированная и стохастическая, позволяющие проводить дальнейшие исследования и оценки алгоритмов формирования стратегий.

Ключевые слова: обучение с подкреплением, игровое приложение, Unity Game Engine, *Flappy Bird*, агент, окружение, действие, состояние, стратегия, принятие решений, Temporal Difference Learning, model-free.

Every year there is more and more news about how artificial intelligence is superior to a person in various gaming competitions. Gaming is changing now because sufficient computational resources are finally available. However, there is a significant difference between artificial intelligence and artificial behavior. In game development it is imperative that agents are as smart as necessary for fun. Agents in games should not outsmart the players. The player's opponent must imitate human behavior but not be perfect. In many cases, reinforcement training was applied. This learning approach became very popular in the recent years. However, using only basic reinforcement learning algorithms is not always sufficient for high-level gameplay. The research in the reinforcement learning field is gaining huge momentum mostly thanks to Google Deepmind [1].

Google Deepmind shows how Deep Learning can be used in conjunction with existing Reinforcement Learning (RL) techniques to play Atari games [2], beat a world-class player in the game of Go, and solve complicated riddles [3].

Autonomous agents can earn and take optimal actions in their assigned environment to achieve certain goal. Agents play a key role in reinforcement learning. Understanding their functioning is very crucial when there is a need to create own learning system or modify existing one. This is a very common problem that covers tasks ranging from controlling a machine by a robot, optimizing production processes, simulation, to learning to play board games and defeat the world champions.

Game applications for decades have provided ideal conditions for training and testing the performance of software agents for systems with artificial intelligence. But to teach an agent to perform certain tasks takes time and computing power, which in turn leads to the choice of the optimal algorithm for conserving resources.

Problem statement. This work describes the design process of the learning system for the *Flappy Bird* game. *Flappy Bird* is a popular mobile game, and it was used as a simulation environment. The main idea of this game is to maximize rewards by overcoming obstacles, which reminds the main concept of the reinforcement learning [4, 5]. *Flappy Bird* is originally intended as a game in which a player or agent can play endlessly. But the main concept remains to maximize rewards by accumulating points for successfully completing obstacles. The task can be realized both with an endless process of passing a level and with a final finish. The agent or player has only two actions available: make a jump or do nothing in this way allowing a bird in the game to fall down according to the laws of physics. The agent must interact with the environment to maximize the reward received. The environment itself is fully observable. It means that the full state is visible on the screen and it is fully accessible to the agent [5].

To train the agent and compare the effectiveness of learning methods in the future, two versions of *Flappy Bird* were created. The first one represents a deterministic environment in which game levels will not undergo changes after restarting the game. The second version represents a stochastic environment that will change after each restart of the game. Using a stochastic and deterministic environment at the same time allows the user to get more useful information about the behavior of the agent. The environment is developed on the Unity Game Engine [6] with C# as the main scripting language.

Reinforcement learning. Reinforcement Learning (RL) is a machine learning paradigm which trains the policy of the agent, so that it can make a sequence of decisions. RL problems involve learning what to do - how to map situations to actions to maximize a numerical reward signal. Essentially, they are closed-loop problems because the learning system’s actions influence its later inputs. Moreover, the learner is not told which actions to take, as in many forms of machine learning, but instead must discover which actions yield the most reward by trying them out. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards [7].

The agent learns to achieve a goal in an uncertain, potentially complex environment. In reinforcement learning, an artificial intelligence faces a game-like situation. The computer employs trial and error to come up with a solution to the problem. To get the machine to do what the programmer wants, the artificial intelligence gets either rewards or penalties for the actions it performs. Its goal is to maximize the total reward [8]. The main points of this process in more detail will now be considered.

The learner and decision-maker are called the agent. Whatever 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 gives rise to rewards, special numerical values that the agent tries to maximize over time. The complete specification of the environment defines a task, one instance of the reinforcement learning problem.

The agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3\dots$. At each time step, the agent receives some representation of the environment's state, $s_t \in S$, where S is a set of possible states, and on that basis the agent selects an action, $a_t \in A(S_t)$, where $A(S_t)$ is a set of actions available in state S_t.

One-time step later, in part as a consequence of its action, the agent receives a numerical reward, $r_{t+1} \in R$, and finds itself in a new state, S_{t+1}.Fig. 1 illustrates the agent-environment interaction [7].

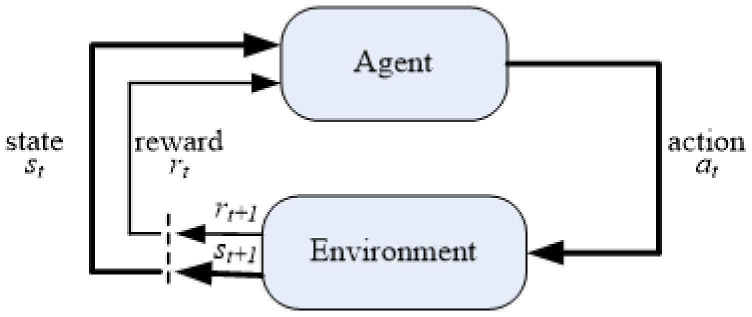

Figure 1 – Agent-environment interaction

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 π_t, where $\pi_t(s, a)$ is a probability that $a_t=a$ if $s_t=s$. 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. This framework is abstract and flexible and can be applied to many different problems in many ways.

In the context of video games, the agent that takes actions or performs a behavior is the game agent. Thinking of a character or a bot in a game, it must understand the state of the game, where the players are, and then, based on this observation, it should make a decision based on the situation of the game. In RL, decisions are driven by rewards, which in a game could be provided as a high score, or a new level for reaching a specific goal.

One of the main elements of reinforcement learning is a model of the environment. This model predicts the behavior of this environment. As an example, an environment can use state and action data and predict the outcome of this choice. As a result, this will be the next state and reward. This approach is used for planning. The planning means any kind of decision making that covers every possible space of future situations before any of this occurs. In reinforcement learning, methods that use planning and models are called model-based algorithms [9]. Model-free are the opposite of model-based methods. These methods explicitly use a trial-and-error approach [10]. Thus, model-free algorithms allow the agent to interact with an unknown environment and make exploration of it.

***Flappy Bird* learning system design.** Each learning system must have a specific environment. Choosing a gaming environment is an economical option in which anything can be simulated without spending more resources than needed.

Learning can be considered as a process that includes improving the performance of the original system relative to some task based on the experience gained. Then, in the developed system, the following key points will be considered as the first steps: choosing task T; choosing training experience E; choosing performance measure P.

The task for the agent is to live as long as possible by avoiding collisions and moving through gaps/spaces between the pipes. The agent should not touch the upper or lower border of the screen, and it is also necessary to avoid collisions with objects in the game.

The position of the agent is in the middle of the X-axis, it has only two possible actions: do nothing or perform a jump (flap). By doing nothing the bird will fall down.

Any problem with the learning agent results in a credit assignment problem. An agent may make optimally correct decisions, but in the end, the result will be unacceptable. Often this may also involve a trade-off between instant reward and long-term reward. This issue concerns indirect learning. Flappy Bird game is more about indirect learning, where learning goes via trial-and-error experience and does not use labeled data.

Whether it is a terminal or non-terminal environment, an agent can collect and use data for its learning. In case of the terminal environment with Q-Learning and SARSA approaches for Flappy Bird, the experience will be stored in the Q-table.

In the designed learning system, each time step number represents a state, and there are two columns for actions. Each action for certain state has a Q-value. When the agent is about to exploit the environment and to pick an action, being at some certain state, it picks an action with the maximum Q. An example of this Q-table is shown in Table 1.

Table 1 – «Q-table data» example

State «t step №»	Action «Do Nothing»	Action «Dump»
0	0	2,839
1	0	9,917
2	-800	40,98

Experience data for Flappy Bird in the non-terminal environment will be different. In this case the environment is not deterministic anymore. An episode can last forever, and objects will be spawning randomly. The simple use of the Q-table is no longer acceptable because the data will be useless. The second approach uses neural networks for training and behavior, and one of the main features which is in use is Replay Memory.

In the designed approach the state consists of some state attributes: horizontal distance between the bird and a next pipe; bird velocity; vertical distance between the bird and a top pipe; vertical distance between the bird and a bottom pipe.

For the neural network, there will be one neuron per each state attribute in the input layer. The replay memory collects the states and rewards for transition each time step. This data can be used for training each time step or it can wait until the end of the episode. In this case training data will have the following view:

Table 2 – «Replay Memory» data samples

State attr. 1	State attr. 2	State attr. 3	State attr. 4	Reward
-3,1195824146	0,9912981986	2,1912982463	1,6961996555	0,01
-3,1784424781	1,0548670291	32,2548670768	1,6466000080	0,01
-3,2373025417	1,1196130514	2,3196130990	1,6466000080	0,01
-3,2961626052	1,1855362653	2,3855363130	1,5968996286	0,01
-3,3550226688	1,2526366710	2,4526367187	1,5968996286	-1

In the case of Flappy Bird , it depends on the task, since the developer can choose the rules in the environment. For example, levels may be fixed, or levels may be randomly generated for each episode. The only dependent factor is which kind of episodes is in the game. They can either have terminal states at the end, or the level can be infinite. The first case is about episodic tasks, the second is associated with continuous tasks. There will be no terminal states in the system being developed, so the task can be considered as continuous. The cumulative reward and the number of iterations will serve as a performance measure.

Reinforcement Learning Algorithms. The next step is choosing a target function in the learning system. After comparing algorithms, it was decided to use the most suitable method for this task – Temporal Difference Learning.

Model-free algorithms estimate the policy or a value function through trial-and-error experience. There are two planning problem types: prediction and control. A prediction problem can be solved with next approaches [7]: Monte-Carlo Learning; Temporal Difference Learning.

Monte Carlo (MC) refers to algorithms without a model [7]. Since the agent is not aware of the environment dynamics, greedy acting would not guarantee 100% success in getting a correct optimal policy. The agent can stick at some point and the goal will be unreachable, or the value of the total reward might have been much bigger.

The Temporal Difference Learning (TD) approach is inspired by Monte Carlo and Dynamic Programming ideas [11]. TD learning can be done via experience of the episodes, like in Monte Carlo. It solves a prediction problem. As in the Dynamic Programming, TD learning uses bootstrapping for updating values. Since this is a model-free algorithm, it requires no model of the environment.

Unlike the previous method, in the Temporal Difference methods, direct learning takes place on incomplete episodes [11]. For example, an agent may be taught after each action that has been performed and received feedback from the environment immediately, rather than waiting for the end of the episode. Algorithms of this kind work well with continuous tasks. But, regardless of this, it is also possible to work with episodic tasks. As a reminder, the episodic tasks do have an end state, whereas continuously tasks episodes can last forever or until the goal is succeeded or failed. TD learning updates the value function each time step and does not wait for the end of the episode. This allows agent to learn immediately without waiting of the outcome, like MC algorithm does. Therefore, TD learning can be applied in non-terminating environments.

TD learning methods planning algorithms are divided by off-policy and on-policy types.

SARSA (state-action-reward-state-action) is an on-policy algorithm., that learns MDP policy. In this algorithm, actions depend on the current policy. This means the implementation of actions at and at+1 will be performed using the current policy π.

Q is a quality function for state and action pairs:

$$Q : S \cdot A \to R. \tag{1}$$

Consider the abbreviation of the algorithm, where each letter (word) corresponds to a parameter of the function for updating the Q-value, which is calculated as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right], \tag{2}$$

where S_t – current state;

A_t – current action;

R_{t+1} – reward for performed action A_t;

S_{t+1} – next state;

A_{t+1} – next action;

α – learning rate;

γ – discount rate.

Every time step includes:

1) policy evaluation with estimating *Q*; and

2) policy improvement with the ε-greedy strategy.

ε-greedy algorithm is used for choosing an action with respect to the exploration-exploitation:

1) initialize ε variable;

2) generate a random value between 0 and 1; that would be $r \in [0,1]$; and

3) if r is lesser than ε, then an action would be random; or

4) else an action will be selected according to the one with the maximum Q with respect to the specific state.

Q-Learning is an off-policy control algorithm, which uses action-value function Q to improve the behavior of the agent. This iterative improvement can be done with use of the Bellman equation. Q stands for the quality as it is working out what the quality of performing a certain action is. While an agent is in a certain state S at time step t, then the quality of performing action in this state is equal to the maximum reward, obtained in the future:

$$Q^{new}(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left(R_{t+1} + \gamma \cdot \max Q(S'_{t+1}, A') - Q(S_t, A_t)\right), \qquad (3)$$

where $Q(S_t, A_t)$ – old value;

R_{t+1} – reward;

$\max Q(S'_{t+1}, A')$ – an estimate of optimal future value.

The Bellman equation states the maximum future reward for a specific state and action. The agent is not aware of the reward until an action is executed during a specific state. Only after, the agent can rely only on historic data. One way to store Q-values is the Q-Table.

In this work, Q-Learning as TD extension and SARSA were used as main methods, which can help with the continuous and episodic tasks. Both methods can be extended to the neural network solution. Also, the agent can learn in an unknown environment after each time step in the episode.

Environment development. In this work, the environment for interacting with the agent is provided by the Unity game engine. In addition, Unity provides the visual development environment, cross-platform support, and a modular component system, Unity also includes useful libraries related to machine learning and intelligent agents.

This game engine was used to create a game application as an environment and for writing the necessary logics for an agent providing a unique solution. The advantages of this approach can also be identified as the ability to control all aspects of the environment and agent's functioning, the ability to expand functionality and make adjustments. Also, experience with this engine and its programming language (C#) has a positive effect on the stages of application development, algorithms implementation, and their evaluation.

The environment for the agent is seen as an interactive game. The main window for the development and testing of the project is presented in Fig. 2.

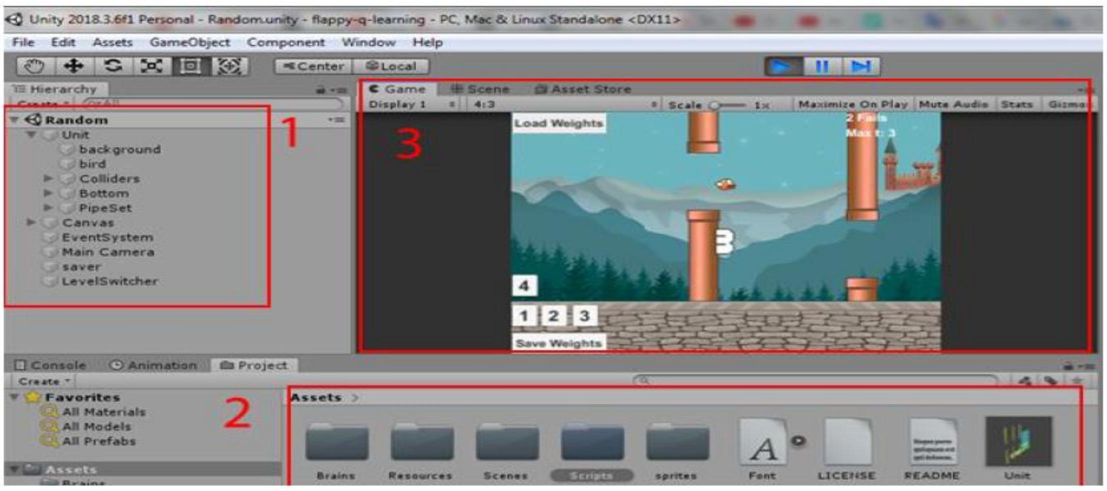

Figure 2 – Main window for developing and testing

Three areas are selected on the figure: main objects of the unity project (hierarchy area); the root folder of the project with all subfolders, scripts, files etc.; and the game screen, where the testing itself is shown.

In terms of the paper two environments were created. The first environment is a deterministic environment with specific rules: obstacles/pipes are fixed and have a limited quantity; the terminal end state is at the end; the tabular version of algorithms Q-Learning and SARSA is used (it means that Q-Learning and SARSA store Q-values in the Q-Table); different design and implementation.

The second environment is a stochastic environment with the following differences: it contains four levels with different difficulty; objects/pipes have sequential order and they looped (this is scene #1 with some deterministic conditions, but it can last forever until the agent has failed); it contains scene #2 with the objects/pipes that appear randomly (the agent can perform the task in maximizing a reward continuously, unless failure).

The following rules are defined in deterministic environment:

– the bird is stationary on the x-axis, but actions to jump or do nothing can change the position on the x-axis;

– in the game there are obstacles/pipes that move/scroll to the left side along the x-axis;

– the agent’s goal is to go through the gaps in between in pipe pairs until the terminal state is reached;

– a certain amount of the reward that is added at each step is accumulated;

– in addition to moving obstacles, a bird/agent cannot touch the upper or lower boundaries; and

if the agent/bird hits any obstacle, then the game will be lost and completely restarted.

The hierarchy of objects is as follows: Canvas – contains UI game object elements (it contains a camera for displaying the game screen); Score – UI element, uses TextMeshPro component for displaying a game score; Top – represents the top border with collisions; Main Camera – displays the game screen; Bird – represents a bird (controlled by the agent), contains a bird and agent scripts (Bird.cs, Agent.cs), and has a tag «Player»; Scrolling – contains objects, which represent a bottom border collision, bottom sprites and a background, contains attached scripts (ScrollingObject.cs), is responsible for managing these contained objects; Ground object, which represents a bottom border and has a tag «Ground»; Cols – this object contains a pipe/obstacle pairs, each pair has a fixed position and an attached script (Columns.cs), one pair of the pipes is labeled as the terminal end state, each pair has a tag «pipe».

Stochastic environment was developed as a separate application so that the projects would not be mixed. The environment is divided into two scenes. Scene «Looping Scene» consists of fixed levels. The terminal state is not considered since the levels were designed so that after passing a specific object, they change their positions in order to loop. This approach was developed to train the agent gradually with increasing complexity. Scene «Random Scene» consists of obstacles that spawn on random positions. There are four pipe/obstacle pairs where each pair consists of one pipe at the top and one pipe at the bottom. The main logic is that each time an agent passes one of them (pipe/obstacle pairs), this object will be generated at the end with a random position in each range. Each of the two scenes has characteristic equally main object. Unit – represents a collection of objects. One of the most important is the bird object which is basically an agent. Other is «PipeSet» which represents the moving obstacles. To implement moving objects, two scripts were written: Pipeset.cs, Pipes.cs.

Pipeset.cs class responds for the set of pipes (obstacles). On the scene, there two pipe objects with a gap between. This is considered as an obstacle, and the pipeset includes many of these object pairs. The agent’s goal is to find this gap and go through it.

Pipes.cs class represents each pair of the pipe obstacles. This component is attached to each object, which is a set of these obstacles. Methods are responsible for randomizing obstacle positions, and they also are capable to assign their initial positions and responsible for the movement.

The main logic of the environment is that the agent's object itself does not move along the x-axis, while obstacles move along the negative abscissa. The ordinate axis is allocated to the agent. If the action does not occur, then, according to the laws of physics, the agent's object moves down. But if the agent decides to take a jump, then one will rise with force up. Thus, one of the tasks is to balance in the given environment, and the other is to get into the passable area. All this can be reduced to the goal of not hitting any of the obstacles and finding the most optimal actions for this, based on experience.

Some of the most successful examples of reinforcement learning are in the field of developing agents for games. Agent and environment are core concepts of reinforcement learning. The agent interacts to maximize the cumulative reward. The agent has a policy that represents one's behavior or decision strategy. The environment can be implemented as a fully observable environment or a partially observable environment. In this report, a fully observable environment was selected since in this case all the complete states of this environment are visible on screen and everything is under control of the agent. Environments of this type use Markov decision processes. The main problem is to find an optimal policy. Finding optimal policy is basically the planning problem and consists of prediction and control.

In this paper, model-free methods are found to be more suitable for the *Flappy Bird* problem due to unknown environments. To solve the planning problem, Temporal Difference Learning methods were selected since TD methods are able to perform in the terminal and non-terminal environments. Two separate environments were created for further research and evaluation of algorithms. As future work, Deep Q Learning and Deep SARSA algorithms should be implemented.

REFERENCES:

1. Pavel Kordík. "Reinforcement Learning: Artificial Intelligence in Game Playing". <https://pavelkordik.medium.com/reinforcement-learning-the-hardest-part-of-machine-learning-b667 a22 995 ca> accessed Dec 7, 2020.
2. Volodymyr Mnih, David Silver, Koray Kavukcuoglu, Alex Graves. "Playing Atari with Deep Reinforcement Learning" <https://www.researchgate.net /publication /259367763_ Playing_ Atari_ with_ Deep_Reinforcement_Learning> accessed Dec 7, 2020.
3. Debidatta Dwibedi, Anirudh Vemula. "Playing Games with Deep Reinforcement Learning". <https://debidatta.github.io/assets/10701_final.pdf> accessed Dec 2, 2020.
4. Survy Vaish. "*Flappy Bird* hack using Reinforcement Learning". <http://sarvagyavaish.github.io/FlappyBirdRL/> accessed Dec 2, 2020.
5. Louis-Samuel Pilcer, Antoine Hoorelbeke, Antoine d'Andigne. "Playing *Flappy Bird* with Deep Reinforcement Learning". <https://www.researchgate.net /publication /324066514_ Playing_Flappy_Bird_ with_Deep_Reinforcement_Learning> accessed Dec 2, 2020.
6. Unity Scripting Tools IDEs <https://docs.unity3d.com/Manual/ScriptingToolsIDEs.html> accessed Dec 11, 2020.
7. Sutton, Richard S. and Andrew G. Barto Reinforcement Learning: An Introduction. 2nd ed. Cambridge, MA: The MIT Press, 2015, p. 352. Print.
8. Błażej Osiński, Konrad Budek. "What is reinforcement learning? The complete guide". <https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/ > accessed Dec 2, 2020.
9. Michael Janner. "Model-Based Reinforcement Learning: Theory and Practice". <https://bair.berkeley.edu/blog/2019/12/12/mbpo/ > accessed Dec 2, 2020.
10. Ryan Wong. "Model-Free Prediction: Reinforcement Learning". <https://medium.com/@taggatle> accessed Dec 2, 2020.

11. Dan Lee. “Reinforcement Learning, Part 5: Monte-Carlo and Temporal-Difference Learning”. <https://medium.com/ai%C2%B3-theory-practice-business/reinforcement-learning-part-5-monte-carlo-and-temporal-difference-learning-889053aba07d> accessed Dec 16, 2020.

ЛІТЕРАТУРА:

1. Kordík P. Reinforcement Learning: Artificial Intelligence in Game Playing [Електронний ресурс] / Pavel Kordík. – 2016. – Режим доступу до ресурсу: https://pavelkordik.medium.com/reinforcement-learning-the-hardest-part-of-machine-learning-b667a22995ca (дата звернення 07.12.2020).
2. Playing Atari with Deep Reinforcement Learning [Електронний ресурс] / V.Mnih, D. Silver, K. Kavukcuoglu, A. Graves. – 2013. – Режим доступу до ресурсу: https://www.researchgate.net /publication /259367763_ Playing_ Atari_ with_ Deep_Reinforcement_Learning (дата звернення 07.12.2020).
3. Dwibedi D. Playing Games with Deep Reinforcement Learning [Електронний ресурс] / D. Dwibedi, A. Vemula. – 2016. – Режим доступу до ресурсу: https://debidatta.github.io/assets/10701_final.pdf (дата звернення 02.12.2020).
4. Vaish S. *Flappy Bird* hack using Reinforcement Learning [Електронний ресурс] / Survy Vaish. – 2014. – Режим доступу до ресурсу: http://sarvagyavaish.github.io/FlappyBirdRL (дата звернення 02.12.2020).
5. Pilcer L. Playing *Flappy Bird* with Deep Reinforcement Learning [Електронний ресурс] / L. Pilcer, A. Hoorelbeke, A. d’Andigne. – 2018. – Режим доступу до ресурсу: https://www.researchgate.net/publication/324066514_Playing_Flappy_Bird_with_Deep_Reinforcement_Learning (дата звернення 02.12.2020).
6. Unity Scripting Tools IDEs [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com/Manual/ScriptingToolsIDEs.html (дата звернення 11.12.2020).
7. Sutton R. S. Reinforcement Learning: An Introduction. 2nd ed / R. S. Sutton, A. G. Barto. – Cambridge, MA: The MIT Press, 2015. – 352 с.
8. Osiński B. What is reinforcement learning? The complete guide [Електронний ресурс] / B. Osiński, K. Budek. – 2018. – Режим доступу до ресурсу: https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/ (дата звернення 02.12.2020).
9. Janner M. Model-Based Reinforcement Learning: Theory and Practice [Електронний ресурс] / Michael Janner. – 2019. – Режим доступу до ресурсу: https://bair.berkeley.edu/blog/2019/12/12/mbpo/ (дата звернення 02.12.2020).
10. Wong R. Model-Free Prediction: Reinforcement Learning [Електронний ресурс] / Ryan Wong. – 2019. – Режим доступу до ресурсу: https://medium.com/@taggatle (дата звернення 02.12.2020).
11. Lee D. Reinforcement Learning, Part 5: Monte-Carlo and Temporal-Difference Learning [Електронний ресурс] / Dan Lee. – 2019. – Режим доступу до ресурсу: https://medium.com/ai%C2%B3-theory-practice-business/reinforcement-learning-part-5-monte-carlo-and-temporal-difference-learning-889053aba07d (дата звернення 16.12.2020).

DOI 10.33243/2518-7139-2020-1-2-82-91