Hybrid Rule-based QoS-Optimized Modification of Deep Q-Learning Network: Properties and Specifics

Irina Grishanova[1,†], Julia Rogushina[1,2,*,†], Pylyp Andon[1,†]

[1] *Institute of Software Systems, National Academy of Sciences of Ukraine, 44 Glushkov Pr., Kyiv, 03680, Ukraine,*
[2] *Institute for Digitalisation of Education, National Academy of Educational Sciences of Ukraine, 9 M. Berlynskoho Str., Kyiv, 04060, Ukraine*

Abstract

The study analyzes the capabilities of reinforcement learning methods and their components. We propose the hybrid rule-based QoS-optimized modification of Deep Q-learning network method developed to facilitate automatic determination of key learning parameters and incorporate mechanisms for continuos learning. This method enables the agent to adapt to the environment dynamics, such as the availability of possible actions and changes of their parameters, without requiring complete relearning and ensure flexibility, universality and enhanced efficiency of Machine Learning method across various domains. We test the proposed algorithm and its characteristics on the practical task of the personalized learning pathway constructing used in educational process.

Keywords
Learning, Deep Q-learning Network, Personal Learning Pathway, multigoal planning

1. Introduction

Reinforcement Learning (RL) is a machine learning approach where an agent interacts with an environment and learns on base of rewards and penalties [1]. RL is used to solve optimization and decision-making tasks in complex dynamic environments. The fundamental components of RL are: agents, the environment, and interactions.

An *agent* is an autonomous entity that makes learning-based decisions and adapts its behavior according to the feedback between its actions and changes in the environment state. The agent state represents its internal belief about environment, and this environment can be partially or fully observable. The agent's actions policy π is the strategy that agent uses to select next actions based on its current state. An agent has the task that is defined by initial state and goal state.

The *environment* $E =< S, A, P, R, s_{init}, S_{target} >$ is the space of the agent operations, where: S is a *space of states* that the agent can observe; A is a *space of actions* defined as a set of all possible actions the agent can take; a *transition function* P determines the new state of the agent after performing action a; a *reward function* R evaluates the agent's actions; an *initial state* s_{init} marks the starting point; *target states* S_{target} define the conditions for completing the learning process. If the agent reaches a target state $s_{goal} \in S_{target}$, the episode ends, and the agent receives a reward.

The *state space* of the environment $S = \{s_q =< x_1,...,x_n >\}, q = \overline{1,2^n}$ is the set of all possible state configurations that the agent can recognize. A *state* $s =< x_1,...,x_n >, s \in S$ is defined as an n-dimensional vector.

It is important to distinguish between the state of the environment and the state of the agent. Every state of the environment $s \in S$ is a combination of available input and output parameters that vary

Workshop "Intelligent information technologies" UkrProg-IIT`2025 co-located with 15th International Scientific and Practical Programming Conference UkrPROG'2025, May 13-14, 2025, Kyiv, Ukraine.
* Corresponding author.
i26031966@gmail.com (I. Grishanova); ladamandraka2010@gmail.com (J.Rogushina); andon@isofts.kiev.ua (P.Andon)
0000-0003-4999-6294 (I. Grishanova); 0000-0001-7958-2557 (J.Rogushina); 0009-0000-1737-5579 (P. Andon)

depending on the actions performed by the agent. The state of the agent $s_a \in S$ is an internal set of parameters of the learning model, that can include knowledge of prior experiences (e.g., an action-value table $Q(s, a)$or neural network parameters), the current level of environment exploration (epsilon-greedy value), knowledge about executed transitions, etc. In further explanations, we consider s as an environment state, because it serves as the input data for the learning algorithm.

In this work we introduce additional terms to clarify the proposed modification of learning algorithm.

State signals $x_i \in \{0,1\}, i = \overline{1,n}$ are state attributes that characterize the state of the environment: $x_i = 1$ if the agent has information about this environmental parameter; $x_i = 0$ if the agent has no information about this parameter of the environment.

Terminal states $S_{term} \subset S$ are a subset of states that conclude training episodes. They are divided into several non-overlapping subsets $S_{target} \cup S_{deadlock} \cup S_{limit} = S_{term}$.

Target states $S_{target} \subseteq S_{term}$are reperesented by a set of states where all signal values defined as the agent learning goal are equal to 1 (other signal values can be 1 or 0). The goal state $s_{goal} \in S_{target}$ needs to meet the condition: for all elements $s_{target} =< x_{target_1},...,x_{target_n} > \in S_{target}$ and $s_{goal} =< x_{goal_1},...,x_{goal_n} >$, if $x_{goal_k} = 1$, then $x_{target_k} = 1, k = \overline{1,n}$

Deadlock states $S_{deadlock} \in S_{term}$ are a set of states where the agent cannot perform any action to expand its knowledge. These states can arise if the agent has no available services to execute or if all possible actions don't change the state or repeat actions that have already been performed.

Limit states $S_{limit} \subseteq S_{term}$ are a set of states that the agent reaches within a limited number of steps but fails to complete the assigned task.

Action space A is the set of all possible actions that agent can perform within the environment. The agent *actions* can be represented as services invoked by the agent, where each service is characterized by input parameters, output parameters, a description of their transformation, and quality parameters (QoS) that define this transformation. *Service* $w \in W$ is a specific entity that, in the presence of the appropriate input parameters, changes the state of the environment and determines the values of the output parameters (either modifies or computes them) [2].

Flow is a sequence of actions, where each action is permissible in a given current state, starting from the initial state. These actions change the environment from one state to another and complete when the environment reaches a target state.

The agent finds a set of flows $M = \{m_j\}, j = \overline{1,p}$ that ensure the transition from the initial state s_{init} to the target state $s_{goal}$and selects the optimal one based on the QoS parameters of the services within the flow: $f(m_{opt}, QoS(m_{opt})) \geq f(m_j, QoS(m_j)), j = \overline{1,p}$.

Composite service $w_{compoz} =< w_1,..., w_r >$ is an ordered sequence of services $w_k \in W, k = \overline{1,r}$ such that after execution of each service the set of environmental signals becomes suitable for executing the next service. After completing all services, the state of the environment is a subset of the terminal state $S_{target} \subseteq S_{term}$.

$w_{compoz} : s_{init} \rightarrow s_{goal}$

To find the optimal composite service that satisfies the given initial and target states, it is necessary to define the optimal conditions for QoS parameters, i.e., their minimization or maximization.

The mechanism for updating the agent knowledge about the environment depends on the method used for ML. Q-learning method [3, 4] represents this mechanism by updating the Q-table according to the Bellman equation. In DQN (Deep Q-learning) [5, 6], it involves updating the parameters of a neural network that approximates the Q-function $Q(s, a)$ through gradient descent, utilizing a Replay Buffer and a Target Network (this approach enables the agent to handle large or continuous state spaces).

Transition Function $T(s,a,s`)$ determines the agent state $s`$ after performing action a from state s. Depending on the transition function, environments are categorized as either deterministic or stochastic.

Reward Function $R(s,a,s`)$ evaluates the agent's actions.

Interaction is the process used by the agent for the environment discovering: by choosing an action (based on its ε-greedy policy), the agent receives feedback (reward R — positive or negative) and updates its knowledge according to the ML method (e.g., Q-learning or DQN) or passes to a new state. The ε-greedy policy is used to maintain a balance between exploration and exploitation.

The RL components form a closed loop of interactions between the agent and the environment: the agent selects an action → the environment responds by changing its state and providing a reward → the agent updates its strategy based on the experience gained using an optimization algorithm, such as Q-value table updates or gradient descent. This cycle repeats until the agent learns for optimal performing the task or achieves its goal in the environment. The agent works with the environment states, perceiving them as input data for decision-making. At the same time, the ML process itself is guided by the agent internal parameters that are change with each interaction.

Additionally, most RL methods (except for classical Q-learning) use optimization algorithms that enhance learning stability and accelerate convergence. DQN uses a Replay Buffer to reduce correlations between consecutive steps, while parameter updates are stabilized through the use of a Target Network.

2. Deep Q-Learning network method

DRL is a group of ML methods that combine Deep Learning and Reinforcement Learning [1, 7]. These methods avoid the need to store a large Q-table (as in traditional Q-learning) by utilizing neural networks to approximate the Q-value function. Instead of explicit table for every state and action (that becomes computationally inefficient for large state spaces), DQN uses a neural network that takes a state as input and predicts Q-values for all possible actions. This approach enables the modeling and processing of continuous or very large state spaces where classical Q-tables are impractical. DQN is a modification of Q-learning that employs a deep neural network for approximating the Q-function. Examples of DRL methods are DDPG (Deep Deterministic Policy Gradient), A2C/A3C (Advantage Actor-Critic), PPO (Proximal Policy Optimization), and DQN (Deep Q-Network).

The fundamental components of DQN are the Policy Q-Network, the Target Q-Network, the Replay Buffer, and the action selection mechanism (ε-greedy strategy).

Q-value update function $Q(s,a)$ (it is an analogue of the Bellman equation for Q-learning: $Q(s,a) = \alpha[r + \gamma \max_{a`} Q(s`,a`) - Q(s,a)]$, where $\alpha$ is the learning rate, $\gamma$ is the discount factor, r is the reward, and $s`, a`$ are the next state and action) evaluates the expected total reward for selecting action a in state s, considering future steps.

Unlike Q-learning, where the function values are stored in a table, in DQN the function is approximated by a neural network $Q(s,a;\theta)$ that takes the current state s (as a vector of attributes) as input and transforms it into Q-values for each possible action a based on the network parameters θ.

Replay Buffer is a specialized memory space where past interactions between the agent and the environment are stored (as opposed to classical Q-learning that updates Q-values on base of current experience).

Target Network $Q`$ is a separate copy of the policy network that is updated more slowly (using older parameters $\theta-$) to avoid instability during training. The target values $Q`(s`,a`;\theta)$ are computed as the sum of the reward r for action a in state s, and the highest Q-value for the future state $s`$, discounted by the factor $\gamma$. The target value Y for the loss function is calculated as follows: $Y = r + \gamma \max_{a`} Q(s`,a`;\theta^{-})$. This approach helps to reduce training instability, smoothes Q-value updates, and prevents abrupt changes in the agent policy.

ε-greedy policy defines how the agent selects actions (similar to classical Q-learning): it selects a random action (exploration) with a probability of ε and it selects the best action based on Q-values

(exploitation) with a probability of $1 - \varepsilon$. The ε -decay gradually decreases, that allow the agent to focus more on exploration at the beginning and optimize its behavior later.

Loss function in DQN learning is based on gradient descent, where the loss function (typically the Mean Squared Error – MSE between the predicted Q-values and the target values from the Bellman equation) is minimized**.**

Neural network update uses an optimizer (e.g., Adam, RMSprop) for updating weights of the model. It utilizes a deep learning process where the neural network gradually improves its estimates of the Q-values.

DQN hyperparameters that influence its efficiency include:

- *Discount factor* γ determines the importance of future rewards;
- *Learning rate* α defines the speed at that the neural network updates its parameters;
- *Exploration rate* ε indicates how often the agent explores random actions;
- *Size of Replay Buffer* specifies how many interactions the agent stores in memory;
- *Batch size* is the number of selected samples for training;
- *Target Network update frequency* determines how often the target network is synchronized.

Let us consider the Replay Buffer in more detail. It is a key mechanism in DQN that addresses the problem of correlated data and stabilizes learning. The Replay Buffer enables the agent to effectively utilize accumulated experiences, reducing randomness in Q-network updates and accelerating algorithm convergence. It is a data structure that stores the agent's experiences during interactions with the environment as a set of tuples $B = \{(s, a, r, s`)\}$, where:

- s is the input state of the environment before the agent performs action a;
- a is the action chosen by the agent in state s;
- r is the reward received by the agent for performing the action a in the state s;
- s` is the state of the environment after the agent performs action a in the state s.

Experience in the form of transactions $(s, a, r, s`)$ is stored in memory and selectively used to update the model, that improves training stability. At each step, DQN randomly selects a mini-batch from the memory B for training the policy neural network. The main advantages of the Replay Buffer include the ability to use past data for learning (to reduce correlation between consecutive states), that enables efficient use of experience by selecting the best samples.

The most common types of Replay Buffer are Fixed-Size Replay Buffer (FIFO – First In, First Out) and Prioritized Experience Replay Buffer (PER). The classical implementation of DQN involves using the FIFO type, that is recommended for the slowly changed environment and if the preserving recent experiences is important. FIFO is advised for initial usage due to its simplicity of implementation. PER is recommended for rapid learning, and if the agent needs to focus on key states. We have implemented the capability to use these two types, allowing for the determination of the optimal one for a specific case depending on the task environment.

3. The DQN neural network architecture

We consider a single-layer neural network for DQN, implemented in the DQNAgent class and built on the basis of a Multilayer Perceptron (MLP). This neural network takes an encoded state vector as input and outputs Q-values for all possible actions. Its architecture consists of input, hidden and output layers.

Input layer accepts the state, that is represented as a vector of attributes with a length corresponding to the number of possible state signals in the environment. The layer transforms this input into a 64-dimensional feature vector. This state is encoded using the one-hot method, where each bit indicates whether the corresponding input signal is present in the current state.

The *hidden layer* consists of 64 neurons and uses the ReLU activation function to introduce nonlinearity.

The *output layer* provides the Q-values for each possible action. It transforms the 64 neurons into action size, that corresponds to the number of possible actions (available services). The output layer does not use an activation function, as the output values are interpreted as Q-values for each action.

The *Adam-optimizer* is used for optimization, serving to update the weights. The standard Mean Squared Error (MSE) is employed as the loss function to minimize the squared differences between the predicted and target Q-values.

For updating the weights, we use the ε-greedy strategy to balance between exploration and exploitation. The formula for ε calculation is not predetermined but is automatically determined using an optimizer, that evaluates the effectiveness of various approaches: linear, exponential, or adaptive decay, and selects the most efficient one for the given environment.

Target Network is implemented as a separate copy of the main neural network, that is periodically updated to stabilize learning. The update frequency is automatically determined on base of the environment parameters and the task specifics.

This simple neural network is sufficiently effective for solving the task of service composition. For more complex tasks, the number of hidden layers can be increased to improve Q-value approximation. This architecture enables the agent to learn optimal actions by predicting long-term rewards for each possible choice, effectively approximating the Q-function values for RL tasks.

4. RL use for construction the optimal personalized learning pathway

The application of RL methods and the choice of their modifications significantly depend on the specifics of the tasks they are used for and the rules of the corresponding domains. Let us consider the task of construction the *Personal Learning Pathway* (PLP) that defines a personalized sequence for *Learning Objects* (LO) [8] studying (this problem is defined in details in [9]), that a student needs to acquire a set of competencies for some *Learning Course* (LC). Each LO is considered as a service [10], where inputs are the competencies required for the student to perceive the information from this LO, outputs are the competencies the student acquires as a result of studying this LO, and QoS defines the quality characteristics of the LO, such as study time, cost and rating. PLP is constructed as a composite service [11].

The state of the agent $s =< x_1,..., x_n >, s \in S$ indicates whether the student has acquired specific LC competencies, i.e., $x_i = 1$ if the student has the corresponding competency. The initial state $s_{init} \in S$ represents the set of competencies that a particular student has before starting the LC study. Achieving the target state $s_{goal} \in S$ means acquiring the whole set of LC competencies.

The task is to construct a set of composite services that propose pathways to transition the agent from the initial state to the target state from the subset S_{target}. Subsequently, the optimal pathway should be selected according to the task specifics and the needs of the particular student (e.g., minimizing studying time or maximizing the ratings of the studied LOs).

The features of this task are:

- A very large number of LOs with outputs contained elements of s_{goal} is available;
- A student who acquired a specific competency do not need to study this competency again. Thus, pathway constructing has to prioritize LO services that do not duplicate each other's outputs;
- Each student begins LC study with an individual set of competencies, that also do not require repeated study;
- An availability of LO services, their inputs, outputs, and QoS characteristics can be changed during the study process;
- A set of competencies used as inputs and outputs for LO services is fixed before the start of the learning process, and this set is relatively small.

According to ML terminology, we work with a dynamic environment (the properties of LO services can be changed), and this environment has a large number of states (all possible configurations of available characteristics of LO services).

Previously, we examined the solution to the PLP construction task based on the Q-learning method and identified the need for significant modifications of this algorithm according to the task requirements [9]. Overall, the obtained solutions are satisfactory, but we identify certain issues that are not inherent to DQN.

Classic Q-learning uses Q(s,a) table, that stores values for each state s and action a. However, for PLP construction task in real-world conditions, such table becomes extremely large, making its use impractical. DQN addresses this issue by replacing the Q-value table with a neural network that approximates the Q(s,a) function. Thus, we decide to explore the feasibility of applying DQN to this practical task and determine specific modifications to this algorithm that are necessary for the efficient PLP construction.

Additionally, during the study of the DQN method operation, we encountered the need to set certain rules (as in the previous Q-learning modification), that significantly improve the results (speed and quality). Moreover, the task requires the inclusion of QoS in the reward calculation — taking into account both their values and their semantics. These features led to the necessity of DQN modifying to meet the task requirements. Proposed modificated DQN can be applied for other tasks from various domains with simular requirements.

5. DQN modification

In previous studies, we proposed a modification of the Q-learning method aimed on the PLP constructing as a composite service. However, despite these improvements, Q-learning has several fundamental limitations related to scalability and the handling of large dynamic structures that constrain its effectiveness in complex environments with a high number of states and actions, where policy generalization is required.

Thus, we propose to use DQN method, that provides better generalizability, enhanced efficiency in dynamic environments, and stabilization of the learning process through advanced mechanisms such as the experience *Replay buffer* [12] and the *Target neural network* based on the neural network approximated the Q-function.

DQN remove the following limitations of the classical Q-learning:

- Eliminates dependency on the Q-table size, as the neural network approximates the Q-value function, reducing memory requirements and enabling work with a large space of states;
- Generalizes knowledge for similar states, ensuring adaptation to new conditions;
- Works more effectively in complex dynamic environments due to the ability of the model to learn regularities. However, DQN still operates with a discrete set of actions, so it requires additional modifications or alternative methods for continuous action spaces;
- Provides more efficient utilization of experience through Replay Buffer, that stores and reuses transitions for stabilizing the learning process;
- Enhances the stability of learning through the use of Target Network;
- Optimizes learning process by using random sequences from Replay Buffer, improving efficiency and the uniformity of model updates.

DQN method proposes important advantages compared to classical Q-learning, but it has certain drawbacks, including issues with stability, scalability, adaptability and efficiency in complex and specific environments. To address these, we propose the following modifications:

- incorporating action constraints and implementing mechanisms to detect deadlock states and loops. The implementation of such rules helps to avoid undesirable situations and improves decision-making efficiency;

- more accurate state space representation, contributing to better alignment with the modeled environment because the lack of structured state space modeling in DQN can lead to suboptimal solutions;
- automated selection of hyperparameters instead of manually tuning parameters such as learning rate or discount factor to optimize model productivity;
- an adaptive reward function, enabled through the normalization of QoS metrics, the use of global weights to account for the importance of each parameter and the ability to customize the optimality criteria (maximization or minimization) improves model flexibility and efficiency, stabilizes the process and enhances learning outcomes;
- use of alternative state transition modes adds an accumulative mode, that allows for retaining and expanding knowledge to traditional DQN mode where the agent fully updates its knowledge during each transition. Providing the ability to switch between these modes ensures the algorithm adaptability to tasks with varying requirements and improves training stability;
- automated determination of key parameters such as buffer type, memory size, number of episodes, number steps without progress, etc., and ε-function ensuring better adaptation to diverse environments;
- supporting different learning modes by adding modes with fixed and random initial states makes the learning process more universal and effective across a wide range of environments;
- implementing Multi-Goal Learning enables efficient operation in environments with complex performance metrics (base DQN does not support learning with multiple objectives);
- providing retraining during exploitation allows adaptation to changes in the environment.

When the agent selects an action, we propose to apply task-specific *optimization rules* that improve the efficiency and stability of learning:

- *Limitation steps without progress.* If selected available action does not lead to progress, a counter for non-progressive steps increases. In the accumulating mode, progress is defined as adding new signals to the agent's state. In the non-accumulating mode, progress is defined as the agent transitioning to a new state different from the previous one. If the counter exceeds a set threshold, the agent terminates the episode or changes its strategy;
- *Check the action benefit.* If the available action does not provide new information, it is discarded, and an other action is selected. In the accumulating mode, an action is considered useful if it adds new signals (expands the agent's knowledge). In the non-accumulating mode, an action is considered useful if it changes the agent's state;
- *Avoid repeated actions.* The agent avoids repeating actions already performed within the current episode. Unlike standard DQN, where the agent can repeat the same actions without restrictions, in this case, such actions are deemed unnecessary. They are discarded, and an alternative action is selected. This optimization reduces the number of operations and speeds up the learning process;
- *Handling deadlock states.* If agent has not any available action, it performs forced exploration by selecting a random action. If this situation repeats and the non-progressive step counter exceeds the threshold, the episode ends with a penalty;
- *Check the action feasibility.* The agent analyzes the availability of each action before executing. If the environment is changed, then certain services become unavailable or their input parameters are changed. In such cases, the agent searches for an alternative action.

We use two learning modes: one with a *fixed initial state* and another with a *random initial state*. In the fixed initial state mode, the agent starts learning with the predefined set of initial and target states. The main goal of this approach is to achieve a stable solution for a specific scenario. A fixed state allows for quicker evaluation of the model efficiency and provides the ability to verify policy convergence. However, this mode has limitations, as the model can become overly specific to one scenario and lack generalized for other conditions.

The *random initial state* mode enhances the adaptability of the model, enabling the agent to work with various sets of initial states. This approach ensures policy generalization across a wide range of environment variations. However, due to the variability of rewards and states, the learning curve can be unstable, complicating the assessment of the agent's progress. Nevertheless, this mode allows the agent to operate under changing environment parameters.

6. Tuning parameters and the learning strategy function of the Modified DQN

The efficiency of the modified DQN algorithm relies heavily on learning parameters fine-tuning. Manual selection of these parameters is often labor-intensive and can cause non-optimal results, especially in complex environments with a large number of states and actions. Automating this process significantly simplifies the task and ensures more accurate value selection. Dynamic parameter adjustment during learning process allows the algorithm to adapt to changes in the environment, improving its effectiveness.

We propose to automate the determining the required number of episodes based on the number of available actions in the environment. In addition to optimizing the main DQN hyperparameters, such as learning rate, discount factor, and selecting the strategy for calculating the ε -function (linear, exponential, or adaptive) with predefined parameters (initial value, minimum value, decay rate), our modification requires fine-tuning the following parameters:

- *Type of Replay Buffer* (FIFO or Prioritized) for optimal memory and experience management;
- *Batch size* – the number of data samples processed by the agent simultaneously during a single learning step;
- *Replay Buffer Size* – the capacity of the *Replay Buffer* to store experiences, i.e., the number of records the agent can hold to accumulate transitions;
- *Maximum number of steps without progress* to set a threshold to detect looping or deadlock situations;
- *Target network update frequency* to determine intervals to stabilize the learning process and reduce correlation between updates.

Automated tuning of these parameters enhances the model efficiency and adaptability to complex environments. To automate parameter selection, we use the Optuna library, that enables efficient parameter search in the space of parameters based on Bayesian optimization methods. The stage of searching for optimal parameters is carried out in the mode with random initial states.

7. Agent Learning and Exploitation Algorithm

Algorithm of the agent learning and exploitation consists of three main stages.

Stage 1. Optimization of parameters using Optuna, where Bayesian optimization is employed for efficient parameter searching. Based on the input data (spaces of the environment states and actions, a set of hyperparameters for searching with respective ranges or value sets, and ε -function strategies), the search function returns the agent average reward over the last episodes and determines the best parameter set for the agent. Multiple workers can be used for parallel acceleration. Here we use random initial state mode.

Stage 2. Learning the agent with a random initial state using the optimal parameters obtained in Stage 1. Upon completion of the learning process, the agent model is saved in a format that includes: hyperparameters and settings for exploitation, model and target network weights and memory buffer that contains transition experiences.

Stage 3. Exploitation with *Fine-Tuning* mode. After loading the saved agent model, the following steps are performed:

- Initialization of input data set the initial and target states;

- Learning with Fine-Tuning mode where the agent adapts its policy to new environmental conditions through weight optimization;
- Optimal flow search that is performed using parameter tuning to improve the flows found by the agent.

This approach allows for effective agent learning in a large environment, ensuring model adaptability and policy generalization. The use of automated parameter search enables optimization not only of hyperparameters but also of strategies for balancing exploration and exploitation. As a result, the agent achieves high accuracy during exploitation mode.

8. Hybrid Rule-based QoS-Optimized DQN properties

DQN modified according to the elements described above differs from the classical approach and expands its capabilities. The proposed *Hybrid Rule-based QoS-Optimized DQN* (HR-QoS-DQN) method combines a rule-based action selection approach with Quality of Service (QoS) optimization. The algorithm not only selects actions with the highest Q-values but also integrates additional QoS parameters and a set of rules to construct a PLP, where the final state is represented as a flexible set of knowledge. This enables the agent to effectively adapt its policy to a dynamic environment, ensuring high decision-making stability.

1. *The universality of the knowledge representation format.* The vector of binary signals, that describes the environment state, can easily be customized for various types of tasks and environments, including environments with high semantic complexity. This format ensures scalability, as it allows the addition of new signal features without altering the core structure of the model, facilitating its application in large-scale systems. Unlike classical methods focused on coordinates or gradients, the proposed in HR-QoS-DQN format minimizes reliance on complex specific data such as images or state vectors, ensuring universality for diverse types of data. For example, in the task of PLP construction, the state is represented as a binary value vector, where each feature indicates the presence (value is equal to 1) or absence (value is equal to 0) of some competency.

2. *The flexible reward function* (unlike classical DQN, that typically uses fixed rewards – e.g., +1 for a correct action, -1 for a mistake), HR-QoS-DQN utilizes also QoS parameters, their *global importance weights*, and *optimization functions*: cost and time are minimized, while reliability is maximized. Thereby, QoS parameters influence learning by altering the agent's strategic decisions, that is a significant departure from classical DQN.

Such reward function can be customized to meet various requirements. Additionally, we incorporate the ability to set *aggregate functions* for calculating QoS parameters for the resulting flow. For example, in the task of PLP construction, cost and time are summed, while rating is determined as the arithmetic mean.

3. *The alternative modes of the state transition.* We support the possibility of using two modes of state update: mode without accumulation imitated the traditional DQN learning system, and mode with accumulation, that enable extended functionality. The accumulating mode implementation can expand the applicability of the proposed method.

In classical DQN, during each action the state is updated by replacing the current state with the one returned by the environment after executing the action, and this approach does not allow for accumulation or retention of parts of the previous state. We implemented an accumulating mode in HR-QoS-DQN, where the agent retains and accumulates all obtained signals for future use. While this approach complicates the model, it provides a more realistic representation of the knowledge acquisition process and in some cases simplifies the format of the environment presentation and makes it more understandable for humans.

4. *The rules prevented loops and deadlock states.* We propose to add in HR-QoS-DQNthe following checks: limit on steps without progress to prevent the agent from continuously taking ineffective actions, benefit check for actions to ensure they contribute new information or progress, avoidance of repeated actions to eliminate redundant steps, handling of deadlock states by implementing emergency exit mechanisms when no progress can be made. The proposed rules accelerate learning by avoiding unnecessary actions, because classical DQN does not verify such situations.

5. *Different types of Replay Buffer*. We use in HR-QoS-DQN two types: FIFO and PER with prior determination of the optimal type for a specific environment during the parameter optimization stage (classical DQN typically uses FIFO, where all experiences have equal importance). We set the buffer type as a parameter for the agent's learning function, adding flexibility for applying the method to different tasks. This modification also enables effective expansion of experience buffer types without additional complexities.

6. *The adaptive selection of* the agent *actions.* As opposed to classical DQN, where actions are selected solely based on Q-values, in HR-QoS-DQN the agent additionally verifies the feasibility of the chosen action according to predefined rules. These rules eliminate repeated actions and prevent unnecessary transitions, improving efficiency and decision-making.

7. *The modified learning process* that differs from classical DQN, where the agent is trained on predefined initial states. In HR-QoS-DQN, two learning modes are provided: fixed initial state and random initial state, with the additional option for explicit specification.

In the first stage, the *Optuna* library is used for automated selection the optimal key parameters with random initial states. The agent is then trained in an environment with random initial state that promotes model generalization. However, this approach requires an increased number of episodes to achieve stable policies due to the agent's interaction with a wider range of initial conditions.

After learning, the model is saved into the file for further use. During the exploitation phase, the model is loaded from the file, and fine-tuning is carried out with the fixed initial state mode. Fine-tuning stabilizes the policy and searches for optimal parameters through additional adjustments. This approach ensures model adaptability, improves generalization, and enhances performance under operational conditions.

8. *The dynamic set of target states* (Multi-Goal Learning) is used in HR-QoS-DQN for target state formation, as opposed to a single fixed target state in classical DQN.

9. *The automated selection of optimal parameters and strategies* for subsequent use in the learning and exploitation processes of HR-QoS-DQN.

10. *The integration of learning and exploitation with fine-tuning support* differs from classical DQN, where the learning and exploitation processes are separated. In HR-QoS-DQN, a flexible three-stage algorithm is implemented. The first stage involves selecting optimal parameters and functions. At the second stage the agent is trained in an environment with various initial states, promoting policy generalization. The third stage is to operate the model with the possibility of fine-tuning, enabling adaptation to gradual changes in the environment, such as changes of service availability , changes in QoS parameter values and graph modifications (altering the input and output parameters of services).

11. *The continuous learning* of the agent is provided by HR-QoS-DQN after completing learning by saving not only the weights of the Q-network (as in classical DQN) but also additional elements such as the memory buffer and other configuration parameters. This approach simplifies the relearning process and supports the algorithm adaptability. It significantly enhances the agent's flexibility and stability, allowing it to function effectively under gradual changes in environmental parameters and data structures.

9. Software Implementation of HR-QoS-DQN

We developed an implementation of the modified DQN algorithm HR-QoS-DQN within an environment that simulates the execution of service sequences to achieve a target state, with the environment dynamically changing in our modification. The project is implemented using Python, one of the most popular programming languages in the fields of AI and ML. Python offers an extensive set of libraries and frameworks that significantly simplify the development and deployment of deep learning algorithms. The use of Python in this project ensured the efficient implementation of the DQN algorithm along with the modifications, allowing for the PLP construction, facilitates the development, optimization, and scaling of the solution, as well as its adaptation to environmental changes.

In this work, we use several powerful Python libraries to enable the effective DQN implementation, along with automated hyperparameter tuning and the formula for strategy calculation: *NumPy* (Numerical Python) – a library for working with multidimensional arrays and performing numerical operations, *PyTorch* – one of the most popular libraries for implementing deep learning, *Optuna* – a modern and

powerful library for automated parameter tuning, and *Collections* (*deque*) – a part of Python's standard library, providing data structures for efficient processing (double-ended queues) used to implement the Replay Buffer, organizing a FIFO queue to store the recent experiences of the agent and supporting neural network learning on buffer samples. These libraries enable the implementation of the agent learning in a dynamic environment of services. NumPy ensures fast numerical computations, PyTorch implements the neural network and gradient descent, deque optimizes memory management for storing experiences, and Matplotlib is used for analyzing learning results. This combination of libraries makes the system efficient, productive and scalable.

We test the HR-QoS-DQN algorithm in an environment with 1029 services and 126 signals that generate 526 states. Fragment of printout on Figure 1 describes the execution of algorithm in this environment (number of states and services) with selected initial and goal states, displays one of resulting optimal paths and their final QoS values, calculation time (optimal path search time) and automatically determined optimal parameters of learning.

```
Optimal path from Basic_Mathematics to Data_Science_Expertise (buffer FIFO):
<Basic_Mathematics> -> (Calculus)
<Calculus_Knowledge, Basic_Mathematics> -> (Linear_Algebra)
<Calculus_Knowledge, Linear_Algebra_Knowledge, Basic_Mathematics> -> (Probability)
<Calculus_Knowledge, Probability_Knowledge, Linear_Algebra_Knowledge, Basic_Mathematics> ->
(Advanced_Programming_Theory_&Python)
<Probability_Knowledge, Basic_Mathematics, Programming, Calculus_Knowledge,
Linear_Algebra_Knowledge, Python_Knowledge> -> (ML Basics)
<Probability_Knowledge, Machine_Learning_Knowledge, Basic_Mathematics, Programming,
Calculus_Knowledge, Linear_Algebra_Knowledge, Python_Knowledge> -> (Data Science)
<Probability_Knowledge, Machine_Learning_Knowledge, Basic_Mathematics, Programming,
Data_Science_Expertise, Calculus_Knowledge, Linear_Algebra_ Knowledge, Python_Knowledge> (Data
Science)
Total QoS: [26000.0, 400.0, 5.0] | Search time: 0.12 sec

Number of signals: 123
Number of states: 526
Number of alternative services: 1029
lr: 0.0035911034864585293
gamma: 0.9465310980409525
epsilon: 0.9358432119414084
epsilon_min: 0.08347997122260542
epsilon_decay: 0.9224750480640653
batch_size: 128
memory_size: 20000
max_no_progress: 10
max_steps: 58
target_update_freq: 67
epsilon_function: linear
```

Figure 1: Printout of HR-QoS-DQN algorithm execution (fragment)

The learning stage of HR-QoS-DQN with a random initial state is characterized by the graphs (Figure 2) of the learning curve defined by the QoS-based reward (Figure 2-left) and TD Error (Figure 2-right). The learning curve graph shows the cumulative reward an agent receives for completing tasks. The learning curve is unstable due to the variability of rewards and states, that complicates to assess the agent's progress, but graph demonstrates the goal arrival after 9999 episodes.

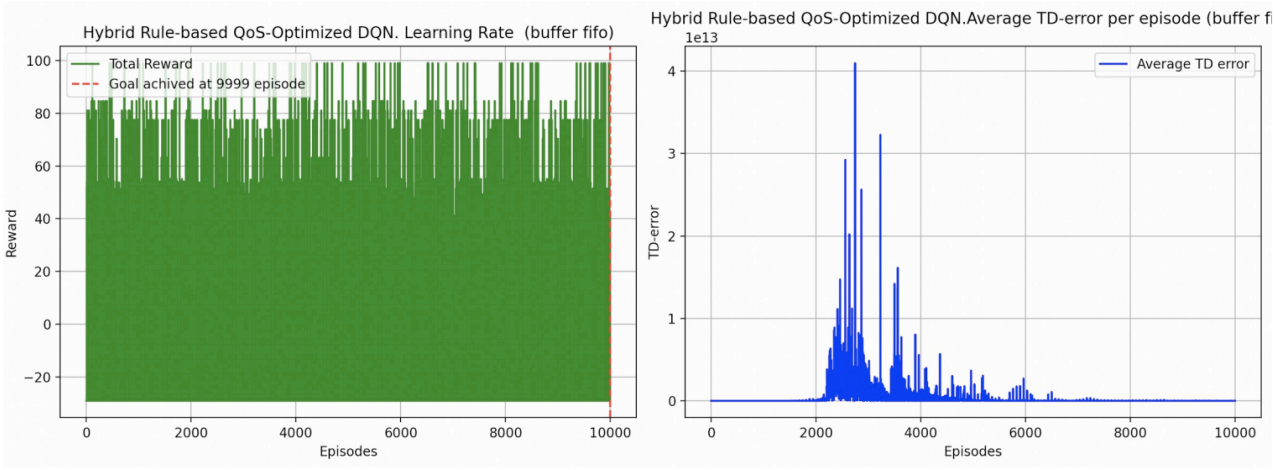

Figure 2: The learning curve and the TD error curve graphs of HR-QoS-DQN algorithm

The TD-error graph displays the average value of Temporal Difference error (TD-error) - the difference between the predicted Q-value and the adjusted value from the Bellman equation, which is a key indicator of the accuracy of the agent's policy during training. The TD error curve graph shows big values for 2000-4000 episodes but fixes the learning progress after 6200 episodes. The decrease in TD-error over time demonstrates that the model successfully adapts to changes in the environment and becomes increasingly efficient in decision-making. Error spikes are expected and even beneficial, as they may indicate that the agent encounters new scenarios and refines its policy.

Learning in the random initial state mode demonstrates effectiveness in creating a universal model that can adapt to dynamic changes in the environment. Fluctuations at the beginning are natural and indicate an active exploration process. The gradual increase in reward and achievement of the target indicator, together with the gradual decrease and stabilization of TD error, confirm that the agent is successfully mastering the policy and is capable of generalization. The graphs confirm that this learning strategy increases the flexibility and adaptability of the algorithm that is critically important for complex environments.

10. Conclusion

The study analyzes the capabilities of reinforcement learning methods and their components. We consider the Deep Q-Network (DQN) method and its distinctions from Q-learning. Based on this analysis, the challenges that necessitate modifications to these methods are identified. These challenges involve practical applications such as adaptation to complex and dynamic environments with changing service parameters, consideration of multiple goals and diverse success metrics, adaptation of transition function and automation of optimal learning parameter selection.

Proposed solutions ensures flexibility, universality and enhanced efficiency method across various domains.

We propose the Hybrid Rule-based QoS-Optimized modification of Deep Q-learning network method HR-QoS-DQN developed to facilitate automatic determination of key learning parameters and incorporates mechanisms for continious learning. This method enables the agent to adapt to the environment changes, such as the removal of services or modification of their parameters, without requiring complete relearning. The algorithm is implemented in Python and tested on the practical task of constructing personalized learning pathways used in the educational process management. Experimental results indicate that HR-QoS-DQN delivers better adaptability to environmental changes compared to Q-learning results obtained from the same datasets. A significant advantage of HR-QoS-DQN is its ease of adaptation to diverse application domain – it requires adjustments to learning parameters, policies, and domain-specific rules, but does not necessitate reprogramming the entire algorithm.

These modifications are aimed to adapt the DQN algorithm for solving tasks related to constructing of the personalized learning pathways used in educational process.

Declaration on Generative AI

During the preparation of this work, the authors used AI programs Chat GPT 4o and Copilot for grammar and spelling check. After using this services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] A.G. Barto, R. S. Sutton, Reinforcement Learning: An Introduction, The MIT Press: Cambridge, Massachusetts, 2018. https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

[2] V. Uc-Cetina, F. Moo-Mena, R. Hernandez-Ucan, Composition of web services using Markov decision processes and dynamic programming. In: The Scientific World Journal, (1), , (2015): 545308.

[3] C. Watkins, P. Dayan, Q-learning. In: Machine learning, 8 (3-4), pp.279–292, (1992): 10.1007/BF00992698, https://www.researchgate.net/publication/220344150_Technical_Note_Q-Learning

[4] B.Jang, M. Kim, G. Harerimana, J. W. Kim, Q-learning algorithms: A comprehensive classification and applications. In: IEEE access, (2019) : 10.1109/ACCESS.2019.2941229 https://www.researchgate.net/publication/335805245_Q-Learning_Algorithms_A_Comprehensive_Classification_and_Applications.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, Playing Atari with Deep Reinforcement Learning, Deep Mind Technologies, (2013) ArXiv preprint arXiv:1312.5602, https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, et al., Human-level control through deep reinforcement learning. Nature, 518(7540), pp.529–533, (2015): 10.1038/nature14236.

[7] J. Fan, Z. Wang, Y. Xie, Z. Yang, A Theoretical Analysis of Deep Q-Learning, Proceedings of Machine Learning Research vol 120:1–4, 2nd Annual Conference on Learning for Dynamics and Control, (2020): 10.48550/arXiv.1901.00137, https://arxiv.org/pdf/1901.00137.

[8] J. Rogushina, A. Gladun, O. Anishchenko, S. Pryima, Semantic Support of Personal Learning Trajectory Development, In: UkrPROG 2024 International Scientific and Practical Programming Conference, CEUR Workshop Proceedings (2024), CEUR Vol-3806, (2024):487-505.

[9] I.Gryshanova, J. Rogushina, Planning Sequences of Learning Object Services Based on Reinforcement Machine Learning. In: // Materials of the XXIV International Scientific and Practical Conference ITB-2024. (2024): 74-77 https://drive.google.com/file/d/1wsLVnueNRd62g-k179IHihuJNOYZqR9G/view

[10] H. Wang, Xuan Zhou, Xiang Zhou, W. Liu, W. Li, and A. Bouguettaya, Adaptive Service Composition Based on Reinforcement Learning, In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds) Service-Oriented Computing. ICSOC 2010. Lecture Notes in Computer Science, vol 6470, pp 92–107, (2010). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-17358-5_7.

[11] V. Uc-Cetina, F. Moo-Mena, R. Hernandez-Ucan, Composition of web services using Markov decision processes and dynamic programming. In: The Scientific World Journal (1), (2015): 545308.

[12] L. J. Lin, Self-improving reactive agents based on reinforcement learning, planning and teaching. Machine learning, 8, (1992): 293-321. https://doi.org/10.1007/bf00992699.