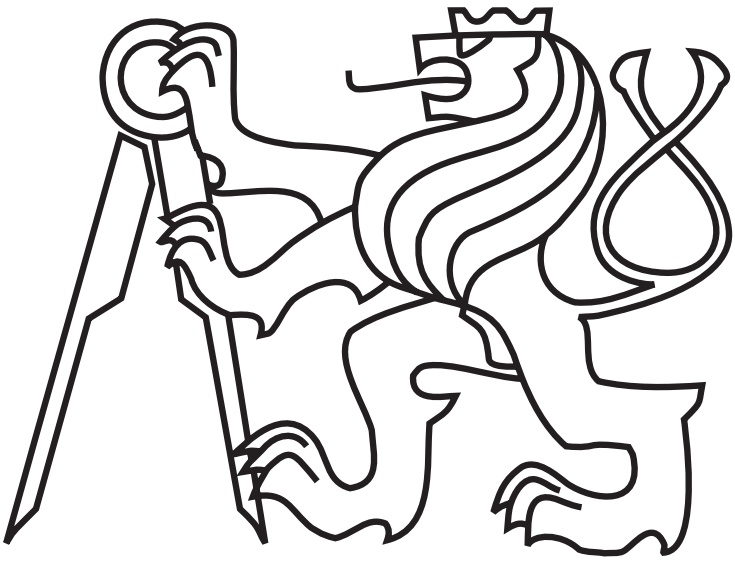
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

BACHELOR THESIS



Jan Pikman

Federated Learning for Robotic Navigation

Department of Cybernetics

Thesis supervisor: **Ing. Zdeněk Rozsypálek**

May, 2022

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Pikman Jan** Personal ID number: **492098**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Federated Learning for Robotic Navigation

Bachelor’s thesis title in Czech:

Společné učení pro robotickou navigaci

Guidelines:

The project aims at deployment of federated learning methods for navigation of small robots with constrained sensors. The goal is to investigate the capabilities of the federated learning methods in sharing experiences among robots with constrained perception.
1) Get to know the core principles of deep reinforcement learning [1].
2) Get to know the principles of federated learning algorithms [2].
3) Learn the basic scenarios used to evaluate the efficiency of machine learning methods for navigation of robots with constrained sensory equipment [2,3]
4) Based on known literature choose an appropriate scenario, where federated learning could be deployed for small robots and replicate the experiments presented.
5) Propose modifications of the learning process and formulate hypotheses how to assess their impact to the navigation efficiency and robustness.
6) Conduct preliminary experiments evaluating the hypotheses.

Bibliography / sources:

[1] Goodfellow Ian, et.al.: Deep Learning. MIT Press, 2016
[2] Yang Qiang, Liu Yang, Cheng Yong, Kang Yan, Chen Tianjian, Yu Han, Federated Learning, Morgan & Claypool, 2019.
[3] Na Seongin, et.al.: Federated reinforcement learning for swarm robotic systems. In IROS 2022 (in rewiev).
[4] Sutton R. S., Barto A. G., Reinforcement Learning: An Introduction, 2nd ed. The MIT Press, 2018.

Name and workplace of bachelor’s thesis supervisor:

Ing. Zdeněk Rozsypálek Department of Computer Science FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **15.12.2021** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

Ing. Zdeněk Rozsypálek	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____ Date of assignment receipt

_____ Student's signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Author statement for undergraduate thesis:

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, May 20, 2022 .....

Jan Pikman

Acknowledgements

I would like to thank all people who supported me while working on this thesis and during my studies. My gratitude goes to Tomáš Krajník for bringing me to the topic of this thesis. Special thanks belong to my supervisor Zdeněk Rozsypálek for his guidance and always helpful consultations during which I learned many new things. I am most grateful to my parents for their love, encouragement, and support whenever I needed it. Last but not least, I want to thank all members of my extended family who always make me feel better.

Abstract

This work focused on the use of federated learning in combination with deep reinforcement learning to complete the task of robotic navigation with restrained sensory equipment. An attempt is made to replicate the success of federated learning with the soft weight update, which was introduced in the paper *Federated Reinforcement Learning for Collective Navigation of Robotic Swarms*. This included the creation of a scenario where several *TurtleBot3* robots in independent playgrounds navigated to goals. The replicated method does not reach the expected evaluation performance. Four changes and additions to the method were introduced to improve its behavior. Two propositions enhanced the algorithm performance during evaluation and even managed to outperform other current methods. Although their success was not as significant as the success of the replicated method in the original paper.

Abstrakt

Tato práce se zaměřila na použití federovaného učení v kombinaci s hlubokým posilovaným učením k řešení problému robotické navigace s omezenou senzorickou výbavou. Byl učiněn pokus replikovat úspěch metody společného učení s měkkou aktualizací vah, která byla představena ve článku *Federated Reinforcement Learning for Collective Navigation of Robotic Swarms*. Součástí replikace bylo vytvoření prostředí, ve kterém se několik robotů *TurtleBot3* umístěných v oddělených hřištích snažilo dojet do stanovených cílů. Replikovaná metoda nedokázala během vyhodnocování dosáhnout předpokládané úspěšnosti. Byly představeny čtyři úpravy pro zlepšení jejích vlastností a chování. Dvě z nich dokázaly zvýšit úspěšnost při vyhodnocování a dokonce jejich výkony překonaly v dnešní době nejlepší trénovací algoritmus, ačkoliv jejich úspěch nebyl tak výrazný jako úspěch replikované metody v původním článku.

Contents

1 Introduction **1**

2 State of the Art **2**

2.1 Reinforcement Learning 2

2.2 Deep Learning 5

2.3 Deep Reinforcement Learning 6

2.4 Federated Learning 8

2.5 Federated Reinforcement Learning 10

2.6 Machine Learning for Robotic Navigation 13

3 Methodology **16**

3.1 Scenario 16

3.2 Proposed Improvements 20

4 Experiments **25**

4.1 Hyperparameter Search 25

4.2 Replication 28

4.3 Results of Proposed Improvements 35

5 Conclusion **40**

List of Figures

1 Agent-environment loop for step t 2
2 Graph of simple feedforward neural network 5
3 Horizontal federated learning diagram 9
4 Vertical federated learning diagram 10
5 Federated transfer learning diagram 11
6 Diagram of compared methods in paper Federated Reinforcement Learning for Collective Navigation of Robotic Swarms 12
7 Classical hierarchy of robot navigation 14
8 Playgrounds used in training and testing 17
9 Diagram of neural network architectures 19
10 Learning playground with additional starting points 24
11 The learning performance of SNDDPG with different experience buffer sizes 27
12 The learning performance of FLDDPG with different federated update periods 29
13 The mean success rates in the last 25 episodes of FLDDPG learning with different federated update periods 30
14 The comparison of learning performance of soft update FLDDPG with other methods 31
15 The average rewards obtained during learning of compared methods . . . 32
16 The proportions of successful runs obtained during learning of compared methods 33
17 The proportion of successful runs during the evaluation using best-performing agents 34
18 The evaluation results of compared methods across all trained networks . . 35
19 The proportions of successful runs obtained during learning of the proposed improvements 36
20 The evaluation results of the proposed improvements 37

List of Tables

1 Evaluation scenarios 15
2 Comparison of hyperparameter values 26
3 Communication efficiency of compared methods 32

List of Algorithms

1 Main body of Federated DDPG 21
2 Federated update: Soft Federated Averaging 21
3 Federated update: Positive Weighting 22
4 Federated update: Real Weighting with absolute normalization 22
5 Federated update: Global Soft Update 23

1 Introduction

Throughout history, humankind has been attempting to automate as many tasks as possible to increase production or improve its livelihood. Many of these problems are already successfully automated, but many are still waiting to be solved. One of those waiting problems specific to robotization is the ability to learn and perform any given task.

The learning of robots can be done by reinforcement learning (RL). It provides techniques and algorithms for maximizing the obtained reward which is done by creating, evaluating, and updating value functions. Searching for the suitable implementation of value functions used to be difficult. Fortunately, in past years, advances in the field of deep learning (DL) have provided us with neural networks which are able to efficiently approximate value functions [1]. For successful RL, a large amount of data must be collected. When learning with several agents in similar environments, it is possible to use principles of federated learning (FL) to improve performance [2]. Some methods of FL can also decrease the necessary amount of communication rounds which could simplify the learning process.

One of the frequent tasks in robotics is autonomous navigation, which corresponds to the robot's ability to move in the environment from one position to another without collision. This complex problem is traditionally solved by dividing it into two tasks - global path planning and local motion control - and requires the combination of various research areas. The different approach to robot navigation exploits the recent advancements in the field of machine learning (ML) and artificial intelligence.

This thesis is structured into three main chapters. The first chapter provides a brief overview of the research fields used in this work. It also introduces the content of the paper [3], which will be replicated in the following chapters. The second chapter describes the scenario used for testing the replicated and proposed methods, which are also introduced. The third chapter describes the conducted experiments - hyperparameter search, replication, and proposed improvements - and discusses the results.

2 State of the Art

This chapter contains a brief overview of the research fields used in this thesis, accompanied by examples of recent achievements. First, RL is presented with a focus on the Q-learning algorithm. Then the basic principles of deep learning are explained. In the next subchapter, these two fields are combined and deep reinforcement learning is introduced including the DDPG method. In the following two chapters, federated learning is briefly described and it is combined with deep reinforcement learning to create federated reinforcement learning. Finally, the robotic navigation is introduced and a comparison of evaluation scenarios with restricted sensory equipment is done.

2.1 Reinforcement Learning

RL approaches the problem of autonomous learning through trial and error. In the past, methods of RL were successfully applied to play computer games and complete tasks in simulations. Recently, RL has shown promising results in robotics, enabling robots to learn complex behavior. The two main entities in RL are called the agent and the environment [4].

The agent is taught to take optimal actions A depending on the current environment. It is rewarded for actions leading to correct outcomes and punished for its mistakes. The environment describes the world outside of the agent. The state S of the environment could change depending on the agent's actions or on its own, according to the environment's transition rules. Every state has its reward value R corresponding to the intended behavior. The agent perceives the environment through imperfect state descriptions called observations O. Sometimes when discussing the agent's behavior, observation is often referred to as a state. This difference is easily recognized depending on the context.

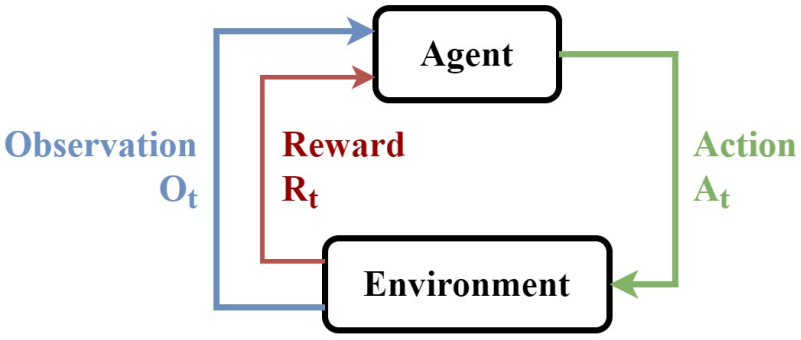


Figure 1: Agent-environment loop for step t. In every step, the agent receives the observation O_t (can also be denoted as S_t) and reward R_t. The agent chooses action A_t which is sent to the environment. The environment then changes its state, and the cycle continues with another step $t + 1$.

Learning takes place in discrete steps, as shown in Figure 1. During every step, the agent receives the observation and reward. The agent then decides which action to take and sends it to the environment. The environment changes its state, and the cycle continues with another step.

The agent selects the action A_t from actions space $\mathcal{A}(S_t)$, which is a set containing all possible actions in the current environment. It is essential to differentiate between discrete action spaces (cardinal directions) and continuous action spaces (real-valued vectors) because it is not possible to apply some RL algorithms to every action space [5]. The action is selected by policy function $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$, which, depending on the current state, produces a probability distribution over action space. When the policy function is deterministic, it can be described as $\mu : \mathcal{S} \to \mathcal{A}$.

During learning at step t the agent receives reward $R_t \in \mathbb{R}$ according to reward function $\mathcal{R}(S_t, A_t, S_{t+1})$. The cumulative reward after step t is then denoted as G_t. Definition of G_t is dependent on the type of task [4]. For episodic tasks which will be running for a finite number of time T:

$$G_t = \sum_{k=0}^{T} R_{t+k+1} \tag{1}$$

And for continuing tasks running for an infinite amount of time, it is necessary to multiply by $0 < \gamma < 1$, so G_t can not reach infinity.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2}$$

The goal of RL is then to find optimal policy π^* maximizing expected return $\mathbb{E}[G_0]$ when the agent acts according to it.

Value functions are an essential part of almost every RL algorithm. They provide information about possible future rewards with respect to specific policy function π. We call function describing expected reward when starting from state S following policy π state-value function, formally defined as:

$$V_\pi(S) = \mathbb{E}_\pi[G_t | S_t = S] \tag{3}$$

Another important function is called action-value, which represents reward of taking action A while in state S:

$$Q_\pi(S, A) = \mathbb{E}_\pi[G_t | S_t = S, A_t = A] \tag{4}$$

Existance of relationship between value functions described above is evident. State-value corresponds to expected action-value of same state, when action A is selected using policy π [5].

$$V_\pi(S) = \mathbb{E}_{A\sim\pi}[Q_\pi(S, A)] \tag{5}$$

Optimal state-value and action-value functions are those following optimal policy π^* and are denoted $V_*(S)$ and $Q_*(S, A)$ respectively.

All value functions satisfy certain recursive relationships. These are described using Bellmann equations for both state-value and action-value functions.

$$V_\pi(S_t) = \mathbb{E}_{S_{t+1}\sim E}[R_t + \gamma V_\pi(S_{t+1})] \tag{6}$$

$$Q_\pi(S_t, A_t) = \mathbb{E}_{S_{t+1}\sim E}[R_t + \gamma \mathbb{E}_{A_{t+1}\sim\pi}[Q_\pi(S_{t+1}, A_{t+1})]] \tag{7}$$

where S_{t+1} is the next environment state, A_{t+1} is next action selected by policy π and reward R_t depends on S_t and A_t. Bellmann equation for state-value function is derived by expanding definition (3) using expected reward (2). The equivalent stands for action-value function. Bellman optimality equations for optimal state-value and action-value functions are almost same as Equations (6) and (7), except for added maximum to search for (next) optimal action maximizing expected reward.

$$V_*(S_t) = \max_{A_t\in\mathcal{A}(S_t)} \mathbb{E}_{S_{t+1}\sim E}[R_t + \gamma V_*(S_{t+1})] \tag{8}$$

$$Q_*(S_t, A_t) = \mathbb{E}_{S_{t+1}\sim E}[R_t + \gamma \max_{A_{t+1}\in\mathcal{A}(S_{t+1})} [Q_*(S_{t+1}, A_{t+1})]] \tag{9}$$

The vast majority of RL algorithms can be split into two approaches: on-policy and off-policy [4]. On-policy methods are directly learning policy functions. This process is usually guided by approximating the state-value function for every new policy and improving it. On the other hand, off-policy methods are learning the action-value function, which is then used to find the optimal deterministic policy by selecting an action with the maximal expected reward.

$$\mu(S_t) = \arg\max_{A_t\in\mathcal{A}(S_t)} Q(S_t, A_t) \tag{10}$$

As mentioned above, RL is used in many tasks where the optimal behavior is difficult to obtain, and it is preferable to learn by trial and error. Below are only two examples of various fields in which RL is used. The methods of RL are applied in finance for portfolio optimization, Robo-advising, option pricing, and hedging [6]. In 2020, RBC Capital Markets launched the Aiden® electronic trading platform based on RL [7].

RL principles are also applied in robotics. Researchers from OpenAI used RL, combined with other machine learning methods, to teach a human-like robotic hand to manipulate Rubik's cube [8]. Solving the puzzle by applying valid moves, they were able to achieve a 60% success rate for half scrambled cubes and 20% for fully scrambled. Many other researchers are training robotic arms for numerous tasks such as autonomous picking and placing of objects [9].

Q-learning

Q-learning is one of the most important RL algorithms [4]. As the name suggests, it is the off-policy method, directly approximating $Q_*(S_t, A_t)$. Learning update can be described in one step, where α is learning rate parameter and γ discount factor:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_t + \gamma \max_{A_{t+1}\in\mathcal{A}(S_{t+1})} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \tag{11}$$

For better intuition, it is possible to rewrite the update as:

$$Q(S_t, A_t) \leftarrow (1-\alpha)Q(S_t, A_t) + \alpha(R_t + \gamma \max_{A_{t+1} \in \mathcal{A}(S_{t+1})} Q(S_{t+1}, A_{t+1})) \tag{12}$$

Now, it is easy to recognize that new Q-value is weighted sum of old value and optimal action-value from Equation (9).

2.2 Deep Learning

In the past years, DL has emerged as a crucial discipline of machine learning. DL focuses on the study and usage of artificial neural networks. Methods of DL are actively used in computer vision, natural language processing, reinforcement learning, and many more. As the name implies, the first neural networks were inspired by nature. However, this similarity slowly diminished as their development continued.

It is possible to describe a neural network as a directed graph of computational layers, which takes a vector as input and returns a corresponding output vector [10]. The most widely used layer is called linear, which performs the affine transformation $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$, where matrix $\mathbf{W}$, vector $\mathbf{b}$ are parameters, $\mathbf{x}$ is input, and $\mathbf{y}$ is output.

The activation function σ is another important layer type as it applies a specified non-linear function to every input value. This nonlinearity is essential for repeated usage of linear layers because, without it, the resulting function would remain linear, which would not improve the network's performance. Commonly used functions are sigmoid, tanh, and ReLU.

These mentioned layer types are only the most basic building blocks for neural networks. Many other layer types are actively used for various specific tasks. For example, convolutional layers are widely used for image processing [11][12], and recurrent layers are applied in natural language processing [13].

Simple feedforward neural network [10] usually consists of consecutive linear layers with activation functions in between as shown in Figure 2, where squares represent individual layers, $\mathbf{x}'$ is layer's input value and layer's output corresponds to result of equation. It

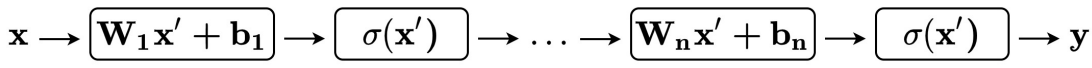


Figure 2: Graph of simple feedforward neural network consisting of consecutive linear layers with activation functions in between. Each square represents one layer, $\mathbf{x}'$ is the layer's input value and the layer's output corresponds to the result of the equation. Matrices $\mathbf{W_1}, \ldots, \mathbf{W_n}$ and vectors $\mathbf{b_1}, \ldots, \mathbf{b_n}$ are parameters.

can be understood as a repeated application of affine transformation always followed by nonlinear transformation, giving considerable expressive power to the neural network.

Learning of neural networks consists of minimizing loss function on training data, while maintaining low error on testing data to avoid overfitting. The most used loss functions are mean squared error (MSE) and cross-entropy. Optimization is traditionally done by an iterative algorithm based on gradient descent. Thus, parameter gradients must be computed.

Gradient computation is performed by an algorithm named backpropagation [10]. This algorithm traverses the computational graph backward and applies the chain rule to calculate required gradients. Stochastic gradient descent (SGD) is a simple and widely used algorithm for training neural networks. SGD is almost identical to the gradient descent algorithm with one important difference: gradients are computed using only a small random part of training data called mini-batch. Because evaluating the whole training dataset would be almost impossible due to its large size.

One possible problem with SGD is the high variance of gradients caused by mini-batches. Momentum algorithms attempt to solve this issue by accumulating an exponential moving average of previous gradients, which is then used for parameter updates [14]. To this category belongs another very popular learning algorithm called Adam [15].

DL is widely used in natural language processing (NLP) for information extraction, text classification, summarization, text generation, and machine translation [16]. DL for NLP often uses encoder-decoder neural network structures combined with various attention methods. State-of-the-art machine translation architecture is called a transformer. It is based on a combination of feed feedforward neural networks with so-called multi-head attention [17].

Another possible field of application is computational biology, which aims to develop models and simulations of biological systems [18]. Recently, a new neural network, called AlphaFold, was created by DeepMind to predict protein structures based on the amino acid sequence [19]. AlphaFold competed in a challenging CASP14 competition, significantly outperforming other participants and reaching accuracy comparable to experimental results.

2.3 Deep Reinforcement Learning

It is only natural to combine two successful machine learning paradigms into one. Exactly this happened with DL and RL giving birth to deep reinforcement learning (DRL). As mentioned above, RL provides us with methods to learn various functions without specifying their exact form. Inventing methods for creating these forms becomes progressively more difficult with an increase in the number of dimensions. Neural networks from DL are applied for exactly this purpose of approximating functions used in RL [1].

The introduction of neural networks as value functions brings a dependence problem since function updates are expected to be independent. However, changing the parameters of a neural network could also affect other updates, which makes learning unstable. This issue is solved by the introduction of target networks [1]. They take part in the update

equations, thus reducing instability. Learning of the target network is done once in a while by exponential averaging with a trained network.

Often used mechanisms in DRL are actor-critic methods [20]. These methods separate learned policy called actor and estimated value function known as a critic. The actor takes consecutive actions in the environment. At the same time, the critic evaluates performed actions. Both models are then updated according to obtained rewards. Iterating this process results in improved learning of both actor and critic functions.

DRL is applied so often when facing problems of RL that sometimes terms RL and DRL are used interchangeably. One of the possible applications is playing various games. For instance, RL was used to teach agents to play hide-and-seek in an environment with moveable obstacles such as walls and ramps [21]. In the beginning, hiders only ran from seekers, but later various strategies and their counters emerged. Ultimately, hiders were able to build simple forts and hide ramps to prevent seekers from finding them.

Another possible usage of DRL is in autonomous driving [22] for solving tasks of motion planning and trajectory optimization. New methods were created for descriptions of the action spaces, behavior cloning, and environment simulations. Nevertheless, this application is still facing many challenges like sampling efficiency and safety.

Deep Deterministic Policy Gradient

Deep deterministic policy gradient (DDPG)[23] is off-policy, DRL algorithm, which can only be used in continuous action spaces. The algorithm uses experience replay buffers to store previous experiences, thus improving the learning stability. DDPG also uses an actor-critic method to learn policy and action-value functions while applying target networks for both the actor and the critic.

First, the policy neural network $\mu(s \,|\, \theta^{\mu})$ called actor and action-value neural network $Q(s, a \,|\, \theta^{Q})$ called critic with parameters θ^{μ} and θ^{Q} are initialized. Target neural networks μ^{targ} and Q^{targ} are initialized with same parameters: $\theta^{\mu^{targ}} \leftarrow \theta^{\mu}$ and $\theta^{Q^{targ}} \leftarrow \theta^{Q}$ The DDPG algorithm then starts by taking the determined amount of steps in the environment using policy μ while collecting experiences to replay buffer $\mathcal{D}$. When it is time to update, batch $\mathcal{B}$ of transitions (S, A, R, S', D) (binary scalar D idicates terminal state) is selected from $\mathcal{D}$. The target value is then computed [24]:

$$y(R, S', D) = R + \gamma(1 - D)Q^{targ}(S', \mu^{targ}(S')) \tag{13}$$

The action-value function is updated (minimization) by using gradient:

$$\nabla_{\theta^{Q}} \frac{1}{|\mathcal{B}|} \sum_{(S,A,R,S',D)\in\mathcal{B}} (Q(S, A) - y(R, S', D))^2 \tag{14}$$

The policy is updated (maximization) by using gradient:

$$\nabla_{\theta^{\mu}} \frac{1}{|\mathcal{B}|} \sum_{S\in\mathcal{B}} Q(S, \mu(S)) \tag{15}$$

Target networks are also updated, where $\rho \in [0, 1]$:

$$\theta^{\mu_{targ}} \leftarrow (1-\rho)\theta^{\mu_{targ}} + \rho\theta^{\mu}, \quad \theta^{Q_{targ}} \leftarrow (1-\rho)\theta^{Q_{targ}} + \rho\theta^{Q} \tag{16}$$

After this update, the cycle starts again with experience collection.

2.4 Federated Learning

The success of artificial intelligence may be partially contributed to an increase in the availability of large datasets. Today, almost every smart device collects data, which is then stored, analyzed, and often sold to other organizations. These actions raise privacy concerns among users and governments. The goal of FL is to provide means for decentralized machine learning without the need to share confidential data with other parties. Another benefit of the FL approach is maximal utilization of the computing power of edge devices in a system combined with a decrease in the number of communications with a central server [2].

For example, a group of shops with various goods has agreed to create a shared machine learning model predicting customer behavior. However, they do not trust each other. Some shops could try to sabotage learning with incorrect data or use shared information to their advantage. This situation is perfect for the FL algorithms.

FL can be divided into three categories depending on data shared by participants. These categories are horizontal federated learning (HFL), vertical federated learning (VFL), and federated transfer learning (FTL). Data of ith data owner can be denoted as $\mathcal{D}_i$, it consists of identifiers $\mathcal{I}$, features $\mathcal{X}$, and labels $\mathcal{Y}$.

HFL is applied when participants share a large portion of feature space and differ in samples. This situation can be formally written as:

$$\mathcal{X}_i = \mathcal{X}_j,\ \mathcal{Y}_i = \mathcal{Y}_j,\ \mathcal{I}_i \neq \mathcal{I}_j, \quad \forall \mathcal{D}_i, \mathcal{D}_j, i \neq j \tag{17}$$

This method of FL is beneficial due to the increase in the number of samples with the same features. For instance, HFL could be used by two banks, A and B, which have very different clients, yet they collect the same information about them, for creating a shared fraud detector based on machine learning, as displayed in Figure 3.

Many HFL algorithms have a client-server architecture [25], where each participant uses the same machine learning model. Firstly, participants compute their local updates, encrypt them, and send them to the server. The server then updates the global model by securely aggregating received local updates. The newly updated model is distributed among participants, and the process repeats.

A commonly used method for update aggregation is called Gradient averaging, also known as synchronous stochastic gradient descent or federated SGD [2]. As the name suggests, it is almost identical to SGD. Local updates correspond to stochastic gradients. The server waits for gradients from all participants and then updates the global model

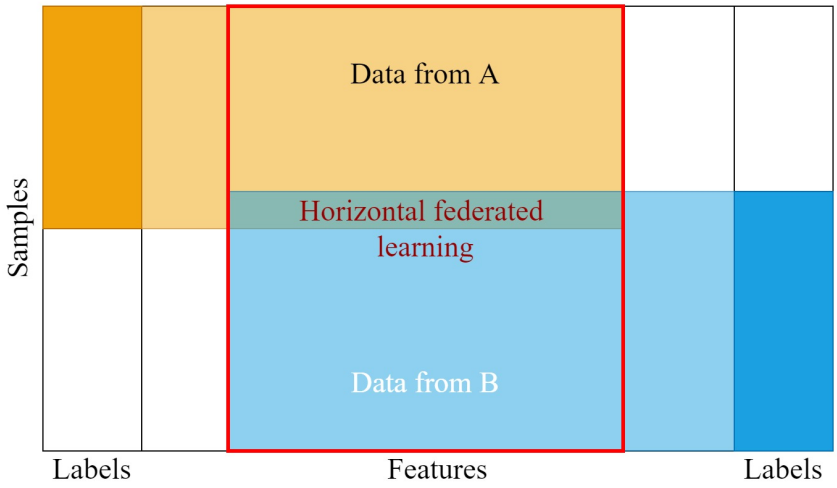


Figure 3: Horizontal federated learning diagram. HFL is used when participants share a large portion of feature space and differ in samples. This method improves learning by increasing amount of samples with the same features. Template from [2].

by their average [26]. This method operates with accurate gradient information and is guaranteed to converge. On the other hand, it requires frequent and reliable communication.

Another used algorithm is Federated averaging (FedAvg) [27], which belongs to the group of model averaging algorithms. In this algorithm, N participants independently train local models. Local updates are parameters $\theta_1, \ldots, \theta_N$ of those locally trained models, which are aggregated in the server by weighted averaging according to the proportion of samples available to participant n_k:

$$\theta_{global} = \sum_{k=1}^{K} \frac{n_k}{n} \theta_k \tag{18}$$

FedAvg algorithm does not require frequent communications. However, its convergence is not guaranteed, and it suffers performance loss compared to Gradient averaging [2].

VFL is convenient when participants differ in feature space but share the same samples. Mentioned relationship can be noted as:

$$\mathcal{X}_i \neq \mathcal{X}_j, \ \mathcal{Y}_i \neq \mathcal{Y}_j, \ \mathcal{I}_i = \mathcal{I}_j, \quad \forall \mathcal{D}_i, \mathcal{D}_j, i \neq j \tag{19}$$

VFL methods expand the number of features of shared samples, which improves the model's ability to learn. This situation can arise when bank A and company B with the same customers would like to train a shared model. Their users are similar, but collected features are very different, and this additional information about customers could improve model performance, as displayed in Figure 4. Among VFL algorithms belong secure federated linear regression and secured federated tree-boosting.

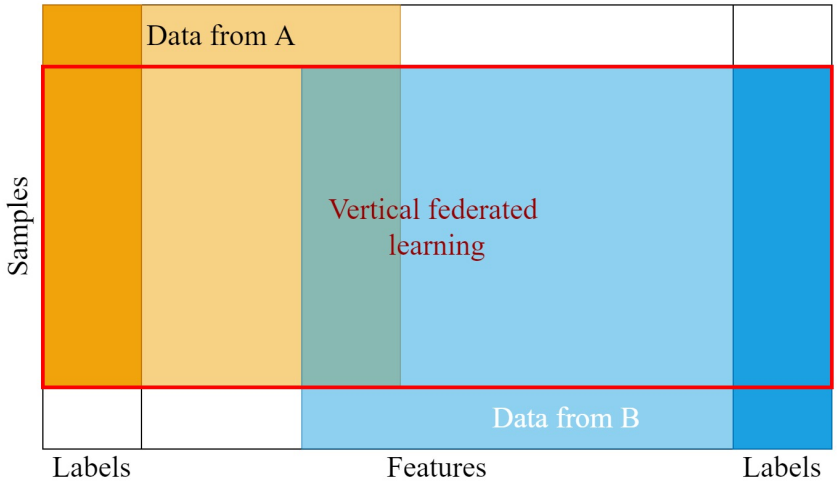


Figure 4: Vertical federated learning diagram. VFL is used when participants differ in feature space but share the same samples. This method improves learning by increasing amount of features of shared samples. Template from [2].

FTL is preferred when only a small portion of the feature and sample space is shared, with the formal description as follows:

$$\mathcal{X}_i \neq \mathcal{X}_j,\ \mathcal{Y}_i \neq \mathcal{Y}_j,\ \mathcal{I}_i \neq \mathcal{I}_j, \quad \forall \mathcal{D}_i, \mathcal{D}_j, i \neq j \tag{20}$$

FTL is a combination of FL and transfer learning (TL). The small shared space is used to build a model, able to predict missing features from one dataset that could be used for other tasks, as shown in Figure 5. FTL methods can be split into three categories: instance-based, feature-based, model-based, depending on the used approach to perform transfer learning.

FL principles mentioned above are ideal for usage in healthcare where data security is essential. Electronic medical records (EMR) from different hospitals could be used for training the shared ML models, which can improve provided care [28]. Models could learn to find patients with similar problems, predict future hospitalizations, drug resistance among patients, or predict chances of survival [29].

Smart homes furnished with the internet of things devices (IoT) are also great for the application of FL. For example, the machine learning model LoFTI [30] was introduced. Its purpose is to eliminate security breaches of IoT devices by learning access control policies.

2.5 Federated Reinforcement Learning

As mentioned above, RL and DRL techniques require a large amount of data, and FL methods are designed to help, thus creating federated reinforcement learning (FRL). For example, FRL was used for so-called personalization - an adaptation of behavior according

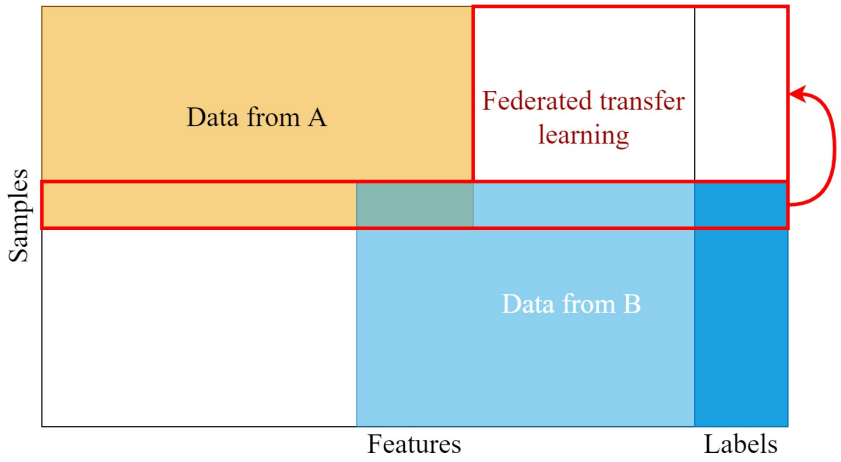


Figure 5: Federated transfer learning diagram. FTL is used when only a small portion of the feature and sample space is shared. The shared space is used to build a model which predicts missing features from the dataset. Inspired by [2].

to a user [31]. Agents were personalized to play the game of Pong at the same skill level as an opponent. RL model of agents was Deep Q-Network (DQN). Global model aggregation was performed by taking the exponential moving average of the previous global model and weights obtained by the FedAvg algorithm. Agents then updated their model by taking an exponential moving average of the agent's model and the new global model. This approach resulted in an improvement of approximately 17% on the personalization time.

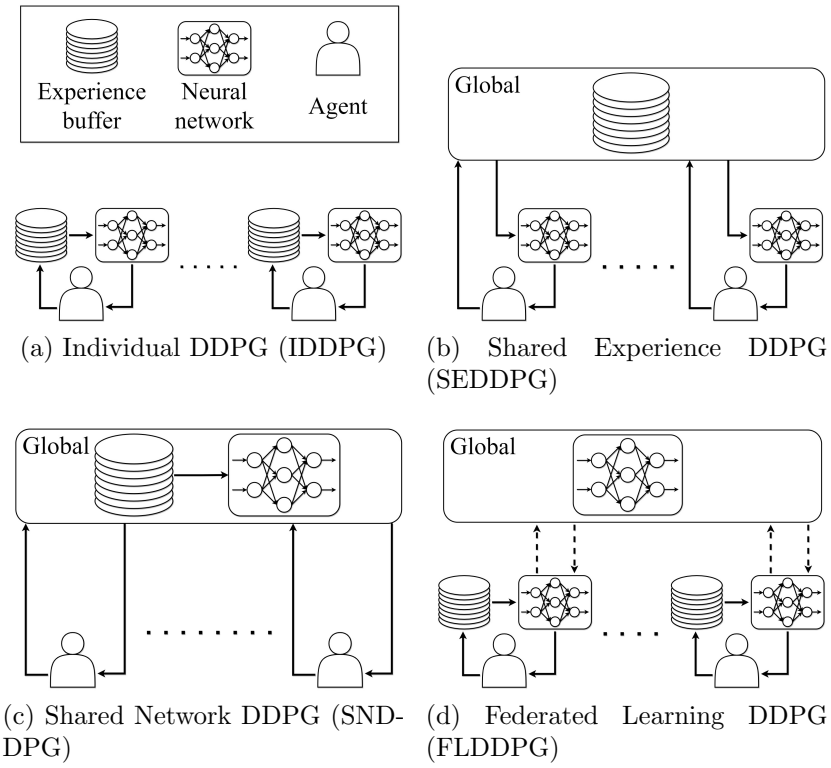
Distributed Reinforcement Learning

A similar field of RL exists, and it is called distributed reinforcement learning. The difference is that distributed RL is not concerned with privacy and data security. It can be split into two groups, depending on the type of global model update: asynchronous and synchronous [2]. Some algorithms of distributed RL are: Asynchronous Advantage Actor-Critic (A3C), General Reinforcement Learning Architecture (Gorila), Advantage Actor-Critic (A2C), and Importance Weighted Actor-Learner Architecture (IMPALA) [32].

Federated Reinforcement Learning for Collective Navigation of Robotic Swarms

The above-mentioned FedAvg algorithm was successfully applied to the DDPG algorithm in a paper called *Federated Reinforcement Learning for Collective Navigation of Robotic Swarms* by Seongin Na [3]. The task was to teach a neural network to control a robot and navigate to a given location without collision with the environment. The learning consisted of six robots, each in its environment with different complexity. The experiences were collected by agents (robots), acting according to a neural network, and stored in

buffers. The neural networks were taught by using collected experience from buffers. The learning was repeated three times with different seeds to gain robust results. Testing was done with only one robot in a more complicated environment than during training. This was repeated 100 times to make results statically significant.



(a) Individual DDPG (IDDPG)

(b) Shared Experience DDPG (SEDDPG)

(c) Shared Network DDPG (SNDDPG)

(d) Federated Learning DDPG (FLDDPG)

Figure 6: Diagram of compared methods in paper [3]. (a) Each neural network was taught independently. (b) Individual neural networks were taught using experience from a shared buffer. (c) A shared neural network was taught using a shared buffer. (d) Each neural network was taught independently while being periodically soft-averaged (similar to FedAvg).

Four different algorithms, as displayed in Figure 6, were compared. The first algorithm was called Individual DDPG (IDDPG) - each robot had its experience buffer and neural network, which was trained without sharing any information with others. Another method was Shared Experience DDPG (SEDDPG) - the experience buffer was shared, while each robot had an independent neural network. Shared Network DDPG (SNDDPG) - both the experience buffer and neural network was shared. The final algorithm, which was the one proposed, was called Federated Learning DDPG (FLDDPG) - each robot had its experience buffer and neural network, similar to IDDPG, but neural networks were periodically soft-averaged.

During training, FLDDPG managed to outperform other methods in both average collected reward and convergence speed while substantially decreasing the number of communication rounds. In testing, FLDDPG reached its goal in 96% of tests, while IDDPG,

SEDDPG, and SNDDPG only in 17%, 34%, and 24%. With regard to mean mission completion time, SEDDPG, SNDDPG, and FLDDPG resulted in a similar time of around 11.8 s. IDDPG was much slower, with a completion time of 14.52 s. FLDDPG was also able to choose a more effective trajectory than others.

2.6 Machine Learning for Robotic Navigation

The ability of a robot to safely and independently move to its goal location in the environment is a crucial part of mobile robotics. The field focused on this problem is called autonomous robot navigation. The successful solution requires a combination of many research areas of robotics, such as mapping, localization, motion control, planning, and perception [33].

The classical approach to robot navigation, as displayed in Figure 7, is to split the task into two parts: global path planning and local motion control. Global path planning takes into consideration knowledge of the environment and goal. The global representation of the environment is created by merging previous and current perception streams. These data are then processed into a global path, often defined as a sequence of local goals. Local motion control is responsible for navigation to a local goal by directly sending commands to motors. This requires more accurate local representation than global planning, and it is usually obtained only from current sensory streams. Even though classical methods enable autonomous navigation in various environments, its performance still lags behind direct human control, especially in complex and diverse environments.

With the recent development of machine learning, the number of its applications in the field of autonomous navigation increased. Some applications respect the classical hierarchy, as mentioned above, and focus on a specific part of the navigation process, while others ignore hierarchical principles and perform end-to-end learning, which searches for direct mapping from sensory inputs to motion commands. ML autonomous navigation methods currently face major problems. Due to large data requirements, training ML models in a real environment is complicated. Thus many methods were studied only in simulations. Another complication is the black-box character of ML, where it is impossible to obtain reasoning behind performed actions and to point out which sections of the algorithm are causing unexpected behavior.

An essential part of every research paper is the scenario used for evaluating the behavior of the proposed method and for comparison with other approaches. It is possible to divide scenarios for autonomous navigation according to the sensory equipment of the robot. A portion of researchers is using robots with cameras or LiDARs, which are able to collect large amounts of observed values. Others are for their simplicity using robots with constrained sensory equipment, often consisting of a relatively low number of rangefinders.

Evaluation scenarios of six papers focused on ML for robotic navigation with constrained sensory equipment are listed in Table 1. The task was always to reach a stationary goal while avoiding collision with surroundings. The environment usually consisted of a playground

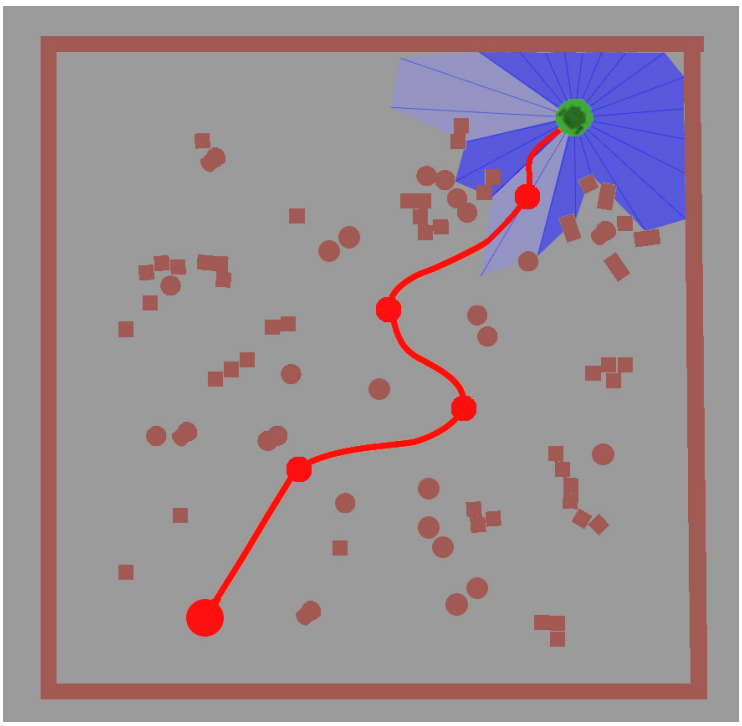


Figure 7: Classical hierarchy of robot navigation. The robot's (green) task is to reach the global goal (large red circle). A global path planner created a global path (red line) as a sequence of local goals (small red circles) by using information about the location of obstacles from previous runs. The local motion controller sends commands to the robot's motors and navigates towards the local goal by using current perception streams (blue rangefinders). Template from [33].

with randomly scattered obstacles or a more trivial maze, possibly with moving obstacles. Almost every time, observation space included relative position to goal in polar coordinates and typically rangefinders surrounding the robot to some extent. A few times, the previous action was also included. The character of action space was highly dependent on the chosen base algorithm because of its properties. For example, the basic Q-learning algorithm is only applicable to discrete action spaces. However, even discrete actions were always described in linear and angular velocities similarly to continuous ones.

Table 1: Scenarios for evaluation of machine learning methods for robotic navigation with constrained sensory equipment. In every listed paper, the task was to reach a stationary goal while avoiding collision with surroundings. The environment column contains a short description of the evaluation environment. The algorithm on which the paper builds is mentioned in the base algorithm column. Mentioned algorithms are: Deep Deterministic Policy Gradient (DDPG), Q-learning, Deep Q-Network (DQN), Double Deep Q-Network (DDQN), Asynchronous Advantage Actor-Critic (A3C). The remaining two columns describe observation and action space.

Paper	Environment	Base algorithm	Observation space	Action space
[3]	randomly placed obstacles	DDPG	24 rangefinders (360 degrees), relative position to goal in polar coordinates, previous action	linear velocity, angular velocity
[34]	simple spiral	Q-learning	3 sonar sensors (left, front, right) with discrete measurements	forward, turn left, turn right
[35]	stationary obstacles, moving obstacles	DQN	360 rangefinders (360 degrees), relative position to goal in polar coordinates	discrete actions
[36]	randomly placed obstacles	DDPG	10 rangefinders (-90 to 90 degrees), relative position to goal in polar coordinates, previous action	linear velocity, angular velocity
[37]	randomly placed obstacles	DDQN	13 rangefinders (-90 to 90 degrees), relative position to goal in polar coordinates	5 possible angular velocities
[38]	simple maze	A3C	70 rangefinders (-135 to 135 degrees), relative position to goal in polar coordinates	forward, left turn, right turn, sharp left turn, sharp right turn

3 Methodology

The following chapter contains the descriptions of the used scenario, replicated methods, and newly proposed method improvements. The paper Federated Reinforcement Learning for Collective Navigation of Robotic Swarms by Seongin Na [3], already mentioned in Section 2.5, serves as a foundation for this thesis. Therefore, the objective of teaching neural networks for robotic navigation is the same, and similar scenarios, experiments, and base methods were used.

3.1 Scenario

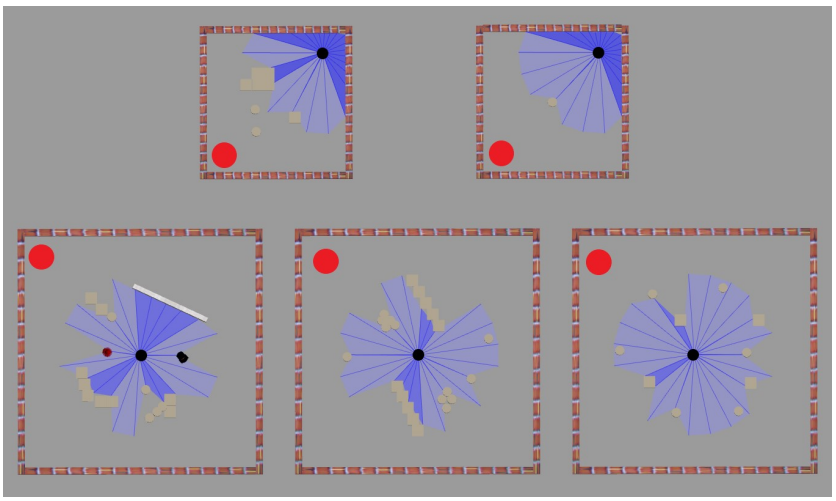
The comparison of the proposed improvements was conducted in a simulation using Robot Operating System (ROS) and Gazebo robot simulator. This allowed for lower learning and evaluation time than in real life, which led to the ability to conduct more experiments in a limited amount of time. The robot used in the simulation was *TurtleBot3 Burger* with a differential drive and laser rangefinders.

During the learning phase, five robots were trained at the same time, each in a different square playground, as displayed in Figure 8. Two smaller playgrounds had sizes 6×6 m and focused on the robot's ability to drive around obstacles. The other three playgrounds were larger with size 10×10 m and attempted to teach navigation with obstacles placed all around robots. In these playgrounds, the goal was always in the same position. In the evaluation phase, only one robot navigated in the playground of size 12×12 m. The playground was more complicated than those during learning, and the goal was positioned in a different corner for each run. The learning playgrounds are exactly the same as used in the paper [3] and were provided by the author. The evaluation playground was constructed to resemble the original one as much as possible.

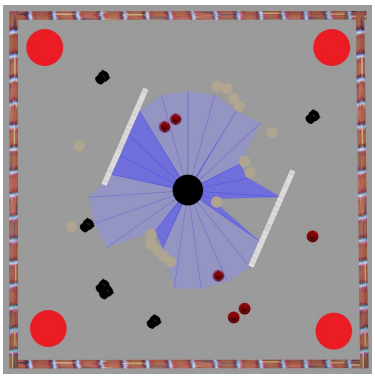
The simulation run was split into E episodes during which the robot attempted to reach its goal. During learning, when the robot collided with the environment or reached the goal, it was returned to its starting position to start its task anew. This was not the case during the evaluation when the collision caused the episode to end immediately. Each complete episode consisted of T steps, every one of which lasted 0.1 s in simulation time.

Observation Space

The observation space of each robot can be split into three parts. The first part consists of 24 rangefinder readings X_t with maximal measured distance $X_{max} = 3.5$ m. Rangefinders are evenly distributed around the robot covering all 360 degrees. These readings provide the robot with information about its surroundings and enable it to avoid collisions. Another part is the current polar coordinates $P_t = (d_t, \theta_t^d)$ of the robot with the goal used as a reference point. The usage of precise coordinates is dependent on the assumption that the perfect localization of the robot is available. Providing this information enables the



(a) Training



(b) Testing

Figure 8: Image of playgrounds used for training and testing. Black dots represent starting position of the robot. Red dots represent the goal. (a) During training, five robots were simultaneously learning in playgrounds with different difficulty levels. (b) Testing was done by one robot navigating to the goal located in one of the corners.

robot to move towards the goal. The last part is an action performed in the previous step A_{t-1} consisting of translational and rotational velocities (v, ω). Thus, observation can be represented as a 28-dimensional vector S_t as displayed in Figure 9.

Action Space

Every action A_t consits of two values - translational and rotational velocity (v, ω). Translational velocity is constrained to $v \in [0, 0.25]\,\mathrm{m/s}$, whereas rotational velocity lies in range $\omega \in [-1, 1]\,\mathrm{rad/s}$. These values were choosen in consideration with real world characteristics of *TurtleBot3* and other research papers using similar scenarios [39].

Reward

The reward function $\mathcal{R} : \mathcal{S} \to \mathbb{R}$ is constructed as a sum of three functions denoted in Equation 21.

$$\mathcal{R}(S_t) = \mathcal{R}_c(S_t) + \mathcal{R}_g(S_t) + \mathcal{R}_d(S_t) \tag{21}$$

The first function $\mathcal{R}_c$ can be understood as a penalty R_c for a collision when the minimal rangefinder value is smaller than the collision distance D_c.

$$\mathcal{R}_c(S_t) = \begin{cases} R_c, & \text{if } \min_{x \in X_t} x < D_c \text{ (collision)} \\ 0, & \text{otherwise} \end{cases} \tag{22}$$

Function $\mathcal{R}_g$ represents a reward R_g for reaching the goal. This fact is checked by comparing whether robot's distance to goal d_t is smaller than the goal distance.

$$\mathcal{R}_g(S_t) = \begin{cases} R_g, & \text{if } d_t < D_g \text{ (reached goal)} \\ 0, & \text{otherwise} \end{cases} \tag{23}$$

The reward for getting closer to the goal was provided by the function $\mathcal{R}_d$, where the difference between current and previous distances from the goal was multiplied by the distance factor R_d.

$$\mathcal{R}_d(S_t) = R_d(d_t - d_{t-1}) \tag{24}$$

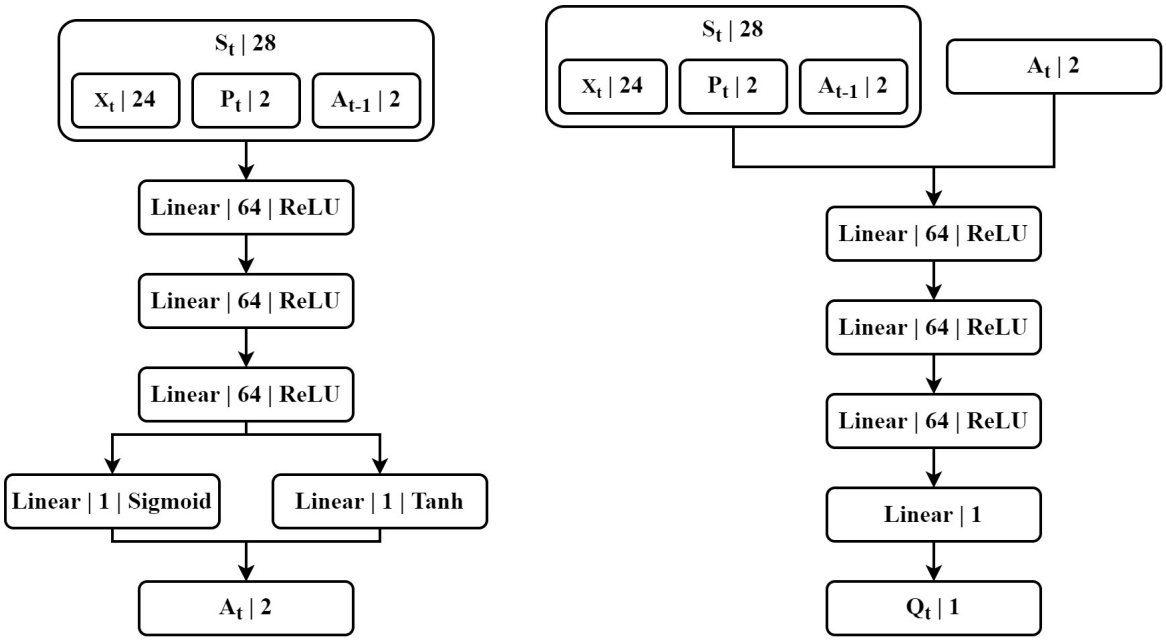
During experiments the rewards were $R_c = -10$, $R_g = 100$, and $R_d = 40$. The distances were $D_c = 0.25$ m, and $D_g = 0.5$ m.

Neural Network Architecture

The DDPG algorithm uses for learning two types of neural networks - actor and critic, as mentioned in Section 2.3. The architectures of the actor and critic neural networks are displayed in Figure 9.

The input of the actor network was a 28-dimensional current state S_t, followed by three 64-dimensional consecutive linear layers with ReLU activation functions between them. Then followed by two 1-dimensional output linear layers. One with a sigmoid activation function, the other with a hyperbolic tangent activation function, which limited their value range to $[0, 1]$ and $[-1, 1]$, respectively. The output layers were initialized using Xavier uniform initialization to prevent problems with vanishing gradients. Two resulting values were then rescaled and concatenated to create action A_t.

The critic network input values were current state S_t and action A_t. Those two value arrays were concatenated and fed into three 64-dimensional consecutive layers with ReLU activation functions. The output action-value Q_t was obtained by a 1-dimensional linear layer without an activation function.



(a) Actor (b) Critic

Figure 9: Diagram of neural network architectures. Every layer is represented by its type, resulting dimension, and following activation function. State S_t is created by concatenating 24-dimensional rangefinder measuments X_t, robot's polar coordinates to goal P_t, and previous action A_{t-1}. (a) Actor neural network takes the state as input and produces action. (b) Critic neural network outputs expected reward from the input of state and action.

Metrics

The performance of algorithms was measured using several metrics during training and evaluation. The most intuitive way for method comparison in RL is to compare the mean of rewards collected during an episode. Unfortunately, higher rewards do not necessarily translate into a better ability to reach the goal. Thus, the mean proportion of successful runs in an episode was also gathered. For example, the value of 0.6 was obtained when during episode three, out of five, robots reached the goal. Another tracked metric was the number of communication rounds N_{comm} needed to complete learning. The communication round is understood as any exchange of information between the robot and the central server. The N_{comm} was calculated as described in the Equation 25,

$$N_{comm} = (N_{data} + N_{NN})E \tag{25}$$

where N_{data} corresponds to the amount of data exchanged in one episode, the N_{NN} is the number of neural network updates in an episode, and E is the total number of learning episodes.

In the evaluation phase, three metrics were used for comparison. First, the mean proportion of successful runs across all episodes. Second, the mission completion time. It corresponds to the average time in which the robot reached the goal without collision. And third, the trajectory efficiency which is the average Euclidean distance between the starting point and the goal divided by the actual distance traveled by the robot. Therefore, it necessarily lies in the interval $(0, 1]$.

3.2 Proposed Improvements

The base of the Federated DDPG algorithm as used in the paper [3] is described in Algorithm 1. First, N actor and critic neural networks are initialized, while their corresponding target networks are initialized with the same parameters. Additionally, N empty replay buffers are initialized. Then algorithm loops for E amount of episodes, which were split into T steps as already described in Section 3.1. During each step, the actions are sampled according to the ϵ-greedy approach - uniformly sampled action is selected with probability ϵ, and action produced by actor neural network is used with probability $1 - \epsilon$. Collected transition samples are then stored in corresponding replay buffers. If the current step is training one according to period T_{train}, then each actor and critic neural networks are independently trained as described in the DDPG algorithm in Section 2.3. Similarly, an independent update of target neural networks is performed with period T_{target}. At the end of every step, the ϵ value is decreased by multiplying with coefficient ϵ_{decay}. Federated update of actor and critic neural networks is done with period E_{update} at the end of the episode.

For federated update, the Federated Averaging algorithm with the soft update was used, as shown in Algorithm 2. The addition of the soft update, as shown in line 3, was done to enable the adaptation of neural networks to different environments, and to reduce the possible performance decline caused by the update. The parameters of neural network θ_k were multiplied by τ and then multiplied weights averaged across all neural networks $(1-\tau)\theta_w$ were added. τ parameter lies in range $[0, 1]$. When τ is set close to 1, the method is analogous to the FedAvg algorithm. On the other hand, when τ is near 0, it becomes identical to Individual DDPG as described in Section 2.5.

The above-mentioned approach could be changed in several ways to improve its behavior. Firstly, FedAvg does not take into account the agent's performance while computing the parameter averages. Making the parameters of successful agents more important during averaging could improve the learning speed and success rate. Another improvement might be achieved by changing the individual soft update into a global one which may result in more stable learning. Finally, the success rate could be increased by diversifying the sampled experiences.

Algorithm 1: Main body of Federated DDPG

Randomly initialize N actor neural networks $\mu_{1,\dots,N}(s|\theta^{\mu}_{1,\dots,N})$, and N critic neural networks $Q_{1,\dots,N}(s,a|\theta^{Q}_{1,\dots,N})$ with weights $\theta^{\mu}_{1,\dots,N}$ and $\theta^{Q}_{1,\dots,N}$
Initialize N target neural networks $\mu^{targ}_{1,\dots,N}$ and $Q^{targ}_{1,\dots,N}$ with same weights as non-target ones $\theta^{\mu^{targ}}_{1,\dots,N} \leftarrow \theta^{\mu}_{1,\dots,N}$ and $\theta^{Q^{targ}}_{1,\dots,N} \leftarrow \theta^{Q}_{1,\dots,N}$
Initialize replay buffers $\mathcal{D}_{1,\dots,N}$
for *episode* $= 1, \dots, E$ **do**
  for $t = 1, \dots, T$ **do**
    Run one step of simulation with actions sampled randomly with probability ϵ or according to actor neural networks $\mu_{1,\dots,N}$ with probability $1-\epsilon$. Collect N transition samples $(S_t, A_t, R_t, S_{t+1}, D_t)_{1,\dots,N}$ into corresponding replay buffers $\mathcal{D}_{1,\dots,N}$
    if $t \mod T_{train} = 0$ **then**
      for $k = 1, \dots, N$ **do**
        Train μ_k and Q_k using $\mathcal{D}_k$ buffer as in Equations (13, 14, 15)
    if $t \mod T_{target} = 0$ **then**
      for $k = 1, \dots, N$ **do**
        Update μ^{targ}_k and Q^{targ}_k as in Equation (16)
    $\epsilon \leftarrow \epsilon\epsilon_{decay}$
  if *episode* $\mod E_{update} = 0$ **then**
    Perform federated update for actor and critic neural networks

Algorithm 2: Federated update: Soft Federated Averaging

Input: Parameters of neural networks $\theta_{1,\dots,N}$

$\theta_w \leftarrow \frac{1}{N}\sum_{k=1}^{N}\theta_k$
for $k = 1, \dots, N$ **do**
  $\theta_k \leftarrow (1-\tau)\theta_k + \tau\theta_w$

Positive Weighting

Positive Weighting (PWDDPG) is a possible improvement of the FedAvg algorithm. It attempts to include information based on collected rewards, where better-performing parameters are more important than others, as denoted in Algorithm 3. New input values $r_{1,\dots,N}$ are average rewards of agents collected since the last federated update. Weights are then computed using the softmax function with parameter β, which controls the scale.

Algorithm 3: Federated update: Positive Weighting

Input: Parameters of neural networks $\theta_{1,\dots,N}$ and their averaged rewards since last update $r_{1,\dots,N}$

for $k = 1, \dots, N$ **do**
  $w_k \leftarrow \frac{e^{\beta r_k}}{\sum_{l=1}^{N} e^{\beta r_l}}$
$\theta_w \leftarrow \sum_{k=1}^{N} w_k \theta_k$
for $k = 1, \dots, N$ **do**
  $\theta_k \leftarrow \tau\theta_k + (1-\tau)\theta_w$

Real Weighting

Real Weighting (RWDDPG) is a method similar to PWDDPG, except that the parameter weights can be negative according to their obtained rewards. The ability to use parameters with bad performance as the negative update could provide more expressive power to the algorithm. Thus, improving learning. This approach had created the question: In which way should the received rewards be normalized?

Algorithm 4: Federated update: Real Weighting with absolute normalization

Input: Parameters of neural networks $\theta_{1,\dots,N}$ and their averaged rewards since last update $r_{1,\dots,N}$

for $k = 1, \dots, N$ **do**
  $w_k \leftarrow sgn(r_k)\frac{|r_k|^{\beta}}{\sum_{l=1}^{N} |r_l|^{\beta}}$
$\theta_w \leftarrow \sum_{k=1}^{N} w_k \theta_k$
for $k = 1, \dots, N$ **do**
  $\theta_k \leftarrow \tau\theta_k + (1-\tau)\theta_w$

The algorithm RWDDPG with absolute normalization was introduced, described in Algorithm 4. It is based on the intuition that the sum of weights should be one, which is extended to include negative values. Therefore the sum of absolute weights is normalized to one. Individual weights are computed as normalized absolute reward values with the corresponding sign. The parameter β is added to normalization to control the scaling of rewards.

Global Soft Update

The introduction of a global soft update (GSDDPG) could make the federated update more stable by reducing the sudden change in parameter values. This method could be

applied independently as a substitution for the soft update. Thus, maintaining stability while removing individuality.

The method is implemented through the soft update of previous parameters $\theta_w^{previous}$ with the average of those newly learned $\frac{1}{N}\sum_{k=1}^{N}\theta_k$, as shown in Algorithm 5. Hyperparameter β controls the balance between previously used parameters and those freshly learned. When β is 1, the algorithm acts the same as FLDDPG, and the previous parameter values have no influence on the update. As the value of β is closer to 0, the previous parameters are becoming more influential. Thus the learning becomes more stable until it becomes stagnant when β equals 0.

Algorithm 5: Federated update: Global Soft Update

Input: Parameters of neural networks $\theta_{1,\dots,N}$

$\theta_w \leftarrow (1-\beta)\theta_w^{previous} + \beta\frac{1}{N}\sum_{k=1}^{N}\theta_k$
$\theta_w^{previous} \leftarrow \theta_w$
for $k = 1, \dots, N$ **do**
    $\theta_k \leftarrow \theta_w$

Additional Starting Positions

The learning might be improved by the diversification of tasks. As it is described in Section 3.1, the gathered experiences are only from five different navigation problems. This might lead to undesirable behavior when the agents master how to solve each problem but fail to generalize. Failure to generalize would then decrease the performance in the evaluation phase when the agent is operating in a previously unknown environment.

The indicated issue could be solved by the addition of multiple starting positions (FLS-DDPG) for each robot that would be randomly used during training. This could lead to an increase in the number of different experiences and consequently to improvement in generalization. New starts were added in a way to diversify robot's experiences as much as possible. Two more starting positions were added to smaller playgrounds and three to bigger ones, as can be seen in Figure 10

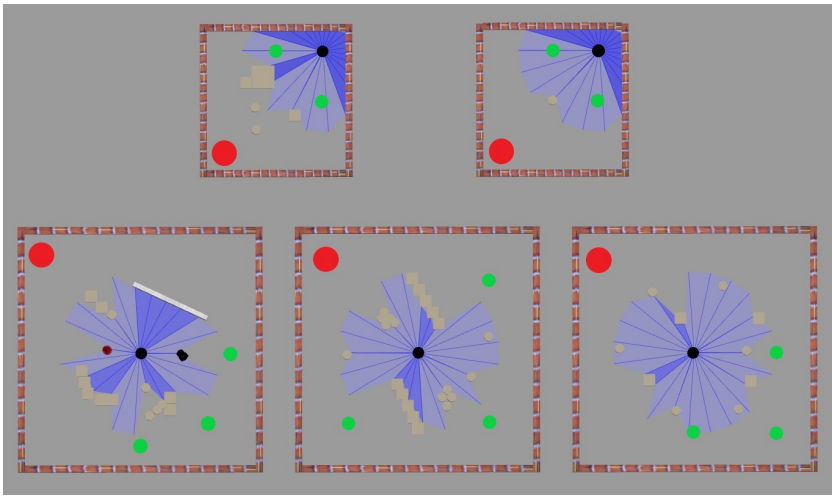


Figure 10: Image of learning playgrounds with additional starting points. Black dots represent the original starting positions of the robots, while green dots are additional starts. The addition of more starting positions could improve the robot's ability to generalize. Red dots represent the goal.

4 Experiments

This chapter describes the conducted experiments. The experiments were not exact copies of those done in [3] because it was not possible to reproduce the stated results while using the same hyperparameter settings, possibly due to differences in simulation implementation. Thus, a new search for better hyperparameters was carried out. New hyperparameters were selected in a way to improve the success rate across all compared methods (IDDPG, SEDDPG, SNDDPG, FLDDPG) as much as possible. Then, an attempt to achieve the same results as in the replicated paper is described. Finally, the performance of the proposed improvements is compared and discussed.

4.1 Hyperparameter Search

The initial hyperparameter search was done ad hoc due to time-consuming training. The replicated paper used a set of six robots and playgrounds while training, but during initial experiments, one of the robots performed so poorly that it stalled and sometimes even ruined the learning process. Therefore, the problematic robot and playground were removed from the simulation. At the same time, the collision penalty was reduced. The layer size of actor and critic neural networks was decreased from 512 to 64 neurons. The critic's architecture was simplified. These changes resulted in improved learning and lower computational cost. Experiments showed that the large improvement was done by slowing the speed of robots from 1 m/s to 0.25 m/s. Thus, the number of steps per episode had to be increased to 1024. Other changes in learning rate α, starting probability of random action ϵ and its decay ϵ_{decay} resulted in faster learning while performance stayed the same. Therefore the number of episodes could be decreased to 125, which reduced training time. The original and used parameter values are listed in Table 2.

Three values were considered for the discount factor $\gamma \in \{0.8, 0.9, 0.999\}$. Each tested value was used in training 24 times - six times for every tested method. During each run, the mean reward and the mean success proportion were collected, which were then averaged across all runs and methods. The mean rewards collected during the last 25 episodes were $1.16, 1.35, 1.13$, respectively. The mean proportions of successful runs were $0.40, 0.49, 0.43$ in the same interval. In both metrics, the value $\gamma = 0.9$ showed the best results, and consequently, it was used in future experiments.

This result shows the importance of a balance between short-term and long-term planning. A low optimal discount factor would mean that the distance reward is sufficient for navigation, and robots can quickly react to near obstacles. On the other hand, a high optimal discount factor would indicate the difficulty in fast obstacle evasion and the necessity of long-term planning.

In a similar way, three target update values $\rho \in \{0.1, 0.5, 1.0\}$ were compared, used as mentioned in Section 2.3. Every value was tested during learning 16 times. Across the last 25 episodes the mean rewards were $1.11, 1.10, 1.31$ and the mean proportions of successful

Table 2: Table with a comparison of hyperparameter values used in the paper [3] and this work. Changes in parameter values were done because the original parameters did not reproduce the wanted behavior, possibly due to differences in simulation implementation.

Parameter	Original value	Used value
The Number of robots, N	6	5
The Number of episodes, E	1 000	125
The Number of steps per episode, T	256	1 024
The Number of transitions stored in the buffer for IDDPG, SEDDPG, SNDDPG, and FLDDPG, respectively, $\lvert\mathcal{D}\rvert$	$\{50\,000, 50\,000, 50\,000, 50\,000\}$	$\{30\,000, 50\,000, 70\,000, 10\,000\}$
Periods		
Training period, T_{train}	5	5
Target update period, T_{target}	5	5
Exploration		
Starting probability of random action, ϵ	0.9	0.9
Decay of random action probability, ϵ_{decay}	0.99995	0.99997
Learning		
Optimizer	*Adam*	*Adam*
Learning rate, α	0.0001	0.001
Discount factor, γ	0.9	0.9
Target update coeficient, ρ	0.01	1.00
Batch size, $\lvert\mathcal{B}\rvert$	512	512

runs were $0.42, 0.33, 0.50$. Value $\rho = 1.0$ - hard target update - was chosen because it outperformed others in both criteria.

The last performed hyperparameter search was to find the optimal experience buffer size $|\mathcal{D}|$. It was necessary to search for optimal buffer sizes independently for each method due to differences in their learning process. Firstly, learning of every method was tested with three values $|\mathcal{D}| \in \{30\,000, 50\,000, 70\,000\}$. Then, the experiments continued with different values according to the previous results to find the value with the maximal proportion of successful runs. Each tested value was used for learning at least 4 times, occasionally even more, when the results were inconclusive. This search had to be thorough because the change in buffer size significantly influenced performance.

Searched buffer sizes for IDDPG were $\{10\,000, 30\,000, 50\,000, 70\,000\}$. Resulting average proportions of successful runs in the last 25 episodes were: $0.26, 0.32, 0.31, 0.24$, respectively. Therefore the buffer size of the IDDPG method was set to $|\mathcal{D}| = 30\,000$. For SEDDPG, only three initial buffer sizes were tested because from their performance could be concluded that the optimal buffer size was $|\mathcal{D}| = 50\,000$. The performance of SNDDPG with buffers of various sizes can be seen in Figure 11. The buffer with a size of 70 000 showed a better success ratio than smaller buffers, combined with steadier learning than the larger buffer. The

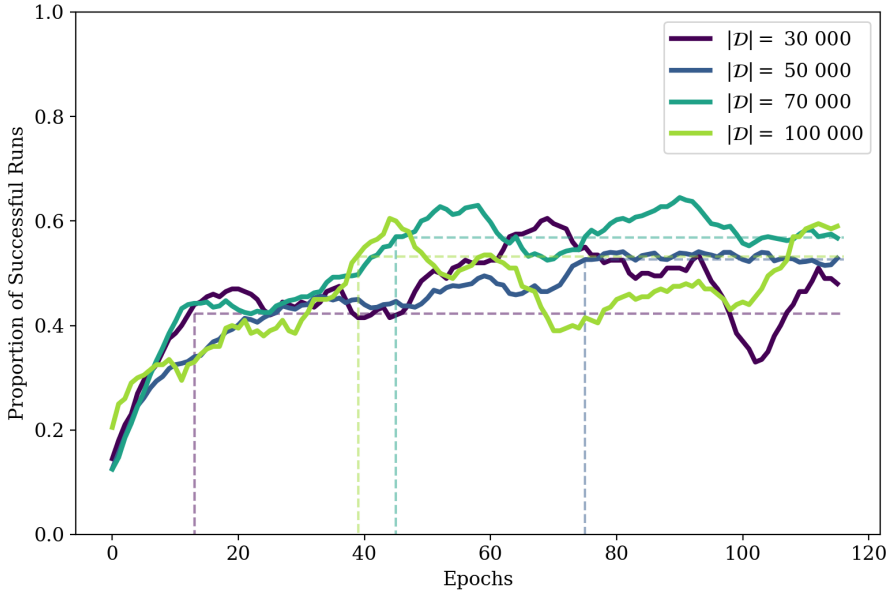


Figure 11: The learning performance of SNDDPG with different experience buffer sizes. The Figure displays the proportion of successful runs of all robots per learning episode. Measured values were smoothed by a 10-epoch simple moving average to make long-term trends more visible. The learning with every tested buffer size was performed at least 4 times. The buffer with a size of 70 000 showed a better success ratio than smaller buffers, combined with steadier learning than the larger buffer.

search for FLDDPG buffer was conducted with sizes $\{5\,000, 10\,000, 30\,000, 50\,000, 70\,000\}$. The performance of the largest buffer was significantly worse than other values. The values 30 000 and 50 000 reached similar results. Previously mentioned sizes were dominated by smallest values - 5 000 and 10 000 - with comparable performance. Hence, several more experiments were conducted between them, which concluded that the optimal size is $|\mathcal{D}| = 10\,000$.

The selected experience buffer sizes corresponded to the used amount of buffers in the method. Smaller buffers with sizes 10 000 and 30 000 performed better in algorithms FLDDPG and IDDPG, where each robot has its buffer. The success of relatively small buffers might originate from frequent training and large batch sizes. At the same time, larger buffers were more successful in methods with only one shared buffer, such as SEDDPG and SNDDPG.

4.2 Replication

In the paper [3], three series of training experiments were conducted. The first one searched for optimal federated learning hyperparameter - federated update period E_{update}. The search showed that lower averaging periods are harmful to learning, while the same can be said about periods set too long. This result agreed with already published research [40]. Another experiment compared the efficiency of the newly proposed soft update with the traditional hard update. The soft update showed faster convergence and did not have temporal decreases in performance after the federated update. Finally, the proposed FLDDPG method was compared to IDDPG, SEDDPG, and SNDDPG already mentioned in Section 2.5. FLDDPG was able to achieve higher rewards during learning and absolutely dominated the evaluation phase.

FedAvg period

The federated update period E_{update} was in previous experiments equal to 1, so it would not be possible to experiment with a smaller period. Thus, step federated update period T_{update} was added to execute the update according to the number of performed steps. Only one of these two update variants was active at once. Therefore, if the T_{update} were equal to 512, the update period would be two times shorter than with E_{update} set to 1.

The tested values were $T_{update} \in \{16, 32, 64, 128, 256, 512\}$ and $E_{update} \in \{1, 2, 3, 5, 10\}$. The learning process of FLDDPG was repeated 8 times for every mentioned period value. The experimental results are displayed in Figure 12 and Figure 13 where the mean proportion of successful runs in the last 25 episodes is shown. The experiments suggest the existence of a complex relationship between the update period and performance. The worst performing periods were the two smallest ones - $\{16, 32\}$. On the other hand, the performance of the largest periods $\{3, 5, 10\}$ was also below-average. These results support the conclusions done in both [3] and [40]. The highest success rate was reached by period values 64 and 2. Period $E_{update} = 2$ was selected as the optimal one when the communication efficiency was also taken into consideration.

Soft Update

The soft federated update was compared with the IDDPG method and the hard update. Tested τ values were $\{0, 0.5, 1\}$ corresponding to IDDPG, soft update, and hard update, respectively. Each tested value was 8 times used in learning to gather information about their performance which is displayed in Figure 14. From collected metrics, it was evident that the soft update significantly improves received rewards and the proportion of successful runs. These improvements are larger than those described in [3] and underscore the usefulness of soft averaging during learning.

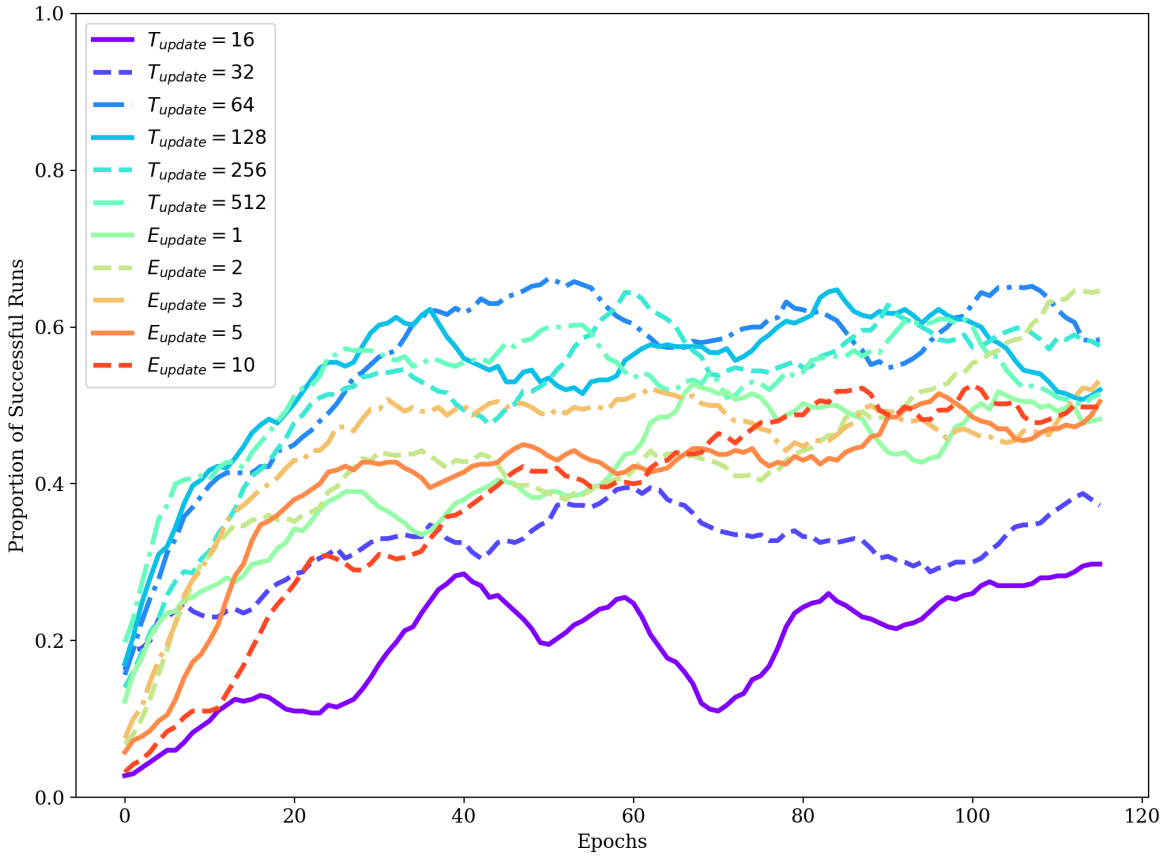


Figure 12: The learning performance of FLDDPG with different federated update periods. The Figure displays the proportion of successful runs of all robots per learning episode. Measured values were smoothed by a 10-epoch simple moving average to make long-term trends more visible. The learning with every tested period value was performed 8 times.

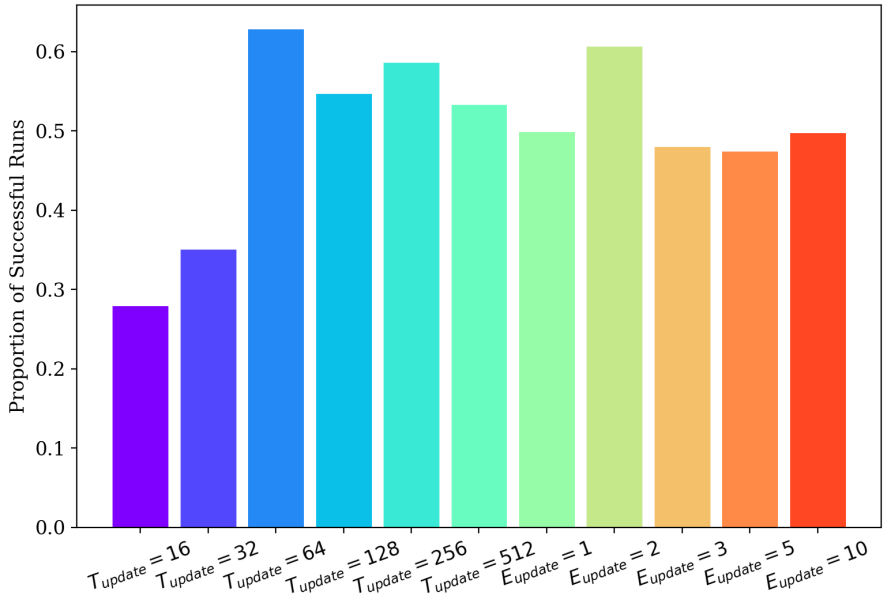


Figure 13: The Figure displays the mean success rates in the last 25 episodes of FLDDPG learning with different federated update periods. The learning with every tested buffer size was performed 8 times. The below-average performance of both the smaller and larger period values supports the conclusions done in [3]. When the number of performed federated updates was taken into consideration, the value $E_{update} = 2$ was selected as optimal.

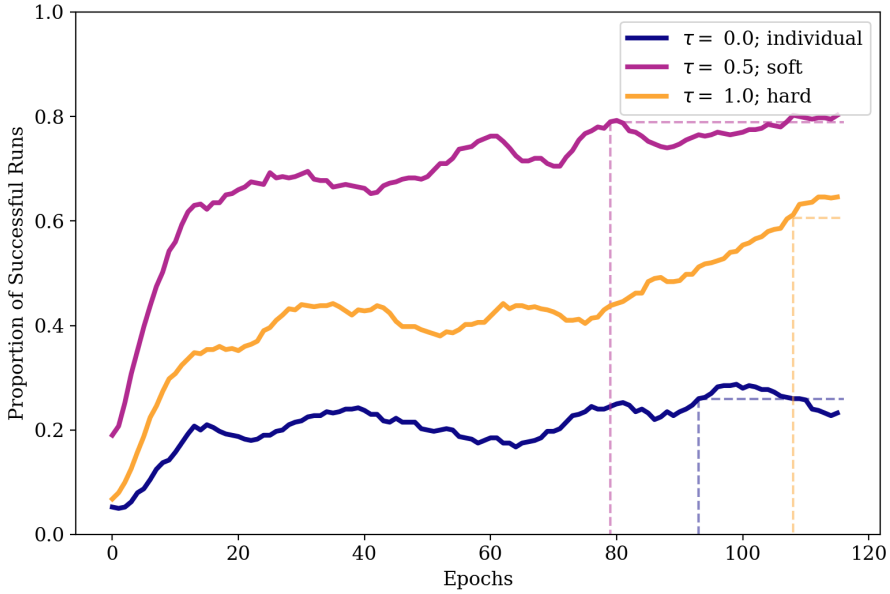


Figure 14: The Figure shows the comparison of the learning performance of the soft update FLDDPG with other methods. The Figure displays the proportion of successful runs of all robots per learning episode. Measured values were smoothed by a 10-epoch simple moving average to make long-term trends more visible. The learning of every compared method was performed 8 times. The results show that the soft update crucially increases the ability to learn.

Comparison with Other Algorithms

After searching for the optimal hyperparameters of the FLDDPG algorithm, it was possible to compare it with the other methods. The experiments for comparison of the learning process were repeated 8 times for each compared algorithm. The performance of algorithms during learning is displayed in Figures 15 and 16. From these figures, it is evident that the IDDPG had the worst learning performance. The SEDDPG and the SNDDPG reached similar values in both measured characteristics. The FLDDPG was most of the time comparable to SEDDPG and SNDDPG in terms of average reward but managed to outperform them in the proportion of successful runs while reaching the average success rate of 0.79 in the last 25 episodes.

The communication efficiency of compared methods is shown in Table 3. IDDPG, by its design, did not require any communication rounds. The SEDDPG method used 13 248 000 rounds, computed by Equation 25 as $((105\,984+0)\,125)$. SNDDPG needed 153 625 communication rounds, computed as $((1\,024+205)\,125)$, to finish learning. The FLDDPG method needed approximately 63 rounds of communication, computed as $((0+0.5)\,125)$. These

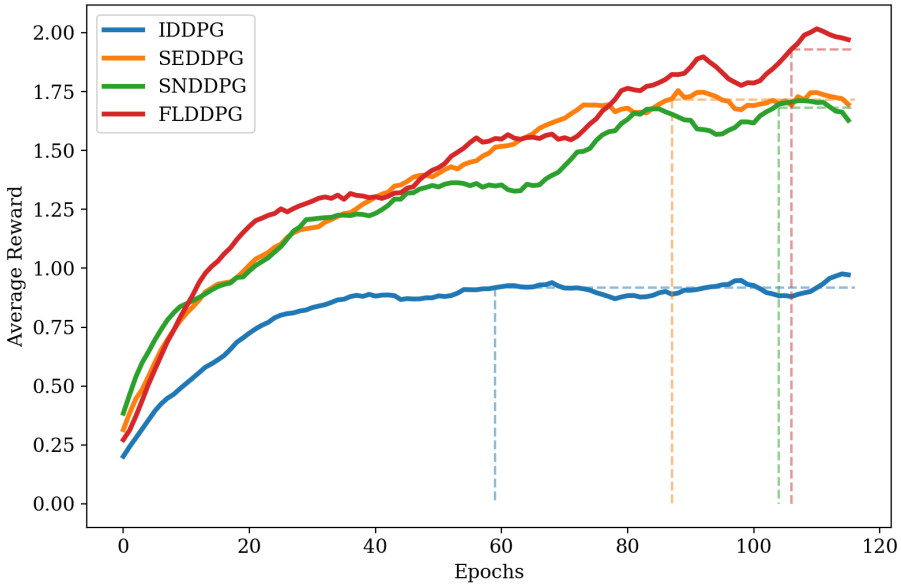


Figure 15: The average rewards obtained during learning of compared methods. Measured values were smoothed by a 10-epoch simple moving average to make long-term trends more visible. During learning, the average rewards received by FLDDPG were comparable and often better than other methods.

results show that FLDDPG significantly reduces the communication cost during learning compared to SEDDPG and SNDDPG.

Table 3: Table with a total number of communication rounds performed during learning by compared algorithms. FLDDPG had a significantly smaller number of communication rounds compared to SEDDPG and SNDDPG.

Method	Communication Rounds
IDDPG	0
SEDDPG	13 248 000
SNDDPG	153 625
FLDDPG	63

The evaluation of each method consisted of 640 evaluation episodes. During learning IDDPG, SEDDPG, and FLDDPG each produced 40 agents, because the learning was repeated 8 times and each run produced 5 agents. Every agent had four attempts to reach the selected goal without collision, this was repeated for every one of the four goals. The learning of SNDDPG produced only 8 agents. Therefore the number of evaluation episodes

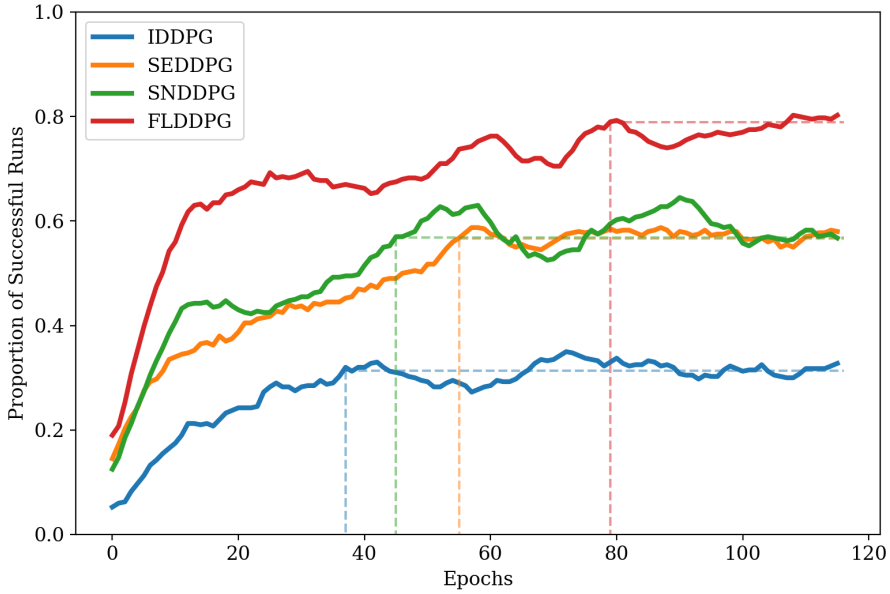


Figure 16: The proportions of successful runs obtained during learning of compared methods. Measured values were smoothed by a 10-epoch simple moving average to make long-term trends more visible. The Figure shows that the FLDDPG with soft averaging outperformed other methods in the proportion of successful runs.

of every agent was increased to 20 so that the total number of attempts stays the same.

From these experiments arose the question: How to calculate methods performance when every run produces several agents? The two approaches were considered. The first one consists of selecting the most successful agent produced by each run and using only its evaluation results. The success rate of best-performing agents across compared methods can be seen in Figure 17. This approach could be considered problematic because the selection is performed based on the data collected during the evaluation, which is in real-life scenarios almost impossible. Therefore the results obtained by this method will not be further discussed. The other approach is to use evaluation results from every one of the trained agents without considering their performance. The results obtained by this method are displayed in Figure 18.

The IDDPG method had the worst proportion of successful runs with a value of 0.1078. More successful was the SEDDPG method, which reached the proportion of 0.1141. The SNDDPG method was the most successful, with a rate of 0.2859. The FLDDPG with soft averaging did not manage to realize its great performance during learning and reached the proportion of 0.2516.

In terms of the average time of successful runs, the IDDPG method had a time of 22.11 ± 4.82 s which was the slowest out of all compared methods. The average time of

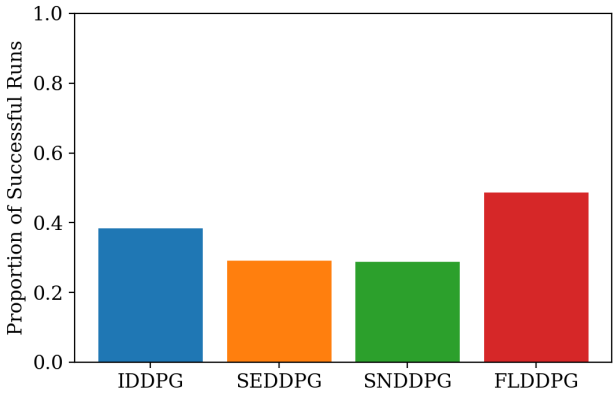
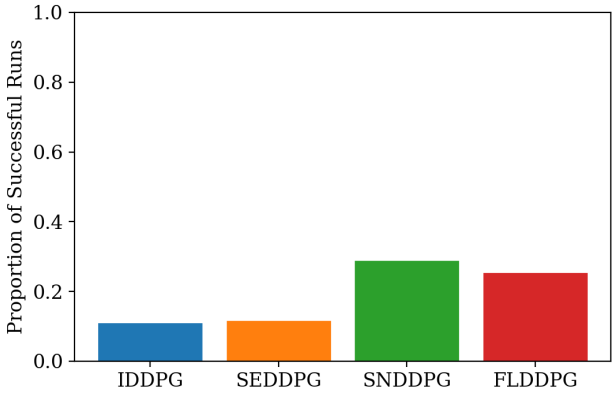


Figure 17: The proportion of successful runs during the evaluation using best-performing agents. Results obtained only by best-performing agents are not further discussed because in real-life scenarios their selection would be almost impossible.

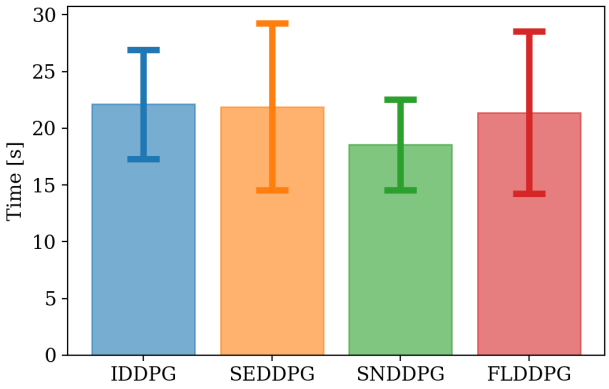
21.90 ± 7.37 s was reached by SEDDPG. The SNDDPG was the fastest, with a time of 18.55 ± 4.00 s. FLDDPG had an average time of 21.39 ± 7.13 s.

The trajectory efficiency of the IDDPG algorithm was 0.8539 ± 0.1091, which was once again the worst out of all methods. The SEDDPG method obtained the efficiency of 0.8759 ± 0.984. The most efficient average trajectory was 0.9341 ± 0.0217, which was achieved by the SNDDPG. FLDDPG had an efficiency of 0.8704 ± 0.1032.

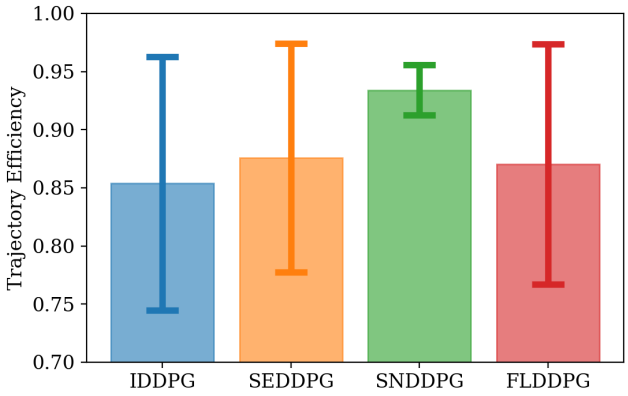
The above-mentioned results show the dominance of SNDDPG in almost all evaluated categories. The soft averaging FLDDPG method managed to outperform the IDDPG and the SEDDPG methods in mean success. The significant discrepancy between learning and evaluation performance might be caused by the algorithm's problems with generalization. This problem could be amplified by the usage of the soft update, which favors individuality. The evaluation outcomes are in conflict with the outstanding performance described in the replicated paper [3]. On the one hand, the perceived difference could derive from differences in implementation and changes in hyperparameters. On the other hand, the main principle of learning to navigate a previously unknown environment stayed the same.



(a)



(b)



(c)

Figure 18: The evaluation results of compared methods across all trained networks. (a) The proportion of successful runs during evaluation. (b) The average time to reach the goal in successful runs. (c) The trajectory efficiency of paths taken by robots, which reached the goal.

4.3 Results of Proposed Improvements

The following subchapter discusses the experiments done with methods proposed in Section 3.2. Each method has a section where its learning and evaluation results are described and possibly explained. The learning performance of the proposed algorithms can be seen in Figure 19, while the evaluation results are shown in Figure 20.

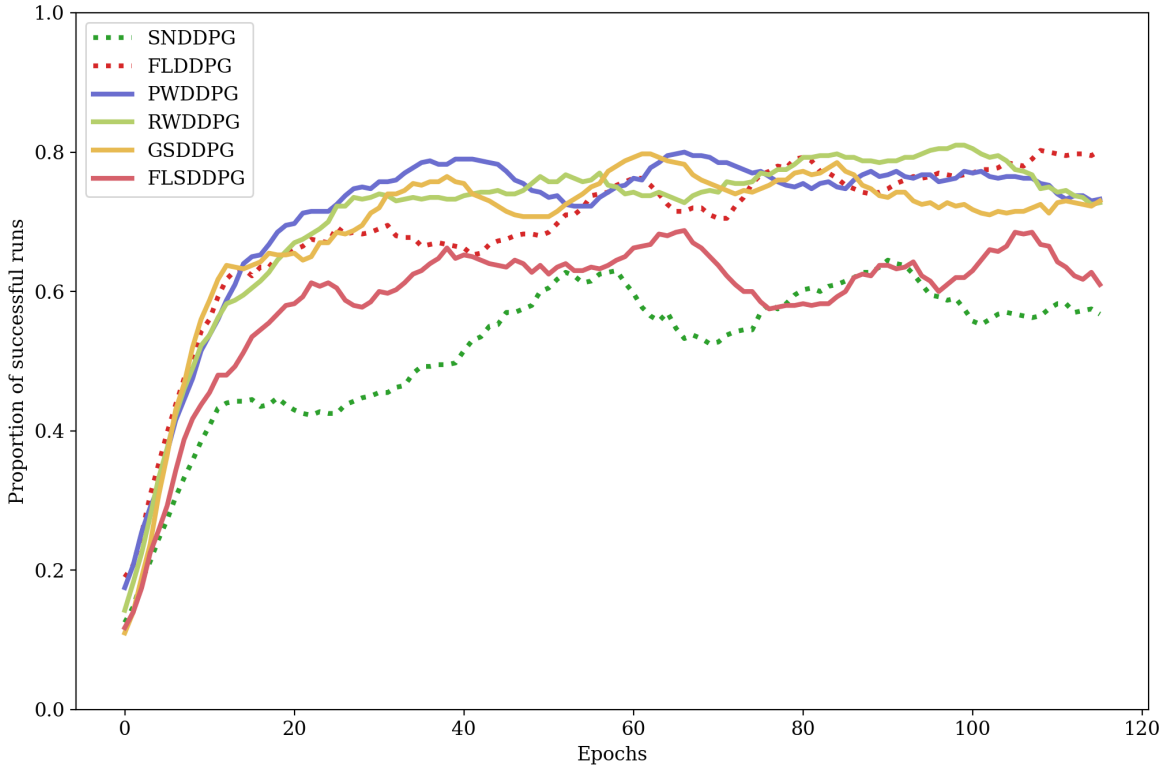
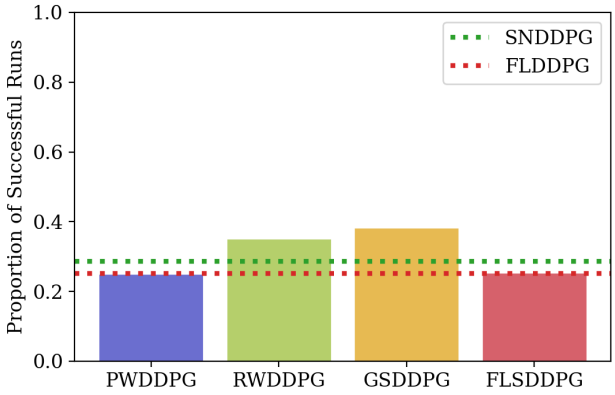


Figure 19: The proportions of successful runs obtained during learning of the proposed improvements. Measured values were smoothed by a 10-epoch simple moving average to make long-term trends more visible. For comparison, the learning results of the best performing method in learning (FLDDPG) and evaluation (SNDDPG) were displayed as dotted lines.

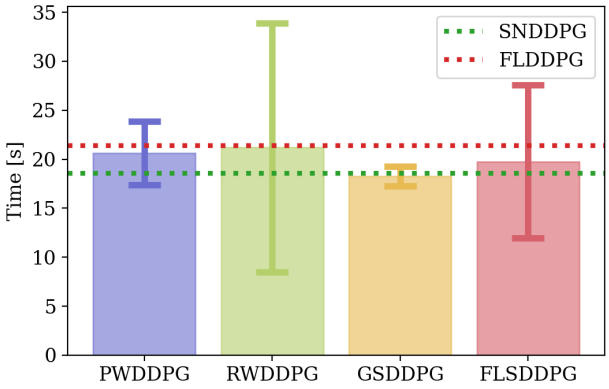
Positive Weighting

Before comparison of PWDDPG to other methods, the optimal value of parameter β had to be found. The learning was repeated 8 times for every one of the tested values $\{0.25, 0.5, 1.0, 1.5\}$. The best results were achieved with β equal to 0.5 when the learning quickly converged, and the mean success rate in the last 25 episodes was 0.7527. The number of communication rounds doubled compared to FLDDPG to 126 because the rewards were also collected.

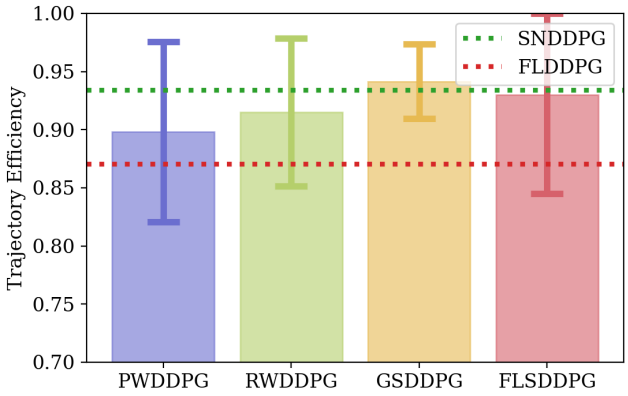
During the evaluation, the PWDDPG managed to reach a goal at an average rate of 0.2467. The mean time of a successful run was 20.63 ± 3.23 s and trajectory efficiency was 0.8981 ± 0.0776. These results suggest that the proposed PWDDPG slightly decreased the success rate of FLDDPG while somewhat improving the other two metrics.



(a)



(b)



(c)

Figure 20: The evaluation results of the proposed improvements. (a) The proportion of successful runs during evaluation. (b) The average time to reach the goal in successful runs. (c) The trajectory efficiency of paths taken by robots, which reached the goal. For comparison, the evaluated results of the best performing method in learning (FLDDPG) and evaluation (SNDDPG) were displayed as dotted lines.

Real Weighting

Firstly, the search for the optimal β parameter was conducted. The learning was performed 8 times for each of values $\{0.25, 0.5, 1.0, 1.5\}$. The optimal learning performance was achieved by the value of 0.5, which was the most stable and obtained the best mean success rate in the last 25 episodes of 0.7627. The number of communication rounds doubled compared to FLDDPG to 126 because the robot's rewards were also transmitted.

In evaluation, the RWDDPG reached the success rate of 0.3484. The mean run time

was 21.19 ± 12.71 s and the trajectory efficiency was 0.9150 ± 0.0635. The success rate of RWDDPG outperformed both the FLDDPG and the SNDDPG algorithms. The proposed method had also better trajectory efficiency than FLDDPG.

The performance disparity between PWDDPG and RWDDPG can be explained by their differences in weight computation. The PWDDPG method uses an exponential function, which transforms relatively minor differences in rewards into large ones in weights. This property could lead to the dominance of the most successful agent, thus complicating the learning of others. The success of RWDDPG might be caused by the induced similarity of network parameters. When the β parameter is smaller than 1, the proportionally largest differences are obtained by smaller reward values. Therefore, at the beginning of the learning, the most successful network dominates the computed parameter average. And in the later stage, the weights are more proportionate, because the agents are receiving higher rewards. This forces all networks to have more similar parameters and combines individuality with experience sharing.

Global Soft Averaging

The GSDDPG method was tested with β parameter value set to 0.5, similarly to FLDDPG with the parameter τ. During the first half of the learning, the GSDDPG performed better than FLDDPG, but later the trend reversed. In the last 25 episodes, the mean success rate was equal to 0.7198. The communication efficiency stayed the same as in FLDDPG.

The mean success rate of the GSDDPG during the evaluation was 0.3797. The success time corresponded to 18.26 ± 0.98 s and the trajectory efficiency was 0.9415 ± 0.0321. Therefore, the GSDDPG had the best performance in all of the measured metrics out of all compared methods.

The success of the GSDDPG method suggests that the individuality introduced by the soft update in FLDDPG might not be an essential part of the algorithm. It also provides evidence that the less radical changes in parameters brought by the global soft average improve the method's ability to generalize.

Additional Starting Positions

During learning, the FLSDDPG method was considerably worse than the other methods, except the SNDDPG. The mean success rate in the last 25 episodes was 0.6498. This decrease in the learning performance could be expected when the addition of starting positions creates a more complex scenario. The communication efficiency stayed the same as in FLDDPG.

The FLSDDPG achieved in the evaluation the mean success ratio of 0.2500, which is only slightly lower than the performance of FLDDPG. On the other hand, it reached the average time of 19.77 ± 7.81 s and the trajectory efficiency of 0.9298 ± 0.0849, thus reaching better performance in both of these criteria than the FLDDPG.

The additional starting positions for the FLDDPG method did not manage to significantly improve the evaluation success. This failure could be caused by the selection of hyperparameters, which were designed to be used in the specific learning scenario. Thus the newly collected experiences could not be leveraged to improve the method's performance.

5 Conclusion

This work focused on the use of federated learning in combination with deep reinforcement learning to complete the task of robotic navigation with restrained sensory equipment. It attempted to replicate the results stated in the paper Federated Reinforcement Learning for Collective Navigation of Robotic Swarms, which introduced a new learning method by combining the federated learning with a soft weight update and Deep Deterministic Policy Gradient.

The scenario used for learning and evaluation was replicated. The five robots *TurtleBot3* jointly learned to navigate to a goal in different playgrounds with increasing complexity. During the evaluation phase, one robot navigated to four different goals in the environment, which was more complex than those during learning. Due to differences in the simulation, the learning did not work with the same hyperparameters as used in the original paper. Therefore, new optimal hyperparameters were found, such as learning rate, robot speed, and the number of transitions stored in the buffer.

Then, the learning and the evaluation performance of the federated algorithm were compared to the results of three other methods. The replicated algorithm did not manage to reach the expected success. Four different improvements to the federated method were introduced. Two of them improved the evaluation performance and even outperformed other current methods in almost every metric without significantly decreasing the communication efficiency.

In future work, it would be beneficial to test the performance of proposed methods on different reinforcement learning scenarios to collect more diverse information. This additional information could be leveraged to make more rigorous conclusions about proposed improvements. It also might be interesting to experiment with combining the proposed methods with federated learning with a hard update to obtain a deeper understanding of their behavior.

References

[1] Chen Lei. *Deep Reinforcement Learning*, pages 217–243. Springer Singapore, Singapore, 2021.

[2] Qiang Yang, Yang Liu, Yong Cheng, Yan Kang, Tianjian Chen, and Han Yu. *Federated Learning*. Morgan & Claypool, 2019.

[3] Seongin Na, Tomáš Krajník, Barry Lennox, and Farshad Arvin. Federated reinforcement learning for collective navigation of robotic swarms, 2022.

[4] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[5] Joshua Achiam. Part 1: Key concepts in rl, Jan 2020.

[6] Ben Hambly, Renyuan Xu, and Huining Yang. Recent advances in reinforcement learning in finance. 2021.

[7] RBC Capital Markets. RBC Capital Markets launches Aiden® – a new ai-powered electronic trading platform, 2020.

[8] OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving rubik's cube with a robot hand, 2019.

[9] Andrew Lobbezoo, Yanjun Qian, and Hyock-Ju Kwon. Reinforcement learning for pick and place operations in robotics: A survey. *Robotics*, 10(3), 2021.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[11] Dan Cireşan, Ueli Meier, and Juergen Schmidhuber. Multi-column deep neural networks for image classification, 2012.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[13] Michael Auli, Michel Galley, Chris Quirk, and Geoffrey Zweig. Joint language and translation modeling with recurrent neural networks. 2013.

[14] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[16] Daniel W. Otter, Julian R. Medina, and Jugal K. Kalita. A survey of the usages of deep learning for natural language processing. *IEEE Transactions on Neural Networks and Learning Systems*, 32(2):604–624, 2021.

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[18] Davide Chicco. Ten quick tips for machine learning in computational biology. *BioData Mining*, 10(1):35, December 2017.

[19] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, and et al. Highly accurate protein structure prediction with alphafold, Jul 2021.

[20] Ivo Grondman, Lucian Busoniu, Gabriel A. D. Lopes, and Robert Babuska. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1291–1307, 2012.

[21] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula, 2020.

[22] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, pages 1–18, 2021.

[23] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[24] Joshua Achiam and Miguel Morales. Deep deterministic policy gradient, Jan 2020.

[25] Peter Kairouz, H. Brendan McMahan, and et al. Advances and open problems in federated learning, 2021.

[26] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd, 2017.

[27] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data, 2017.

[28] Jie Xu, Benjamin S. Glicksberg, Chang Su, Peter Walker, Jiang Bian, and Fei Wang. Federated learning for healthcare informatics. *Journal of Healthcare Informatics Research*, 5(1):1–19, November 2020.

[29] Mohammed Aledhari, Rehma Razzak, Reza M. Parizi, and Fahad Saeed. Federated learning: A survey on enabling technologies, protocols, and applications. *IEEE Access*, 8:140699–140725, 2020.

[30] Tianlong Yu, Tian Li, Yuqiong Sun, Susanta Nanda, Virginia Smith, Vyas Sekar, and Srinivasan Seshan. Learning context-aware policies from multiple smart homes via federated multi-task learning. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 104–115, 2020.

[31] Chetan Nadiger, Anil Kumar, and Sherine Abdelhak. Federated reinforcement learning for fast personalization. In *2019 IEEE Second International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, pages 123–127, 2019.

[32] Mohammad Reza Samsami and Hossein Alimadad. Distributed deep reinforcement learning: An overview, 2020.

[33] Xuesu Xiao, Bo Liu, Garrett Warnell, and Peter Stone. Motion control for mobile robot navigation using machine learning: a survey, 2020.

[34] Bashan Zuo, Jiaxin Chen, Larry Wang, and Ying Wang. A reinforcement learning based robotic navigation system. In *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3452–3457, 2014.

[35] Boyi Liu, Lujia Wang, and Ming Liu. Lifelong federated reinforcement learning: A learning architecture for navigation in cloud robotic systems. *IEEE Robotics and Automation Letters*, 4(4):4555–4562, 2019.

[36] Lei Tai, Giuseppe Paolo, and Ming Liu. Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 31–36, 2017.

[37] Enrico Marchesini and Alessandro Farinelli. Discrete deep reinforcement learning for mapless navigation. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10688–10694, 2020.

[38] Haobin Shi, Lin Shi, Meng Xu, and Kao-Shing Hwang. End-to-end navigation strategy with deep reinforcement learning for mobile robots. *IEEE Transactions on Industrial Informatics*, 16(4):2393–2402, 2020.

[39] Khaled Alaa, Nicolò Botteghi, Beril Sirmacek, Mannes Poel, and Stefano Stramigioli. Towards continuous control for mobile robot navigation: A reinforcement learning and slam based approach. 05 2019.

[40] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. 2016.